

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of “Eiffel: The Language”. Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

Bertrand Meyer

*Eiffel Software (California)
ETH (Zurich)*

STANDARD EIFFEL

(Eiffel: The Language, third edition)

**Compatible with ECMA International Standard 367
*(“Eiffel Analysis, Design and Programming Language”)***

Author's addresses:

Bertrand Meyer

Eiffel Software
356 Storke Road
Goleta, CA 93117 USA
+1-805-685-1006, fax +1-805-685-6869
<meyer@eiffel.com>, <http://eiffel.com>

ETH Zurich, Department of Computer Science
CH-8092 ETH-Zentrum, Switzerland
+41-44-632-0410, Fax +41-44-632-1435
<http://se.inf.ethz.ch>

Must it be assumed that because we are engineers beauty is not our concern, and that while we make our constructions robust and durable we do not also strive to make them elegant?

Is it not true that the genuine conditions of strength always comply with the secret conditions of harmony?

The first principle of architectural esthetics is that the essential lines of a monument must be determined by a perfect adaptation to its purpose.

Gustave Eiffel, 1887

From his response in the newspaper Le Temps to a petition by prominent artists and writers protesting his project of elevating a tower of iron in Paris.

Short contents

(The full table of contents starts on page [xxv](#).)

Preface: Meeting the challenge of software quality (<i>in progress</i>)	vii	29 Constants	787
Preface to the third edition	xii	30 Basic types	815
About the status of Eiffel	xiii	31 Interfacing with C, C++ and other environments	823
About the language description	xv	32 Lexical components	879
Contents	xxv	33 Concurrency (<i>not done</i>)	905
PART I: INVITATION TO EIFFEL	1	34 Style guidelines (<i>not done</i>)	907
1 An Eiffel tutorial	3	PART III: KERNEL LIBRARY CLASSES	925
PART II: LANGUAGE DESCRIPTION	83	35 Universal features and class <i>ANY</i> (<i>in progress</i>)	927
2 Syntax, validity and semantics	85	36 Arrays and strings (<i>not done</i>)	931
3 The architecture of Eiffel software	105	37 Persistence (<i>not done</i>)	941
4 Classes	115	38 Input and output (<i>not done</i>)	965
5 Features	131	A ELKS: The Eiffel Library Kernel Standard	971
6 The inheritance relation	169	PART IV: THE LACE CONTROL LANGUAGE	1015
7 Clients and exports	191	B Specifying systems in Lace (<i>in progress</i>)	1017
8 Routines	217	PART V: COMPLEMENTS	1039
9 Correctness and contracts	229	C On language design and evolution	1041
10 Feature adaptation	261	D Credits (<i>in progress</i>)	1055
11 Types	323	E A brief history of Eiffel	1061
12 Genericity	349	F Language changes from the previous edition	1063
13 Tuples	371	G Changes from early versions	1077
14 Conformance	383	H An Eiffel tutorial	1087
15 Convertibility	399	I Eiffel bibliography (<i>not done</i>)	1089
16 Repeated inheritance	433	PART VI: REFERENCE	1095
17 Control structures	477	K Syntax in alphabetical order	1143
18 Attributes	499	L Reserved words, special symbols, operator precedence	1153
19 Objects, values and entities	505	M Syntax diagrams (<i>not done</i>)	1157
20 Creating objects	523	PART VII: THE LANGUAGE STANDARD	1161
21 Comparing and duplicating objects	565	PART VIII: BACK MATTER	163
22 Attaching values to entities	587	Index	165
23 Feature call	621		
24 Eradicating void calls	657		
25 Typing-related properties	665		
26 Exception handling	689		
27 Agents, iteration and introspection	719		
28 Expressions	761		

Preface: Meeting the challenge of software quality *(in progress)*

Eiffel embodies a “certain idea” of software construction: the belief that it is possible to treat this task as a serious engineering enterprise, whose goal is to yield quality software through the careful production and continuous enhancement of parameterizable, scientifically specified reusable components, communicating on the basis of clearly defined contracts and organized in systematic classifications.

Such aims lead to a *new culture* of software development, focusing on the reuse of industrial-grade components, on the development of complete systems rather than just programs, and on the long-term investment in tools and libraries, capturing the software know-how of an organization. Even more importantly, they mean accepting the challenge of quality in software.

Eiffel is nothing else than these principles taken to their full consequences. In particular, the engineering of quality software components requires an appropriate notation; this book describes such a notation – Eiffel as a language for analysis, design and implementation.

A language, of course, is not enough. To achieve quality and move to the new culture, we must understand the methodological background; we must have access to a large body of good pre-built reusable components; and we need the appropriate development tools. For several years, the work on Eiffel has been proceeding in all of these directions, resulting in a method, a language, a set of libraries, and software development environments.

What you will find below is a description of the language part of Eiffel. Other books cover the complementary aspects: *Object-Oriented Software Construction* presents the method in detail; *Eiffel: The Libraries* and *Eiffel: The Environment* describe the required library and tool support.

For the precise reference to the books mentioned here, see appendix I “An Eiffel Bibliography”, page
====

THOUGHT AND EXPRESSION

Inaccurate as it would be to consider language issues only, the reverse mistake is just as damaging.

A strange idea has become prevalent in some software circles: the claim that languages, after all, are not that important. Requests are even heard here and there for a bizarre animal, the “language-independent methodology” – about as useful as a bird without wings, and just as likely.

In software perhaps more than anywhere else, thought is inseparable from its expression. To obtain good software, a good notation is not sufficient; but it is certainly necessary.

OLD, NEW AND OUT

The language concepts in Eiffel are not, of course, entirely new. Some key ideas came from earlier designs, most notably Simula 67, Algol variants (especially Algol W), Alghard, CLU, Ada, and the early (pre-Oxford) versions of the Z specification language. Among the novel aspects are a number of language constructs — in the areas of inheritance, typing, exceptions, assertions, information hiding and higher-level functions among others — as well as the choice of concepts from various sources and their combination into a coherent edifice. In particular, Eiffel is original in its association of an assertion facility (coming from work on program verification and formal specification) with a full object-oriented approach emphasizing multiple and repeated inheritance and information hiding.

Also notable is the set of ideas that have *not* been retained. To design is to renounce. Although designers rarely speak about this aspect, an engineering product is defined by what it has excluded as much as by what it has retained. In Eiffel, where so much attention was devoted to keeping the language small and trying to make it elegant, readers will be surprised, perhaps shocked in some cases, not to find such broadly accepted concepts as global variables, enumeration types, subrange types, goto instructions (as well as the many disguises that have been invented for them over the years, such as “exit”, “break”, “continue” and the like), routine pointers, undisciplined type conversions (“casts”), in-class overloading, pointer arithmetic, language-defined input and output, and even the notion of main program.

The exclusion of these concepts does not mean that all of them are intrinsically bad, although in light of what we know today about software engineering some obviously are; others are simply redundant or incompatible with mechanisms that were more important for the Eiffel software development method.

THE SIGNAL AND THE NOISE

Simple does not mean simplistic. The language definition, as it appears in this book, includes a few notions that are definitely non-trivial, particularly the full feature adaptation mechanism, repeated inheritance, and the details of type checking. Several reasons justify the presence of these more advanced elements: only a small number of constructs are involved; the basic ideas are straightforward, and the more difficult aspects simply result from pushing the basic ideas to their full consequences; simple uses will yield the expected results, according to the “principle of least surprise”; there is no need to understand all the details for ordinary use (and you will indeed find `SHORTCUT` signs inviting you to skip the more specialized sections); and the extra power granted by the language’s most sophisticated facilities, far from being mere gadgetry, addresses some of the most difficult issues of large-scale software development.

In other words, what the language design has sought is the highest possible *signal-to-noise* ratio, allowing users of the language to make the most out of their intellectual energy. This means getting rid of the noise (features which make the language bigger without contributing any really new concept): who needs three forms of loop when a general enough one will do, or special syntax for array access when we can simply view arrays as “container” structures described by a library class? But it also means reaching for the highest possible level of signal: including the appropriate constructs to deal with the truly difficult cases of software development, especially those which arise in the construction of large and ambitious systems.

When the time comes to decide what is essential and what is superfluous, there is no substitute for experience with the language in many projects and application areas. Interactive Software Engineering has been an extensive user of Eiffel for a decade and a half. Other than interfaces to other tools — viewed by the Eiffel side as “external software” according to the techniques of chapter 31 — all our developments, including ISE’s Eiffel compiler-interpreter and the supporting environment, are in Eiffel. In addition to our own practice, we have observed users of our tools in their development of systems ranging over most application areas of computers.

Appendix C draws on the Eiffel experience to discuss the issues of language design and evolution.

Such experience, although not a guarantee against mistakes, provides crucial background for the choices that await language designers.

LANGUAGE LEVEL

If you have read previous publications about Eiffel, including *Object-Oriented Software Construction*, you will notice some differences with the language described here. This book indeed presents Eiffel version 5, which benefits from the accumulated experience mentioned above. The differences with previous versions do not affect the fundamental semantics of Eiffel; rather, they bring in simplifications in some areas and a few extensions in others. Although long-time Eiffel users and enthusiasts may at first be surprised by some of these changes, I hope they will soon realize that this update brings local but significant improvements to the language, without impairing the consistency and simplicity of its basic design.

A list of changes may be found in an [appendix](#), complemented by [another](#) recalling earlier changes from Eiffel 2 to Eiffel 3.

[F, Language changes from the previous edition and G, Changes from early versions.](#)

Language evolution is a delicate issue; one must walk a fine line between unjustified upheaval and undue conservatism. Yet another [appendix](#) — useful reading if you wish to understand Eiffel in some depth — discusses this question

[C, On language design and evolution.](#)

THE FUTURE OF EIFFEL

The publication of the first edition of this book marked a turning point in the evolution of Eiffel: the passage from single to multiple sources and from individual control to collective oversight.

Until then, one company (Interactive Software Engineering) was the sole supplier of Eiffel compilers. The book's publication enabled more people to take a part. D.W. Barron explained many years ago the danger of such a collective approach:

As a working hypothesis we can suggest that the probability of achieving clearly seen and sensible objectives in language design is in inverse proportion to the number of people involved in the design process. Thus the best languages are those designed by a single individual or a small coherent group. The worst languages are those "designed" by large committees.

D.W. Barron, "An Introduction to the Study of Programming Languages", Cambridge University Press, 1977, page 144. Slightly abridged.

Eiffel users have indeed frequently stated their fears about committee-controlled evolution, particularly the propensity of committees to indulge in what was called gadgetry above and is also known as "featurism": repeated addition of special-purpose facilities, which may individually please specific constituencies, but will collectively destroy the consistency of the design. The risk exists, but perhaps less than with some other language, as the Eiffel community may be trusted to understand the virtues of design simplicity. Besides, the original designer might still be around for

some time, as a kind of Commendatore's *statua gentilissima* ready to intervene at the earliest sign of debauchery.

MISSING ELEMENTS

In two areas in which I would have liked this book to go further.

One is the old but still thorny problem of numerical precision in floating-point computation, for which Eiffel relies on quite traditional solutions, with imprecisely defined semantics. A rigorous approach to this problem, in line with the systematic treatment of other aspects in Eiffel, should be possible, but will require the collaboration of experts in numerical analysis as well as software engineering.

The other missing part is concurrency. There is in fact a language design for concurrency in Eiffel, based on a simple extension (one keyword) and consistent with the theory underlying Eiffel ("Design by Contract"). Yet I stopped short of including the description of this mechanism here because as of this writing it has not yet been implemented. No official programming language document should ever be published unless all major components of the language have been successfully implemented – a principle applied throughout the evolution of Eiffel and, before it, to just about every successful language. (The counter-examples, languages that attracted considerable interest upon publication but then failed because they proved too hard to implement, are all too familiar.)

ACKNOWLEDGMENTS

So many people have helped with the development of the language that I have decided to go beyond the usual "acknowledgments" section and include a special appendix listing all the contributions that I was able to remember, including in particular the names of people who invented specific Eiffel constructs and mechanisms. *Appendix D.*

Eiffel would not exist today without the support and enthusiasm of its users. My biggest debt is to all the software developers the world over who have chosen to rely on Eiffel for their projects. I thank them for their trust and hope that this book will be up to their exacting standards of quality.

Santa Barbara, Zürich, Melbourne B.M.
June 2002

Preface to the third edition

NON-CLASSICAL NUMBER THEORY

A practical note about why this book is called a “third edition”. A year after the initial printing, a number of revisions were made, leading to a second *printing*; this became the reference, and the contents remained the same in the many printings that followed. Although substantial, the second printing’s changes were not numerous enough to justify calling the result a second *edition*. Technically, then, the present volume is a second edition. But imagine the endless confusions if it were called that way: “*I see no mention of the Precursor construct in ETL! — Check the second edition. — But I have the second edition! — Are you sure? — Well, here it says on page xii: «Second revised printing» — Oh, but what you want is the second edition — What do you mean? How do I tell the difference?»* and so on. To avoid such silliness I insisted that this edition be called third, leaving only one opportunity for (harmless) confusion: some people, on seeing the title, might think that they have missed an edition. They have not.

MIGRATION AIDS

Language evolution is a natural phenomenon and the users of previous versions should be helped in the transition to the new version. An appendix gives a detailed list of the differences between Eiffel 5 and the previous version, listing in particular the precise changes in validity constraint, rule by rule and clause by clause. This should be of primary interest to compiler writers. For language users, the attraction lies in new facilities such as agents, tuples, creation expressions, generic creations and assignment procedures; in relaxed constraints (permitting for example even more “freedom” for “free operators”); and in a general cleaning-up of the semantics. The transition should be smooth since Eiffel 5 introduces very few incompatibilities with the previous version.

About the status of Eiffel

The design of the Eiffel language is standardized by ECMA, an international standards organization based in Geneva. The standards favors the spread of Eiffel by enabling interoperability between Eiffel tools and implementations. Such tools should adhere to the standard and acknowledge ECMA; please contact ECMA for the details.

The text makes occasional references to the Eiffel implementation of Eiffel Software: EiffelStudio, the portable development environment available on all major industry platforms, and Eiffel EnVISION!, the verion for Visual Studio .NET. Occasional references also appear to the associated libraries EiffelBase (data structures and algorithms) and For information about these and other products and services of Eiffel Software, including Eiffel training and consulting, see <http://eiffel.com>.

About the language description

In a book describing the language support for a radically new approach to software construction, it was natural to take a fresh look not only at the subject matter but also at the techniques used to present it.

To help you enjoy the discussion of Eiffel, here are a few notes about the description method, and about the reasons that led to it.

TYPES OF DESCRIPTION, LEVELS OF DISCOURSE

The describer of a programming, design or analysis language faces an interesting task. He must address several constituencies: interested bystanders, novices starting to use the language, experienced users, authors of compilers and interpreters. He must satisfy several requirements: explaining the concepts in a clear way; teaching the use of the language; giving examples; providing a precise reference to answer questions of details and remove possible ambiguities or contradictions.

The almost universal response to these conflicting goals is to write at least two documents: a *user's manual* and a *reference manual*, the latter also called "report". The user's manual is supposed to be readable by ordinary human beings; the reference manual must be precise, accurate and, as the accepted consequence, hard to read and boring. The reference manual is indispensable, however, for implementors, and users may also need it when they run into tricky questions or apparent ambiguities. Often, there is a third document, a tutorial.

Having experienced this division, on the reader's side of the fence, when using programming languages as well as software tools (for which the same problems arise, usually with the same solutions), I have come to dislike it profoundly. In spite of all the good intentions that justify the multi-document structure, the net result is that, in any serious use of the language or tools, you keep shuffling between the documents involved

when seeking information about the available features and their properties. Sometimes you will look at the user's manual and fail to find the detailed information that you need; but when you look at the reference manual you miss the necessary informal explanations, comments and examples. If, as is too often the case, you end up looking in both places, you find repetitions, which waste your time, or even apparent contradictions.

This book innovates by being both a reference and a user's manual, and by addressing the needs of both users (beginning or advanced) and implementors (authors of compilers or interpreters). It also includes a tutorial overview.

Of course, by trying to cater to several audiences, it may fail to satisfy any of them. This will be for the reader to judge.

FORMALITY

To compound the difficulty, the "reference" parts of the presentation, intended to serve as official answer to questions about the syntax, validity rules and semantics of language constructs, are more formalized than in most existing language references. These official elements have to coexist with much more informal explanations, comments, examples and advice.

Not that this work can claim to offer a truly *formal* description of Eiffel; this would have required a mathematical specification based on the methods of axiomatic or denotational semantics (which I have tried to summarize, at an introductory level, in another book). The aims here are more modest, but every effort has been devoted to specifying Eiffel with as much precision as can be afforded without resorting to mathematics. An earlier article argued that one of the important byproducts of writing formal specifications may be informal descriptions of a new and better kind, *derived from* the formal ones, which they serve as accompaniment and running commentary. Although no such derivation was used here, readers familiar with formal methods will recognize their influence and a style which resembles what one could find in a non-formal specification resulting from a detour through mathematical formality.

See the book "Introduction to the Theory of Programming Languages", Prentice-Hall International, 1990.

The article mentioned is "On Formalism in Specifications", IEEE Software, 2, 1, January 1985, pages 6-26, especially page 24 and figure 5.

For an example of a language specified in a formal way, see R. C. Holt et al., "The Turing Programming Language", Prentice-Hall International, 1988.

As an example of the effort made to achieve more precision than is customary, most of the validity constraints (rules governing such aspects as type compatibility or the number of actual arguments to a routine) are expressed as necessary and **sufficient** conditions. As a user of programming languages manuals, I have long been puzzled by their focus on **necessary** conditions: the source of the assignment *must* be of the same type as the target, the number of actual arguments *must* be the same as the number of formal arguments, and so on. All this is fine, but how do I know that I have considered all the relevant “musts” and that now I *may* submit my result as a valid piece of software?

To address this question in a systematic way, the description of almost every construct (such as Class, Instruction or Expression) includes a validity constraint of the form “this will be valid *if and only if* the following conditions are satisfied...”. The first *if* indicates that the conditions given are sufficient.

Others, it is hoped, will capitalize on this book’s attempt at precision and rigor to produce formal specifications of Eiffel. This style may contribute to making the presentation appear somewhat pedantic at times. It also carries some dangers: if you forget a condition, you give users a misplaced sense of security, whereas with the “must” style, since you never claim exhaustivity, they learn to be wary. But I felt that the sometimes painstaking task of producing the complete list of validity requirements should be the responsibility of the language specifier, not a job left for each language user and compiler implementor to handle individually.

ORDER OF PRESENTATION

As if all this was not already making the task impossible, two of the major goals for this book were to make it, against all odds, not *too* boring, and to encourage readers to study it in the way one should approach any decent book save for dictionaries and railway timetables: sequentially, from cover to cover.

This has meant another change from the common practice in language books, affecting the order of introduction of the various concepts.

The traditional order is bottom-up: first the lexical constituents (character set, constants, identifiers), then on to expressions, instructions and higher-level structuring mechanisms.

Here, in contrast, the presentation is top-down; once you have read the initial overview chapter and a short chapter introducing basic conventions, you will learn about the overall architecture of software systems written in Eiffel, then about the structure of their individual modules (classes), then about the routines, the attributes, the run-time model, and the lower-level constituents of Eiffel software.

“Construct” is defined more precisely below. The reason for this departure from the conventional order is to make sure that the key concepts and constructs are introduced first, enabling you to keep in mind the “big picture” throughout the presentation. After all, the details of bit constants and the representation of special characters in strings, although useful in some cases, are not what makes Eiffel exciting; it is more rewarding to understand right away how to build and read software systems at the highest level.

The presentation is divided into five parts:

Part A, Introduction, includes an overview of the language and a presentation of the conventions used for syntactic and semantic construct descriptions.

Part B, Structure, describes how Eiffel software is organized, explaining the architectural notions of system, cluster, class and feature, the inheritance and client relations between classes, routines, assertions, the feature adaptation mechanism based on inheritance, and the type system with the associated notion of conformance.

Part C, Contents, covers the inner parts of classes and features and their effect on software execution: control structures, instructions, exceptions, attributes, objects, values, expressions, entities, calls, interface with other languages, and the lexical structure of software texts.

Part D, Kernel Library elements, introduces some basic library facilities, covering universal features (available to all classes), persistence, arrays, strings, arithmetic classes, input and output.

Part E, Appendices, includes a presentation of recommended style standards, a discussion of the issues of language design and evolution, a bibliography on Eiffel, a description of Lace (the Language for Assembling Classes in Eiffel, used to combine classes into executable systems), two summaries of the differences between Eiffel 3 and previous versions (in both the old-to-new and new-to-old directions), and reference information: reserved words, precedence, syntax reference in three different forms (textual order, alphabetical order, diagrams), index.

PARAGRAPH TYPES AND ROAD SIGNS

For the reader of this book, the language description strategy described above has two significant consequences.

First, the top-down order of presentation implies that once in a while, as you read the presentation, you will be requested to perform an “act of faith” when seeing a reference to a concept that will only be defined precisely in a later part of the discussion. Such *forward references* are inevitable regardless of the style of presentation, since any useful programming language, even one such as Eiffel for which simplicity was a constant design obsession, includes concepts so intricately related as to make a linear presentation next to impossible; but the top-down style causes even more of them.

Next, the other key decisions – merging the reference manual and user manual into a single book, attempting to satisfy at once the needs of diverse audiences – mean that several types of discourse are interwoven in the discussion, belonging to various levels of formality and not all enjoying the same official status.

This book uses a number of devices to guide you through these different components of the presentation. In particular, it relies on a system of *road signs*, printed in the left margin, and notes, printed in the right margin, to help you grasp right away the type and level of individual components.

Here is the set of road signs:

A road sign indicates the nature of some part of the text: comment, preview, reminder, shortcut, syntax, validity, semantics, examples, caveat. Usually the sign applies to the marked paragraph, although it sometimes covers the following few paragraphs as well; if it is attached to the first paragraph of a section, it applies to the entire section.

*Signposts on
your way to
Eiffel wisdom*

Let us take a look at the various road signs; this will also serve to explain the presentation style.

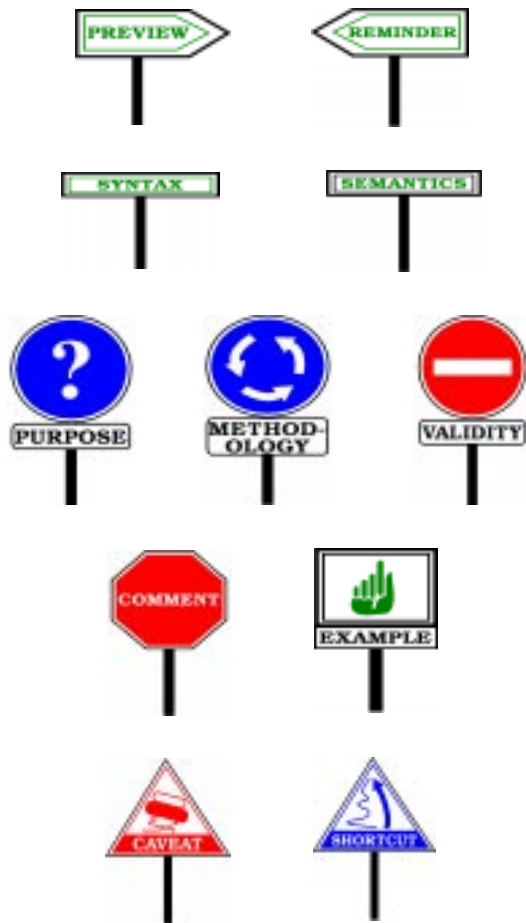
CROSS-REFERENCES AND SHORTCUTS

Forward references are much less confusing if they are explicitly signaled. All forward references are marked by a note in the right margin, preceded by the symbol → and indicating where to look for the final word on a concept being used ahead of its full definition.

The above use of “construct” was a forward reference.

In many cases, it will be necessary to include a partial explanation of the concept at the point of forward reference. The PREVIEW road sign indicates that this explanation is only temporary, although it suffices for the needs of the discussion at this point.

Because this book is meant to be read by human beings with less than unbounded memory, it also includes *backward references*: pointers to the place where a certain concept was originally introduced. Backward references are given as right-margin notes preceded by



Sometimes, a summary review of the concept is given at the point of backward reference; the corresponding paragraph is then labeled by the REMINDER road sign, indicating that the new explanation is only there to refresh the reader's memory, and that the official definition is the earlier one.

The use of cross-references, backward and forward, is particularly generous. To facilitate quick retrieval, most of these references include page numbers, often in addition to section numbers. To avoid bothering readers who follow the presentation sequentially, they are almost entirely confined to the right margin; but they should prove invaluable to anyone seeking to gain a thorough understanding of the concepts involved, with all their intertwining.

The desire to make fast back and forth searches easier also explains why the page number appears (in small print) even on the first page of a chapter – favoring reader's convenience over typographical tradition.

Besides cross references, other notes will also appear occasionally in the right margin; they are not part of the language definition, but give auxiliary information, such as bibliographic references, usually pointing to the detailed Eiffel bibliography of appendix =====

Another “directional” road sign is the `SHORTCUT` warning, which accompanies a note indicating that some part of the discussion (usually an entire section, or the remainder of a chapter) may be skipped at first reading. This applies to finer aspects of the presentation, which will be of interest to readers wishing to become acquainted with the details of Eiffel, but are not essential to an overall understanding. Typically, such aspects would have appeared in a reference manual but not in a user’s manual.

DESCRIBING A CONSTRUCT

The most important parts of the language description are of course the specifications of individual language structures, or **constructs**. Examples of constructs are `Class`, `Instruction`, `Expression`, `Identifier`. For most constructs, the presentation consists of the following sequence of elements, each labeled by the corresponding road sign from the preceding figure.

PURPOSE: a brief explanation of the construct’s role.

EXAMPLES: one or more typical uses. Some of the examples were designed specifically for this presentation; many others come, sometimes in simplified form, from the text of the Basic Eiffel Libraries, which form an important body of carefully written and heavily exercised Eiffel software.

The **EXAMPLES** sign is the “interesting detail” pictogram of the Swedish Recreation Standard.

SYNTAX: Specification of the textual form of software components, or **specimens**, corresponding to each construct. Syntax is described through a simple formalism.

===== starting on page ===== explains the syntax notation.

VALIDITY: Rules such as typing constraints, stating restrictions on permissible specimens of the construct, not captured by the syntactical specification alone.

The notion of validity is discussed in ===== and ===== starting on page ===== the role of validity codes, such as `VIEX` next to the road sign on the figure, is explained on page =====

SEMANTICS: Description of the meaning (that is to say, the run-time effect) of the construct.

For users as well as for implementors, distinguishing clearly between syntax, constraints and semantics is essential to a good understanding of the language. Every construct has a certain structure, is subject to some limitations, and has a certain effect.

The SYNTAX, VALIDITY and SEMANTICS paragraphs constitute the core of the official language definition. They are complemented in this role by paragraphs labeled DEFINITION, which introduce terms to be used thereafter with a precise meaning.

COMMENTS, WARNINGS AND ADVICE

Additional paragraphs, liberally interspersed in the rest of the discussion, serve explanatory purposes and are outside of the language definition proper:

- COMMENT: Time to stop for some explanations, discussions and informal addenda.
- METHODOLOGY: Advice on how to use individual constructs in the Eiffel software development method. The road sign suggests the proper way to go around an intersection — with apologies to readers from the British Isles, Australia, Japan, Singapore etc.
- CAVEAT: Remarks alerting you to slippery road segments – possible misunderstandings which could lead to mistakes or improper use of the language.
- PREVIEW, REMINDER: As discussed above.

Of course, not every paragraph is labeled. Unlabeled paragraphs generally fall under the “comment” category.

GRAPHICAL REPRESENTATIONS

For analysis and design discussions, for explaining software structures to others, for exploring classes and systems with “browsers” and other interactive tools, it is often useful to display information graphically.

The presentation of the most important language constructs includes a description of associated graphical conventions, based on some results of an effort directed by Jean-Marc Nerson to develop standardized pictorial representations of Eiffel software components – classes with their client and inheritance relations, features, assertions, notes etc. – in a simple and immediately understandable visual form.

The notation is known as BON – Business Object Notation – and is described in “Extending Eiffel Toward O-O Analysis and Design”. See the reference in appendix =====

These conventions are obviously not part of the language; they merely assist software construction and understanding. The form presented here is neither final nor complete; other publications will describe the notation in detail.

THE CAKE AND ITS ICING

A more general question arises from the last observation: what then, among the elements explained in this book, *is* officially part of the Eiffel language?

From the preceding discussion, it is clear that some parts, such as the paragraphs labeled SYNTAX, VALIDITY and SEMANTICS, definitely belong to the cake – the actual definition of Eiffel as a language. In contrast, the introductory overview of chapter ===== is only part of the icing, and so is anything labeled COMMENT, CAVEAT OR METHODOLOGY in the subsequent chapters.

This leaves some areas of uncertainty, especially around the Eiffel libraries, which are looted regularly in parts B and C for illustrations of language constructs, and provide most of the facilities described in part D: universal features, arrays, strings, persistence, basic arithmetic, input and output. (The classes of part D belong to the Kernel Library, containing fundamental facilities; the classes providing construct examples, sometimes in adapted or simplified form, are mostly from the Data Structure, Graphics and Parsing libraries.)

Chapter ===== discusses the status of libraries. When reading about the Kernel Library classes of part D, you may consider their interfaces – what in Eiffel is called their **short form**, excluding implementation aspects – to be part of the official description of Eiffel, although separate from the language proper. This does not apply to the other libraries, whose class extracts are used merely as examples.

See ===== starting on page ===== about the short form.

The decision to give an official status to the specification of some Kernel Library classes is subject to discussion. An argument against it is that these classes are not the only possible ones, and that their inclusion here might be unfairly preventing others from coming up with a better notion of (say) arrays. On closer look, however, several reasons suggest that one should not be too shy about enforcing a standard here:

One of the major attractions of Eiffel is precisely the presence of a standardized set of carefully designed libraries. It would be regrettable to forsake such a benefit simply out of a concern for fairness to other potential but as yet unproven solutions.

Most applications will need some of the Kernel Library classes selected for this book.

Only the interface is specified. This leaves room for multiple, competing implementations.

Encouraging the use of a standard set of library classes is not the same as claiming they are the last word. Improvements may be expected in the future. Eiffel's design makes it possible to cushion the effect of such changes on systems using the classes affected, thanks to mechanisms such as "obsolete" features and classes and, more generally, to the client-supplier independence enforced by the language and the method.

See ===== beginning on page ===== on obsolete features.

As with the rest of the language, future decisions on these issues will rest with others. As with the rest of the language, I have tried to lay the ground for the work and to proceed as far as a single person could go.

Contents

Preface: Meeting the challenge of software quality (<i>in progress</i>)	vii
THOUGHT AND EXPRESSION	viii
OLD, NEW AND OUT	viii
THE SIGNAL AND THE NOISE	ix
LANGUAGE LEVEL	x
THE FUTURE OF EIFFEL	x
MISSING ELEMENTS	xi
ACKNOWLEDGMENTS	xi
Preface to the third edition	xii
NON-CLASSICAL NUMBER THEORY	xii
MIGRATION AIDS	xii
About the status of Eiffel	xiii
About the language description	xv
TYPES OF DESCRIPTION, LEVELS OF DISCOURSE	xv
FORMALITY	xvi
ORDER OF PRESENTATION	xvii
PARAGRAPH TYPES AND ROAD SIGNS	xviii
CROSS-REFERENCES AND SHORTCUTS	xix
DESCRIBING A CONSTRUCT	xxi
COMMENTS, WARNINGS AND ADVICE	xxii
GRAPHICAL REPRESENTATIONS	xxii
THE CAKE AND ITS ICING	xxiii
Contents	xxv
PART I: INVITATION TO EIFFEL	1
1 An Eiffel tutorial	3
1.1 OVERVIEW	3
1.2 GENERAL PROPERTIES	4
1.3 THE SOFTWARE PROCESS IN EIFFEL	6
Clusters and the cluster model	7
Seamlessness and reversibility	8
Generalization and reuse	8
Constant availability	9
Compilation technology	9
Quality and functionality	9
1.4 HELLO WORLD	10
1.5 THE STATIC PICTURE: SYSTEM ORGANIZATION	11

	Systems	12
	Classes	12
	Class relations	13
	The global inheritance structure	14
	Clusters	14
	External software	15
1.6	THE DYNAMIC STRUCTURE: EXECUTION MODEL	15
	Objects, fields, values and references	16
	Features	16
	A simple class	18
	Creating and initializing objects	19
	Entities	21
	Calls	22
	Infix and prefix notation	23
	Type declaration	24
	Type categories	24
	Basic operations	26
	Deep operations and persistence	27
	Memory management	28
	Information hiding and the call rule	29
	Execution scenario	30
	Abstraction	31
1.7	GENERICITY	32
1.8	DESIGN BY CONTRACT, ASSERTIONS, EXCEPTIONS	34
	Design by Contract basics	35
	Expressing assertions	35
	Using assertions for built-in reliability	38
	Run-time assertion monitoring	38
	The short form of a class	40
	Exception handling	42
	Other applications of Design by Contract	45
1.9	THE INHERITANCE MECHANISM	45
	Basic inheritance structure	45
	Redefinition	46
	Polymorphism	47
	Dynamic binding	49
	Deferred features and classes	51
	Applications of deferred classes	54
	Structural property classes	56
	Multiple inheritance and feature renaming	57
	Inheritance and contracts	59
	Join and uneffecting	62
	Changing the export status	63
	Flat and flat-short forms	64
	Repeated inheritance and selection	65
	Constrained genericity	68
	Assignment attempt	69
	Covariance and anchored declarations	70
1.10	OTHER IMPORTANT MECHANISMS	73
	Once routines, shared objects, smart initialization and on-demand execution	73
	Constant attributes	75
	Instructions	75

	Lexical conventions	79
1.11	CONCURRENCY AND FURTHER DEVELOPMENTS	79
	SCOOP	79
	Other developments	81
PART II: LANGUAGE DESCRIPTION		83
2	Syntax, validity and semantics	85
2.1	OVERVIEW	85
2.2	SYNTAX: COMPONENTS, SPECIMENS, CONSTRUCTS	85
2.3	TERMINALS, NON-TERMINALS AND TOKENS	87
2.4	THE LEXICAL LEVEL	87
2.5	PRODUCTIONS	88
	Aggregate productions	89
	Choice productions	90
	Repetition productions	90
	Using recursive productions	92
	One production per non-terminal	93
	Non-production syntax rules	93
2.6	REPRESENTING TERMINALS	94
2.7	VALIDITY	96
2.8	INTERPRETING THE CONSTRAINTS	98
2.9	SEMANTICS	99
2.10	CORRECTNESS	99
2.11	TWO-TIER DEFINITION AND UNFOLDED FORMS	100
2.12	THE CONTEXT OF EXECUTING SYSTEMS	101
2.13	TEXTUAL CONVENTIONS	102
3	The architecture of Eiffel software	105
3.1	OVERVIEW	105
3.2	CLASSES	106
3.3	CLASS TEXTS AND CLASS NAMES	107
3.4	CLUSTERS	107
3.5	SYSTEMS	110
4	Classes	115
4.1	OVERVIEW	115
4.2	OBJECTS	115
4.3	FEATURES	116
4.4	USE OF CLASSES	116
4.5	THE CURRENT CLASS	117
4.6	CLASS TEXT STRUCTURE	117
4.7	PARTS OF A CLASS TEXT	119
4.8	ANNOTATING A CLASS	122
4.9	CLASS HEADER	124
	Deferred classes	125
	Expanded classes	126
	Validity of a class header	126
4.10	FORMAL GENERIC PARAMETERS	127
4.11	OBSOLETE MARK	128
5	Features	131
5.1	OVERVIEW	131

5.2	THE ROLE OF FEATURES	131
5.3	FEATURE CATEGORIES	132
5.4	IMMEDIATE AND INHERITED FEATURES	133
5.5	FEATURES PART: EXAMPLE	134
5.6	GRAPHICAL REPRESENTATION	136
5.7	FEATURES PART: SYNTAX	137
5.8	FORMS OF FEATURE	138
5.9	FEATURE DECLARATIONS: EXAMPLES	139
5.10	FEATURE DECLARATIONS: SYNTAX	140
5.11	FEATURE BODIES	143
5.12	HOW TO RECOGNIZE FEATURES	145
5.13	THE SIGNATURE OF A FEATURE	148
5.14	FEATURE NAME	150
5.15	OPERATOR FEATURES	154
5.16	ASSIGNER PROCEDURES	155
5.17	BRACKET FEATURE	158
5.18	SYNONYMS AND MULTIPLE DECLARATION	159
5.19	VALIDITY OF FEATURE DECLARATIONS	162
5.20	SCOPE OF NAMES	164
5.21	OBSOLETE FEATURES	165
5.22	NO IN-CLASS OVERLOADING	167
6	The inheritance relation	169
6.1	OVERVIEW	169
6.2	AN INHERITANCE PART	169
6.3	FORM OF THE INHERITANCE PART	170
6.4	GRAPHICAL CONVENTION	172
6.5	<i>ANY</i>	172
6.6	<i>NONE</i>	175
6.7	RELATIONS INDUCED BY INHERITANCE	175
6.6	PROHIBITING CYCLES	178
6.7	ADAPTING INHERITED FEATURES	180
6.8	NON-CONFORMING INHERITANCE	180
6.9	RENAMING	183
6.10	FEATURES AND THEIR NAMES	185
6.11	INDEPENDENCE OF INHERITANCE AND EXPANSION	189
7	Clients and exports	191
7.1	OVERVIEW	191
7.2	ENTITIES	191
7.3	CONVENTIONS	192
7.4	SIMPLE CLIENTS	193
7.5	EXPANDED CLIENTS	196
7.6	GENERIC CLIENTS	198
7.7	INDIRECT CLIENTS	200
7.8	EXPORT CONTROLS AND INFORMATION HIDING	200
	Restricting exports	201
	Exporting to oneself	202
	Exporting to descendants	203
	Making a feature secret	203
	Adapting the export status of inherited features	204

	Expanding or restricting the export status	206
	The export status of features	207
	Rules on setting the export status	208
7.9	-DOCUMENTING THE CLIENT INTERFACE OF A CLASS	212
	Selecting features	212
	Contract views	213
8	Routines	217
8.1	OVERVIEW	217
8.2	ROUTINE DECLARATION	217
8.3	FORMAL ARGUMENTS	219
8.4	USING A VARIABLE NUMBER OF ARGUMENTS	221
8.5	ROUTINE BODY	222
8.6	LOCAL VARIABLES AND <i>RESULT</i>	225
8.7	EXTERNALS	227
8.8	TYPES OF INSTRUCTIONS	228
9	Correctness and contracts	229
9.1	OVERVIEW	229
9.2	WHY ASSERTIONS?	229
9.3	GRAPHICAL CONVENTION	231
9.4	USES OF ASSERTIONS	232
9.5	FORM OF ASSERTIONS	232
9.6	UNFOLDING ASSERTIONS UNDER INHERITANCE	235
9.7	ASSERTIONS ON INDIVIDUAL FEATURES	235
	Preconditions and postconditions	235
	The contract of a routine	236
	Constraints on routine assertions	237
	“Old” expression	239
	“Only” clause	241
9.8	CLASS INVARIANTS	245
9.9	THE CONSISTENCY OF A CLASS	246
9.10	CHECK INSTRUCTIONS	248
9.11	LOOP INVARIANTS AND VARIANTS	250
9.12	THE CORRECTNESS OF A CLASS	252
9.13	RULES OF RUN-TIME ASSERTION MONITORING	253
	Associated boolean expression	254
	Assertion monitoring	255
	Levels of assertion monitoring	257
	Invariant and qualified calls	258
10	Feature adaptation	261
10.1	OVERVIEW	261
10.2	TERMINOLOGY: REDECLARATION, REDEFINITION, EFFECTING	261
10.3	REDECLARING INHERITED FEATURES: WHY AND HOW	262
10.4	FEATURE ADAPTATION CLAUSES	263
10.5	WHY REDEFINE?	265
10.6	REDEFINITION EXAMPLES	266
10.7	THE REDEFINITION CLAUSE	267
10.8	REDEFINITION IN THE SOFTWARE PROCESS	268
10.9	CHANGING THE SIGNATURE	269
10.10	THE NEED FOR ANCHORED DECLARATIONS	271

10.11	DEFERRED FEATURES	272
10.12	DEFERRED CLASSES FOR DESCRIBING ABSTRACTIONS	273
10.13	DEFERRED CLASSES FOR SYSTEM DESIGN AND ANALYSIS	274
10.14	EFFECTING A DEFERRED FEATURE	276
10.15	PARTIALLY DEFERRED CLASSES AND PROGRAMMED ITERATION	277
10.16	REDECLARATION AND TYPING	280
10.17	REDECLARATION AND ASSERTIONS	283
10.18	RULES ON INHERITED ASSERTIONS	287
10.19	UNDEFINING A FEATURE	290
10.20	REDEFINITION AND EFFECTING	291
10.21	THE JOIN MECHANISM	292
10.22	MERGING EFFECTIVE FEATURES	294
10.23	NAME CLASHES	297
10.24	ADDING TO INHERITED BEHAVIOR: PRECURSOR	299
	The need for a precursor mechanism	299
	Precursor basics and examples	300
	Choosing between multiple precursors	302
	Precursor specification	303
10.25	REDEFINITION AND UNDEFINITION RULES	306
10.26	DEFERRED AND EFFECTIVE FEATURES AND CLASSES	309
10.27	ORIGIN AND SEED	311
10.28	REDECLARATION RULES	312
10.29	RULES ON JOINING FEATURES	315
11	Types	323
11.1	OVERVIEW	323
11.2	THE ROLE OF TYPES	323
11.3	WHERE TO USE TYPES	325
11.4	HOW TO DECLARE A TYPE	327
11.5	INSTANCES AND VALUES	329
11.6	INSTANCES OF A CLASS	331
11.7	BASE CLASS, BASE TYPE AND TYPE SEMANTICS	332
11.8	CLASS TYPES WITHOUT GENERICITY	334
11.9	EXPANDED TYPES	335
	Role of expanded types	335
	Defining expanded types	337
	Basic types	338
11.10	ANCHORED TYPES	339
	Anchored examples	340
	Anchoring to <i>Current</i>	340
	Anchoring to an expanded or generic	341
	Avoiding anchor cycles	343
		344
	Validity and semantics of anchored types	344
11.11	GUARANTEEING ATTACHMENT	346
11.12	STAND-ALONE TYPES	347
12	Genericity	349
12.1	OVERVIEW	349
12.2	GENERIC CLASSES	349

12.3	GENERIC CLASSES AND GENERIC DERIVATIONS	351
12.4	SELF-INITIALIZING FORMAL PARAMETERS	352
12.5	CONSTRAINED AND UNCONSTRAINED GENERICITY	353
12.6	CONSTRAINED GENERICITY	354
12.7	RULES ON CONSTRAINED GENERICITY	356
12.8	CONSTRAINTS AND CREATION	360
12.9	RECURSIVE GENERIC CONSTRAINTS	362
12.10	SEMANTICS OF GENERIC TYPES	363
12.11	CURRENT TYPE, FEATURES OF A TYPE	365
12.12	APPLYING GENERICITY TO TYPES	366
12.13	THE CASE OF MULTIPLE CONSTRAINTS	367
13	Tuples	371
13.1	OVERVIEW	371
13.2	TUPLES IN A NUTSHELL	371
13.3	USING TUPLE TYPES AND TUPLES	372
13.4	ANONYMOUS CLASSES	375
13.5	CONFORMANCE ===== TO BE REWRITTEN	378
13.6	MULTIPLE RESULTS AND VARIABLE NUMBERS OF ARGUMENTS	378
	Emulating multiple results	379
	Emulating a variable number of arguments	380
13.7	TUPLES AS ARRAYS ===== TO BE REWRITTEN	380
14	Conformance	383
14.1	OVERVIEW	383
14.2	CONVERTIBILITY AND COMPATIBILITY	384
14.3	APPLICATIONS OF CONFORMANCE	385
14.4	EXPRESSION AND SIGNATURE CONFORMANCE	386
14.5	DIRECT AND INDIRECT CONFORMANCE	387
14.6	CONFORMANCE TO A NON-GENERIC REFERENCE TYPE	389
14.7	GENERICALLY DERIVED REFERENCE TYPES	390
14.8	FORMAL GENERIC PARAMETER CONFORMANCE	393
14.9	EXPANDED TYPE CONFORMANCE	394
14.10	TUPLE TYPE CONFORMANCE	396
14.11	ANCHORED TYPE CONFORMANCE	398
15	Convertibility	399
15.1	OVERVIEW	399
15.2	WHY IMPLICIT CONVERSION?	399
15.3	CONVERSION BASICS AND EXAMPLES	400
15.4	CONVERSION QUERIES	403
15.5	USING CONVERSIONS PROPERLY	406
15.6	CONVERSION PRINCIPLES	408
15.7	CONVERSION SYNTAX AND VALIDITY	410
15.8	SEMANTICS OF CONVERSION	415
15.9	CONVERTING AN EXPRESSION EXPLICITLY	416
15.10	EXPRESSION CONVERTIBILITY: THE ROLE OF PRECONDITIONS	420
15.11	MULTIPLE CONVERSION TYPES	426
15.12	MIXED-TYPE EXPRESSIONS: TARGET CONVERSION	428
	Using target conversion	429
	Validity of target conversion	430

Target conversion: a discussion	431
16 Repeated inheritance	433
16.1 OVERVIEW	433
16.2 CASES OF REPEATED INHERITANCE	434
16.3 THE TWO QUESTIONS OF REPEATED INHERITANCE	435
16.4 SHARING AND REPLICATION	436
16.5 THE CASE OF REDECLARED FEATURES	442
16.6 THE CASE OF ATTRIBUTES	448
16.7 THE CASE OF CONFLICTING GENERIC DERIVATIONS	450
16.8 KEEPING THE ORIGINAL VERSION OF A REDEFINED FEATURE	451
16.9 USING REPLICATION: COUNTERS AND ITERATION	453
16.10 THE SEMANTICS OF REPLICATION	457
16.11 RETAINING VICTORS FROM ALTERNATIVE BRANCHES	460
16.12 THE NEED FOR SELECT	463
16.13 THE REPEATED INHERITANCE CONSISTENCY CONSTRAINT	463
16.14 THE INHERITED FEATURES OF A CLASS	469
17 Control structures	477
17.1 OVERVIEW	477
17.2 COMPOUND	477
17.3 CONDITIONAL	480
17.4 MULTI-BRANCH CHOICE	482
17.5 OBJECT TEST	491
17.6 USING SELECTION INSTRUCTIONS PROPERLY	491
17.7 LOOP	494
Loop structure and properties	494
Loop semantics	496
Ensuring non-void references in a loop	497
17.8 THE DEBUG INSTRUCTION	497
18 Attributes	499
18.1 OVERVIEW	499
18.2 GRAPHICAL REPRESENTATION	499
18.3 VARIABLE ATTRIBUTES	500
18.4 ATTRIBUTES IN FULL FORM	500
18.5 CONSTANT ATTRIBUTES	501
18.6 CONSTANT ATTRIBUTES WITH MANIFEST VALUES	502
19 Objects, values and entities	505
19.1 OVERVIEW	505
19.2 OBJECTS AND THEIR TYPES	506
19.3 VALUES AND INSTANCES	506
19.4 BASIC TYPES	508
19.5 REFERENCE AND COPY SEMANTICS	508
19.6 COMPOSITE OBJECTS AND THEIR FIELDS	508
19.7 REFERENCE ATOMICITY	510
19.8 EXPRESSIONS AND ENTITIES	512
19.9 SEMANTICS: EVALUATING AND INITIALIZING ENTITIES	514
20 Creating objects	523
20.1 OVERVIEW	523

20.2	FORMS OF CREATION: AN OVERVIEW	524
20.3	BASIC FORM OF CREATION INSTRUCTIONS	525
20.4	OMITTING THE CREATION PROCEDURE	527
20.5	CREATORS AND INHERITANCE	534
20.6	USING AN EXPLICIT TYPE	535
	Specifying the creation type	535
	Choosing between types	536
	Creation and deferred classes	537
	Single choice and factory objects	537
20.7	RESTRICTING CREATION AVAILABILITY	539
20.8	THE CASE OF EXPANDED TYPES	542
20.9	CREATING INSTANCES OF FORMAL GENERICS	543
20.10	PRECONDITIONS OF CREATION PROCEDURES	546
20.11	CREATION SYNTAX AND VALIDITY	547
20.12	CREATION SEMANTICS	556
20.13	REMOTE CREATION	557
20.14	CREATION EXPRESSIONS AND ANONYMOUS OBJECTS	558
20.15	GARBAGE COLLECTION	564
21	Comparing and duplicating objects	565
21.1	OVERVIEW	565
21.2	COPYING AN OBJECT	565
21.3	EQUALITY EXPRESSIONS	565
	Effect of a copy operation	571
	Specification of default copy	572
	Tuning copy semantics	573
21.4	CLONING AN OBJECT	575
	Using cloning	575
	Twin	576
	Specification of default cloning	576
	Cloning, types and factories	578
21.5	DEEP COPYING AND CLONING	579
21.6	OBJECT EQUALITY	580
21.7	DEEP EQUALITY	582
22	Attaching values to entities	587
22.1	OVERVIEW	587
22.2	ROLE OF REATTACHMENT OPERATIONS	588
22.3	FORMS OF UNCONDITIONAL REATTACHMENT	588
22.4	SYNTAX AND VALIDITY OF ASSIGNMENT	589
22.5	THE STATUS OF FORMAL ROUTINE ARGUMENTS	590
22.6	CONVERSIONS	591
22.7	SEMANTICS OF REATTACHMENT	593
22.8	AN EXAMPLE	601
22.9	ABOUT REATTACHMENT	603
22.10	EFFECT ON GENERIC PROGRAMMING	604
22.11	POLYMORPHISM	606
22.12	ASSIGNER CALL	607
22.13	SEMI-STRICT OPERATORS	610
	The notion of strictness	611
	The need for semi-strict operators	611

More on strictness	614
22.14 CONDITIONAL REATTACHMENT	615
Limitations of unconditional reattachment	615
22.15 MEMORY MANAGEMENT	616
22.16 SEMANTICS OF EQUALITY	618
23 Feature call	621
23.1 OVERVIEW	621
23.2 PARTS OF A CALL	622
23.3 USES OF CALLS	623
23.4 UNIFORM ACCESS	624
23.5 OPERATOR AND BRACKET FORMS	624
23.6 COMPLEX TARGETS	625
23.7 CALL SYNTAX	626
23.8 COMPONENTS OF A CALL	628
23.9 NON-OBJECT CALLS	629
23.10 CLASS VALIDITY	631
Export validity	632
Argument validity	634
Target validity and Void-Safe Eiffel	635
Combining the rules	636
23.11 INTRODUCTION TO CALL SEMANTICS	636
23.12 DYNAMIC BINDING	638
23.13 THE IMPORTANCE OF BEING DYNAMIC	640
23.14 ONCE ROUTINES	641
Once basics	641
Once uses	642
Predefined once keys	642
Further once tuning	643
Once routine semantics	644
23.15 ATTRIBUTES AND EXTERNALS	647
23.16 THE MACHINERY OF EXECUTING CALLS	647
Scheme for a routine call	647
Current object and routine	648
Naming the current object	650
23.17 PRECISE CALL SEMANTICS	652
Rule for non-once routines	652
General call semantics	652
23.18 CALLS AS EXPRESSIONS	655
24 Eradicating void calls	657
24.1 OVERVIEW	657
24.2 OVERALL SCHEME	658
24.3 THE OBJECT TEST	658
24.4 VOID TESTS	662
24.5 CERTIFIED ATTACHMENT PATTERNS	663
24.6 ATTACHED EXPRESSIONS	664
25 Typing-related properties	665
25.1 OVERVIEW	665
25.2 SYNTAX VARIANTS	666
25.3 BASIC CONCEPTS	667

25.4		668
25.5	SYSTEM-LEVEL VALIDITY	669
25.6	VIOLATING SYSTEM VALIDITY	670
25.7	NOTES ON THE TYPE POLICY	672
25.8	WHY DISTINGUISH?	677
25.9	A LOOK AT THE DYNAMIC CLASS SET	677
25.10	THE CALL VALIDITY RULE	681
25.11	CREATION VALIDITY (SYSTEM-LEVEL)	686
26	Exception handling	689
26.1	OVERVIEW	689
26.2	WHAT IS AN EXCEPTION?	690
26.3	EXCEPTION HANDLING POLICY	691
26.4	RESCUE CLAUSES AND ORGANIZED PANIC	692
26.5	THE DEFAULT RESCUE	694
26.6	RETRY INSTRUCTIONS AND RESUMPTION	695
26.7	SYSTEM FAILURE AND THE EXCEPTION HISTORY TABLE	698
26.8	SYNTAX AND VALIDITY OF THE EXCEPTION CONSTRUCTS	701
26.9	EXCEPTION CORRECTNESS	702
26.10	SEMANTICS OF EXCEPTION HANDLING	702
26.11	EXCEPTION CORRECTNESS	707
26.12	FINE-TUNING THE MECHANISM	709
26.13	OVERVIEW	712
26.14	PLATFORM-DEPENDENT SIGNAL CODES	712
26.15	CLASS EXCEPTIONS	714
27	Agents, iteration and introspection	719
27.1	OVERVIEW	719
27.2	A QUICK PREVIEW	719
27.3	FROM CALLS TO AGENTS	723
	Feature calls and their operands	723
	Delaying calls	724
	Agents and their operands	725
27.4	AGENT TYPES	726
27.5	CALL AGENTS	728
	All-closed agents	729
	Keeping operands open	730
	The brace convention	732
	Omitting the argument list	732
	A summary of the possibilities	733
27.6	USING AGENTS	733
	GUI programming: establishing a direct connection to the Business Model	734
	Integrating a function	736
	Iteration examples	738
27.7	TWO ADVANCED EXAMPLES	743
	Error processing without the mess	743
	Once per object	744
27.8	USING INLINE AGENTS	746
27.9	ACCESSING FEATURE PROPERTIES	749
27.10	THE BASE CLASS AND TYPE	750

27.11	AGENT SYNTAX	751
	Syntax of call agents	752
	Syntax of inline agents	753
27.12	AGENT VALIDITY	754
	Validity of call agents	754
	Validity of inline agents	755
27.13	AGENT SEMANTICS	756
	Call-agent equivalent of an inline agent	756
	Open and closed operands	758
	Type and value of an agent expression	759
28	Expressions	761
28.1	OVERVIEW	761
28.2	GENERAL FORM OF EXPRESSIONS	761
28.3	SUBEXPRESSIONS	764
28.4	PARENTHEZIZED EXPRESSIONS	765
28.5	OPERATOR EXPRESSIONS	766
	Operator expression basics	766
	Operator expression syntax	766
	Precedence and Parenthesized Form	767
	Accounting for target conversion	770
	Operator expression validity and semantics	771
28.6	SEMISTRICT BOOLEAN OPERATORS	774
28.7	BRACKET EXPRESSIONS	778
28.8	THE EQUIVALENT DOT FORM	780
28.9	BOOLEAN EXPRESSIONS	781
28.10	ENTITIES	781
28.11	THE TYPE OF AN EXPRESSION	782
28.12	EXPRESSIONS AND THE SEMICOLON	784
29	Constants	787
29.1	OVERVIEW	787
29.2	GENERAL FORM OF CONSTANTS	787
29.3	FORCING A TYPE ON A CONSTANT	789
29.4	THE TYPE OF A CONSTANT	790
29.5	INTEGER CONSTANTS	792
29.6	REAL CONSTANTS	792
29.7	CHARACTER CONSTANTS	793
29.8	MANIFEST STRINGS	794
	Basic manifest strings	796
	Verbatim strings	798
	Choosing between basic and verbatim manifest strings	804
	“Once” string expressions	805
	Run-time model for manifest strings	807
29.9	MANIFEST TUPLES	809
29.10	SEMANTICS OF CONSTANT ATTRIBUTES	813
30	Basic types	815
30.1	OVERVIEW	815
30.2	EXPANSION STATUS	815
30.3	BASIC CLASSES AND THEIR INHERITANCE STRUCTURE	817
30.4	BOOLEANS	819

30.5	CHARACTERS	819
30.6	INTEGERS	820
30.7	REALS	820
30.8	ADDRESSES	821
31	Interfacing with C, C++ and other environments	823
31.1	OVERVIEW: THE COMPONENT COMBINATOR	823
31.2	WHAT EIFFEL CAN DO WITH THE REST OF THE WORLD	824
31.3	WHEN TO USE EXTERNAL SOFTWARE	825
31.4	REGISTERED LANGUAGES AND THE ROLE OF C	827
31.5	BASICS OF EXTERNAL ROUTINES	828
31.6	EXECUTING AN EXTERNAL CALL	831
31.7	ARGUMENT AND RESULT TRANSMISSION	831
31.8	PASSING THE ADDRESS OF AN EIFFEL FEATURE	833
31.9	SPECIAL INTERFACE SUBLANGUAGES	837
31.10	GENERAL SUBLANGUAGE MECHANISMS	837
	Specifying an external routine signature	838
	Specifying external files	840
31.11	THE C INTERFACE SUBLANGUAGE	842
	Syntax specification	843
	Specifying C code inline	844
	Controlling the Eiffel-C type correspondence	846
31.12	THE C++ INTERFACE SUBLANGUAGE	847
	The syntax specification	848
	Conditions on C++ features	848
	Processing C++ features	849
	Extra argument	850
31.13	WRAPPING C++ CLASSES: LEGACY++	851
	The role of Legacy++	851
	Calling Legacy++	851
	Result of applying Legacy++	851
	Legacy++ limitations	852
	Legacy++ example	852
31.14	USING DYNAMIC LINKED LIBRARIES (DLLS)	855
	The static DLL sublanguage	856
31.15	DESC: CALLING A DLL ROUTINE DETERMINED AT RUN TIME	858
	DESC overview	859
	Creating a library object	859
	Creating a routine object	860
	Type codes	862
	Calling a routine	863
	Accessing the result of a function	863
	Consistency requirements and protection against errors	863
	Sharing and freeing	864
31.16	THE CECIL LIBRARY	865
	Cecil overview	865
	Cecil role and status	865
	Compiling for Cecil	866
	Avoiding abusive optimization	866
	Basic Cecil conventions	868
	Initializing the Eiffel 4 run-time	869
	Manipulating values of basic Eiffel types	870

Manipulating Eiffel class types	871
Accessing an Eiffel object	871
Creating an Eiffel object	873
Calling routines	874
Requesting a non-existing routine	875
Accessing field objects	876
ISE Eiffel specifics	876
32 Lexical components	879
32.1 OVERVIEW	879
32.2 CHARACTER SETS	879
32.3 CHARACTER CATEGORIES	880
32.4 GENERAL FORMAT	881
32.5 BREAKS	881
32.6 COMMENTS	881
32.7 TEXT LAYOUT	885
32.8 LETTER CASE	886
32.9 TOKEN CATEGORIES	887
32.10 RESERVED WORDS	888
32.11 SPECIAL SYMBOLS	889
32.12 IDENTIFIERS	891
32.13 OPERATORS	892
32.14 CHARACTERS	894
32.15 STRINGS	898
32.16 INTEGERS	899
32.17 REAL NUMBERS	902
33 Concurrency (<i>not done</i>)	905
33.1 OVERVIEW	905
34 Style guidelines (<i>not done</i>)	907
34.1 OVERVIEW	907
34.2 LETTER CASE	907
34.3 CHOICE OF NAMES	908
34.4 GRAMMATICAL CATEGORIES FOR FEATURE NAMES	910
34.5 GROUPING FEATURES	911
34.6 HEADER COMMENTS	912
34.7 OTHER COMMENTS	914
34.8 EIFFEL NAMES IN COMMENTS	914
34.9 LAYOUT	915
34.10 OPTIONAL SEMICOLONS	919
34.11 LEXICAL CONVENTIONS	920
34.12 FONTS	920
34.13 GUIDELINES FOR ANNOTATING CLASSES	921
PART III: KERNEL LIBRARY CLASSES	925
35 Universal features and class ANY (<i>in progress</i>)	927
35.1 OVERVIEW	927
35.2 INPUT AND OUTPUT FEATURES	928
35.3 DUPLICATION AND COMPARISON ROUTINES	928
35.4 OBJECT PROPERTIES	929

35.5	PLATFORM-DEPENDENT FEATURES	929
35.6	OTHER UNIVERSAL FEATURES	930
36	Arrays and strings (<i>not done</i>)	931
36.1	OVERVIEW	931
36.2	REPRESENTATION	931
36.3	RESIZING	933
36.4	BASIC ARRAY HANDLING	934
36.5	COPYING AND COMPARING ARRAYS	935
36.6	MANIFEST ARRAYS	937
36.7	STRINGS	937
37	Persistence (<i>not done</i>)	941
37.1	OVERVIEW	941
37.2	CLASSES FOR PERSISTENCE	941
37.3	OBJECTS AND THEIR DEPENDENTS	942
37.4	RETRIEVAL, TYPING, AND THE ASSIGNMENT ATTEMPT	943
37.5	STORING AND RETRIEVING AN ENTIRE STRUCTURE	945
37.6	CLASS STORABLE	947
37.7	ENVIRONMENTS	950
37.8	OPENING AND CLOSING ENVIRONMENTS	951
37.9	RECORDING AND ACCESSING OBJECTS IN AN ENVIRONMENT	951
37.10	THE OBJECTS OF AN ENVIRONMENT	952
37.11	REQUESTING INFORMATION ABOUT ENVIRONMENTS	953
37.12	STORING ENVIRONMENTS	954
37.13	RETRIEVING AN ENVIRONMENT	955
37.14	AN ENVIRONMENT EXAMPLE	956
37.15	CLASS <i>ENVIRONMENT</i>	960
38	Input and output (<i>not done</i>)	965
38.1	OVERVIEW	965
38.2	PURPOSE OF THE CLASS	965
38.3	INPUT TECHNIQUES	966
38.4	CLASS STANDARD_FILES	968
A	ELKS: The Eiffel Library Kernel Standard	971
A.1	OVERVIEW	971
A.2	CONTENTS OF THIS STANDARD	971
A.3	COMPATIBILITY CONDITIONS	972
A.4	REQUIRED CLASSES	973
A.5	REQUIRED ANCESTRY LINKS	974
A.6	SHORT FORMS OF REQUIRED CLASSES	974
A.6.1	CLASS <i>ANY</i>	975
A.6.2	CLASS <i>TYPE</i>	976
A.6.3	CLASS <i>PART_COMPARABLE</i>	977
A.6.4	CLASS <i>COMPARABLE</i>	978
A.6.5	CLASS <i>HASHABLE</i>	979
A.6.6	CLASS <i>NUMERIC</i>	980
A.6.7	CLASS <i>INTERVAL</i>	981
A.6.8	CLASS <i>BOOLEAN</i>	982
A.6.9	CLASS <i>CHARACTER</i>	983
A.6.10	CLASS <i>INTEGER_GENERAL</i>	984

A.6.11 CLASS <i>INTEGER</i>	987
A.6.12 CLASS <i>INTEGER_8</i>	988
A.6.13 CLASS <i>INTEGER_16</i>	989
A.6.14 CLASS <i>INTEGER_64</i>	990
A.6.15 CLASS <i>REAL_GENERAL</i>	991
A.6.16 CLASS <i>REAL</i>	993
A.6.17 CLASS <i>TYPED_POINTER</i>	994
A.6.18 CLASS <i>POINTER</i>	995
A.6.19 CLASS <i>ARRAY</i>	996
A.6.20 CLASS <i>ANONYMOUS</i>	997
A.6.21 CLASS <i>STRING</i>	998
A.6.22 CLASS <i>STD_FILES</i>	1001
A.6.23 CLASS <i>FILE</i>	1002
A.6.24 CLASS <i>STORABLE</i>	1005
A.6.25 CLASS <i>MEMORY</i>	1006
A.6.26 CLASS <i>EXCEPTIONS</i>	1007
A.6.27 CLASS <i>ARGUMENTS</i>	1008
A.6.28 CLASS <i>PLATFORM</i>	1009
A.6.29 CLASS <i>ONCE_MANAGER</i>	1010
A.6.30 CLASS <i>ROUTINE</i>	1011
A.6.31 CLASS <i>PROCEDURE</i>	1012
A.6.32 CLASS <i>FUNCTION</i>	1013
A.6.33 CLASS <i>PREDICATE</i>	1014

PART IV: THE LACE CONTROL LANGUAGE _____ **1015**

B Specifying systems in Lace (*in progress*) **1017**

B.1 OVERVIEW	1017
B.2 A SIMPLE EXAMPLE	1018
B.3 ON THE ROLE OF LACE	1019
B.4 A COMPLETE EXAMPLE	1020
B.5 BASIC CONVENTIONS	1022
B.6 BASICS OF CLUSTER CLAUSES	1024
B.7 STORING PROPERTIES WITH A CLUSTER	1026
B.8 EXCLUDING AND INCLUDING SOURCE FILES	1026
B.9 SPECIFYING OPTIONS	1028
B.10 SPECIFYING EXTERNAL ELEMENTS	1032
B.11 ONCE CONTROL	1033
B.12 GENERATION	1033
B.13 VISIBLE FEATURES	1034
B.14 COMPLETE LACE GRAMMAR	1037
B.15 LACE VALIDITY RULES	1037

PART V: COMPLEMENTS _____ **1039**

C On language design and evolution **1041**

C.1 SIMPLICITY, COMPLEXITY	1041
C.2 CONSISTENCY	1042
C.3 UNIQUENESS	1044
C.4 TOLERANCE AND DISCIPLINE	1046
C.5 METHODOLOGY	1047
C.6 MEA CULPA, MEA MAXIMA CULPA	1047
C.7 THE LANGUAGE AND THE LIBRARIES	1047

C.8	ON SYNTAX	1048
C.9	THE INVENTOR AND THE ASSEMBLER	1050
C.10	FROM THE INITIAL DESIGN TO THE ASYMPTOTE	1050
C.11	EXTENSIONS	1051
C.12	CHANGES	1052
C.13	THE POLITICS OF LANGUAGE EVOLUTION	1053
D	Credits (<i>in progress</i>)	1055
D.1	OVERVIEW	1055
D.2	IEFFEL SOFTWARE	1055
D.3	ECMA TC39-TG4	1056
D.4	IEFFEL COMMUNITY	1056
D.5	CREDIT FOR SPECIFIC INVENTIONS	1058
E	A brief history of Eiffel	1061
F	Language changes from the previous edition	1063
F.1	OVERVIEW	1063
F.2	REMOVED MECHANISMS	1064
F.3	BACKWARD COMPATIBILITY	1065
F.4	NEW CONSTRUCTS	1066
38.5	SEMANTIC EXTENSIONS AND CHANGES	1067
F.5	KERNEL LIBRARY CHANGES	1068
F.6	LEXICAL AND SYNTACTIC CHANGES	1070
F.7	CHANGES IN VALIDITY CONSTRAINTS AND CONFORMANCE RULES	1071
G	Changes from early versions	1077
G.1	OVERVIEW	1077
G.2	SCOPE OF THE CHANGES	1077
G.3	OLDER POST-OOSC-1 EXTENSIONS	1078
G.4	SEMICOLONS	1079
G.5	FEATURE ADAPTATION	1079
G.6	SPECIFYING EXPORT STATUS	1079
G.7	ADAPTING PRECONDITIONS AND POSTCONDITIONS	1080
G.8	REMOVING AMBIGUITIES IN REPEATED INHERITANCE	1081
G.9	RENAMING, REDEFINING, UNDEFINING AND JOINING	1081
G.10	SYNONYMS	1082
G.11	FROZEN FEATURES	1082
G.12	ANCHORING TO A FORMAL ARGUMENT	1082
G.13	CREATION SYNTAX	1083
G.14	UNIFORM SEMANTICS FOR DOT NOTATION	1083
G.15	MANIFEST ARRAYS	1084
G.16	DEFAULT RESCUE	1084
G.17	EXPANDED CLASSES	1084
G.18	SEMANTICS OF EXPANDED TYPES	1084
G.19	FREE INFIX AND PREFIX OPERATORS	1085
G.20	OBSOLETE CLAUSE	1085
G.21	RESERVED WORDS	1085
G.22	OTHER LEXICAL CHANGES	1086

H An Eiffel tutorial	1087
I Eiffel bibliography (<i>not done</i>)	1089
I.1 OVERVIEW	1089
I.2 BOOKS	1089
I.3 ISE MANUALS	1089
I.4 INFORMATION SOURCES	1094
PART VI: REFERENCE	1095
J.1 INTRODUCTION	1097
J.2 LANGUAGE SPECIFICATION	1097
K Syntax in alphabetical order	1143
K.1 OVERVIEW	1143
K.2 SYNTAX	1143
L Reserved words, special symbols, operator precedence	1153
L.1 OVERVIEW	1153
L.2 RESERVED WORDS	1153
L.3 SPECIAL SYMBOLS	1154
L.4 OPERATORS AND THEIR PRECEDENCE	1155
L.5 KEYWORDS AND SYMBOLS OF SPECIAL INTERFACE SUBLANGUAGES	1155
M Syntax diagrams (<i>not done</i>)	1157
PART VII: THE LANGUAGE STANDARD	1161
1.1 Overview	1
1.2 “The Standard”	1
1.3 Aspects covered	1
1.4 Aspects not covered	1
2.1 Definition	1
2.2 Compatibility and non-default options	2
2.3 Departure from the Standard	2
3.1 Earlier Eiffel language specifications	2
3.2 Eiffel Kernel Library	2
3.3 Floating point number representation	3
3.4 Character set: Unicode	3
3.5 Character set: ASCII	3
3.6 Phonetic alphabet	3
5.1 Standard elements	3
5.2 Normative elements	3
5.3 Rules on definitions	4
5.4 Use of defined terms	4
5.5 Unfolded forms	4
5.6 Language description	5
5.7 Validity: “if and only if” rules	5
6.1 Name of the language	5
6.2 Pronunciation	5
7.1 Design principles	5
7.2 Object-oriented design	6
7.3 Classes	7

7.4	Types	10
7.5	Assertions	11
7.6	Exceptions	14
7.7	Genericity	15
7.8	Inheritance	16
7.9	Polymorphism and dynamic binding	17
7.10	Combining genericity and inheritance	19
7.11	Deferred classes	20
7.12	Tuples and agents	22
7.13	Type- and void-safety	22
7.14	Putting a system together	23
PART VIII: BACK MATTER		163
Index		165

PART I: INVITATION TO EIFFEL

This first part of the book presents Eiffel informally in short introductory chapter, [An Eiffel tutorial](#), *Eiffel in a Nutshell*, useful in particular if you are new to Eiffel and want to get a quick appreciation of what it is all about.

Further introductory material appears in the appendices:

- Instead of [An Eiffel tutorial](#) you may prefer a longer presentation, which essentially covers the entire language: [An Eiffel tutorial](#), in appendix [H](#), sufficient for advanced uses of Eiffel.
- You may also be interested in appendix [E](#), which provides [A brief history of Eiffel](#).

None of this material has any official reference status. Although you shouldn't find any discrepancy with the reference material of part [II](#), it's of course the latter which you should follow if you have the impression of a contradiction.

An Eiffel tutorial

This chapter presents a general description of Eiffel, assuming no prior knowledge. Although all its elements are described in more detail in the subsequent chapters, some readers may prefer to skip it; others, however, may use it as a preparation for the rest of the material.

The reader in search of a quick overview (“Eiffel in a nutshell”) may turn instead to section 7 of the ECMA Eiffel standard (pages [5](#) to [24](#)) which provides a concise presentation of the language essentials.

1.1 OVERVIEW

Eiffel is a method and language for the efficient description and development of quality systems.

As a language, Eiffel is more than a programming language. It covers not just programming in the restricted sense of implementation but the whole spectrum of software development:

- Analysis, modeling and specification, where Eiffel can be used as a purely descriptive tool to analyze and document the structure and properties of complex systems (even non-software systems).
- Design and architecture, where Eiffel can be used to build solid, flexible system structures.
- Implementation, where Eiffel provides practical software solutions with an efficiency comparable to solutions based on such traditional approaches as C and Fortran.
- Maintenance, where Eiffel helps thanks to the architectural flexibility of the resulting systems.
- Documentation, where Eiffel permits automatic generation of documentation, textual and graphical, from the software itself, as a partial substitute for separately developed and maintained software documentation.

Although the language is the most visible part, Eiffel is best viewed as a *method*, which guides system analysts and developers through the process of software construction. The Eiffel method is focused on both productivity (the ability to produce systems on time and within budget) and quality, with particular emphasis on the following quality factors:

- *Reliability*: producing bug-free systems, which perform as expected.
- *Reusability*: making it possible to develop systems from prepackaged, high-quality components, and to transform software elements into such reusable components for future reuse.
- *Extendibility*: developing software that is truly *soft* — easy to adapt to the inevitable and frequent changes of requirements and other constraints.
- *Portability*: freeing developers from machine and operating system peculiarities, and enabling them to produce software that will run on many different platforms.
- *Maintainability*: yielding software that is clear, readable, well structured, and easy to continue enhancing and adapting.

1.2 GENERAL PROPERTIES

Here is an overview of the facilities supported by Eiffel:

- Completely *object-oriented* approach. Eiffel is a full-fledged application of object technology, not a “hybrid” of O-O and traditional concepts.
- *External interfaces*. Eiffel is a software composition tool and is easily interfaced with software written in lower-level languages such as C, C++ and Java.
- *Full lifecycle support*. Eiffel is applicable throughout the development process, including analysis, design, implementation and maintenance.
- *Classes* as the basic structuring tool. A class is the description of a set of run-time objects, specified through the applicable operations and abstract properties. An Eiffel system is made entirely of classes, serving as the only module mechanism.
- *Consistent type system*. Every type is based on a class, including basic types such as integer, boolean, real, character, string, array.
- *Design by Contract*. Every system component can be accompanied by a precise specification of its abstract properties, governing its internal operation and its interaction with other components.
- *Assertions*. The method and notation support writing the logical properties of object states, to express the terms of the contracts. These properties, known as assertions, can be monitored at run-time for testing and quality assurance. They also serve as documentation mechanism. Assertions include preconditions, postconditions, class invariants, loop invariants, and are also used in “check instructions”.
- *Exception handling*. Abnormal conditions, such as unexpected operating system signals or more generally a contract violation, can be caught and corrected.
- *Information hiding*. Each class author decides, for each feature, whether it is available to all client classes, to specific clients only, or just for internal purposes.
- *Self-documentation*. The notation is so designed as to enable environment tools to produce abstract views of classes and systems, textual or graphical, and suitable for reusers, maintainers and client authors.
- *Inheritance*. One can define a class as extension or specialization of others.

- *Redefinition*. An inherited feature (operation) can be given a different implementation or signature.
- *Explicit redefinition*. Any feature redefinition must be explicitly stated.
- *Subcontracting*. Redefinition rules require new assertions to be compatible with inherited ones.
- *Deferred features and classes*. It is possible for a feature, and the enclosing class, to be specified — including with assertions — but not implemented. Deferred classes are also known as abstract classes.
- *Polymorphism*. An entity (variable, argument etc.) can become attached to objects of many different types.
- *Dynamic binding*. Calling a feature on an object always triggers the version of the feature specifically adapted to that object, even in the presence of polymorphism and redefinition.
- *Static typing*. A compiler can check statically that all type combinations will be valid, so that no run-time situation will occur in which an attempt will be made to apply an inexistent feature to an object.
- *Assignment attempt* (“type narrowing”). It is possible to check at run time whether the type of an object conforms to a certain expectation, for example if the object comes from a database or a network.
- *Multiple inheritance*. A class can inherit from any number of others.
- *Feature renaming*. To remove name clashes under multiple inheritance, or to give locally better names, a class can give a new name to an inherited feature.
- *Repeated inheritance: sharing and replication*. If, as a result of multiple inheritance, a class inherits from another through two or more paths, the class author can specify, for each repeatedly inherited feature, that it yields either one feature (sharing) or two (replication).
- *No ambiguity under repeated inheritance*. Conflicting redefinitions under repeated inheritance are resolved through a “selection” mechanism.
- *Unconstrained genericity*. A class can be parameterized, or “generic”, to describe containers of objects of an arbitrary type.
- *Constrained genericity*. A generic class can be declared with a generic constraint, to indicate that the corresponding types must satisfy some properties, such as the presence of a particular operation.
- *Garbage collection*. The dynamic model is designed so that memory reclamation, in a supporting environment, can be automatic rather than programmer-controlled.
- *No-leak modular structure*. All software is built out of classes, with only two inter-class relations, client and inheritance.
- *Once routines*. A feature can be declared as “once”, so that it is executed only for its first call, subsequently returning always the same result (if required). This serves as a convenient initialization mechanism, and for shared objects. You may also specify “once” to mean “once per thread” or “once per instance of the class”.

- *Standardized library.* The Kernel Library, providing essential abstractions, is standardized across implementations.
- *Other libraries.* Eiffel development is largely based on high-quality libraries covering many common needs of software development, from general algorithms and data structures to networking and databases.

It is also useful, as in any design, to list some of what is **not** present in Eiffel. The approach is indeed based on a small number of coherent concepts so as to remain easy to master. Eiffel typically takes a few hours to a few days to learn, and users seldom need to return to the reference manual once they have understood the basic concepts. In fact the description given in the present chapter is, save for a few details, essentially complete. Part of this simplicity results from the explicit decision to exclude a number of possible facilities:

- *No global variables,* which would break the modularity of systems and hamper extendibility, reusability and reliability.
- *No union types* (or record type with variants), which force the explicit enumeration of all variants; in contrast, inheritance is an open mechanism which permits the addition of variants at any time without changing existing code.
- *No in-class overloading* which, by assigning the same name to different features within a single context, causes confusions, errors, and conflicts with object-oriented mechanisms such as dynamic binding. (Dynamic binding itself is a powerful form of inter-class overloading, without any of these dangers.)
- *No goto instructions* or similar control structures (break, exit, multiple-exit loops) which break the simplicity of the control flow and make it harder or impossible to reason about the software (in particular through loop invariants and variants).
- *No exceptions to the type rules.* To be credible, a type system must not allow unchecked “casts” converting from a type to another. (Safe cast-like operations are available through assignment attempt.)
- *No side-effect expression operators* confusing computation and modification.
- *No low-level pointers, no pointer arithmetic,* a well-known source of bugs. (There is however a type *POINTER*, used for interfacing Eiffel with C and other languages.)

1.3 THE SOFTWARE PROCESS IN EIFFEL

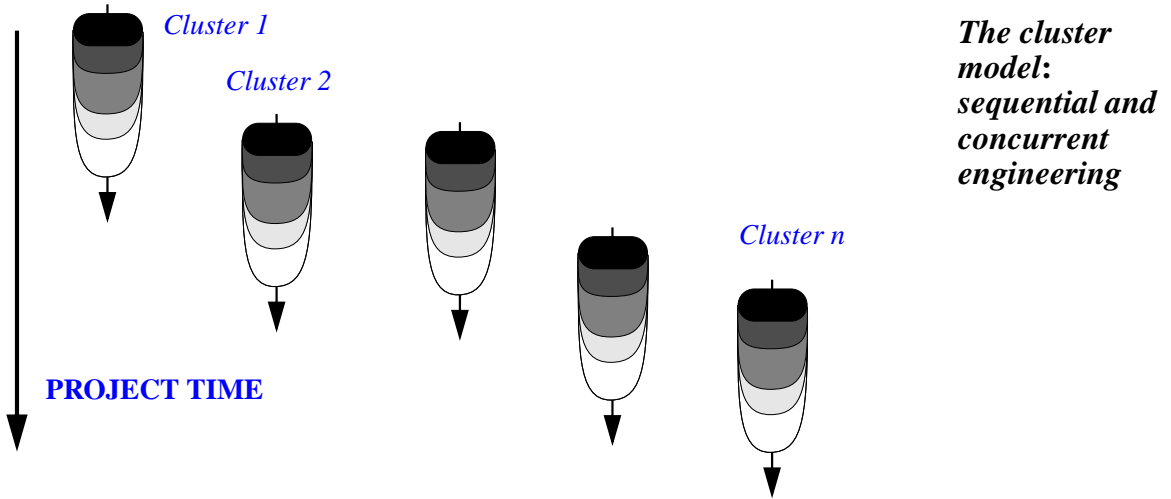
Eiffel, as noted, supports the entire lifecycle. The underlying view of the system development lifecycle is radically different not only from the traditional “Waterfall” model (implying a sequence of discrete steps, such as analysis, global design, detailed design, implementation, separated by major changes of method and notation) but also from its more recent variants such as the spiral model or “rapid prototyping”, which remain predicated on a synchronous, full-product process, and retain the gaps between successive steps.

Clearly, not everyone using Eiffel will follow to the letter the principles outlined below; in fact, some very competent and successful Eiffel developers may disagree with some of them and prefer a somewhat different process model. In the author’s mind, however, these

principles fit best with the language and the rest of the method, even if practical developments may fall short of applying their ideal form.

Clusters and the cluster model

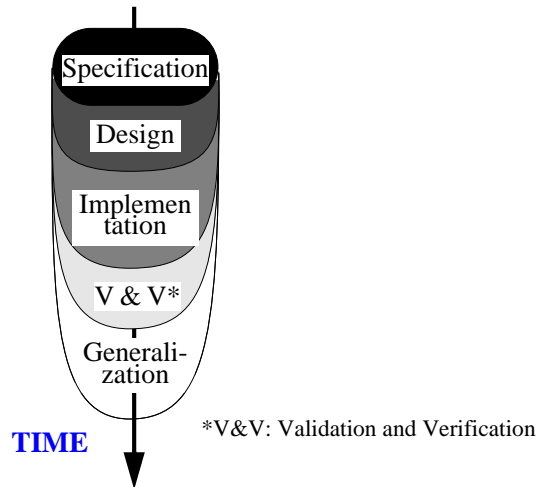
Unlike earlier approaches, the Eiffel model assumes that the system is divided into a number of subsystems or **clusters**. It keeps from the Waterfall a sequential approach to the development of each cluster (without the gaps), but promotes **concurrent engineering** for the overall process, as suggested by the following picture:.



The Eiffel techniques developed below, in particular information hiding and Design by Contract, make the concurrent engineering process possible by letting the clusters rely on other clusters through clearly defined interfaces, strictly limiting the amount of knowledge that must be acquired about a cluster to use it, and permitting separate testing. When the inevitable surprises of a project happen, the project leader can take advantage of the model's flexibility, advancing or delaying various clusters and steps through dynamic reallocation of resources.

Each of the individual cluster lifecycles is based on a continuous progression of activities, from the more abstract to the more implementation-oriented:

This picture should be understood as describing a process of accretion (as *Individual cluster lifecycle* stalactite), where each step *enriches* the results of the previous one. Unlike traditional software development which emphasize the multiplicity of software products — analysis document, global detailed design documents, program, maintenance reports ... —, the principle is here the software as a **single product** which will be repeatedly refined, extended and improved. The Eiffel language supports this view by providing high-level notations that can be used throughout the lifecycle, from the most general and software-independent activities of modeling to the most exacting details of implementation tuned for optimal run-time performance.



These properties make Eiffel span the scope of both “object-oriented methods” (whereas most such methods do not yield an executable result) and “programming languages” (whereas most such languages are not suitable for design and analysis).

Seamlessness and reversibility

The preceding ideas define the **seamless approach** embodied by Eiffel. With seamlessness goes **reversibility**: the ability to go back, even late in the process, to earlier stages. Because the developers work on a single product, they can take advantages of bouts of late wisdom — such as a great idea for adding a new function, discovered only at implementation time — and integrate them in the product. Traditional approaches tend to discourage reversibility because it is difficult to guarantee that the analysis and design will be updated with the late changes. With the single-product principle, this is much easier to achieve.

Seamlessness and reversibility enhance extendibility by providing a direct mapping from the structure of the solution to the structure of the problem description, making it much easier to take care of customers’ change requests quickly and efficiently. They promote reliability, by avoiding possible misunderstandings between customers’ and developers’ views. They are obviously a major boost to maintainability. More generally, they yield a smooth, consistent software process that is particular favorable to both quality and productivity.

Generalization and reuse

The latest part of the cluster lifecycles, Generalization, is unheard of in traditional models. It is meant to prepare the results of a cluster for reuse across projects by looking for elements of general applicability, and transform them for inclusion in libraries.

Of course not all companies using the method will be ready to include this phase in their lifecycles. But those which do will see the reusability of their software greatly improved.

Constant availability

Complementing the preceding principles is the idea that, in the cluster lifecycle, the development team (under the responsibility of the project leader) should at all times maintain a *current working demo* which, although covering only a part of the final system, works well, and can be demonstrated or — after a certain stage — shipped as an early release. It is not a “prototype” in the sense of a mockup meant to be thrown away, but an initial iteration towards the final product; the successive iterations will progress continuously towards until they become that final product.

Compilation technology

The preceding goals benefit from the ability to check frequently that the current iteration is correct and robust. Eiffel compiler writers have developed considerable effort to supporting efficient compilation mechanisms, such as ISE’s **Melting Ice Technology** which ensures immediate recompilation after a change. The recompilation time is a function of the size of the changes, never of the system’s overall size. Even for a system of several thousand classes and several hundred thousand lines, the time to get restarted after a change to a few classes is, on a typical modern computer, a few seconds.

Such a “quick melt” (recompilation) will immediately catch (along with any syntax errors) the type errors — often the symptoms of conceptual errors that, if left undetected, could cause grave damage later in the process or even during operation. Once the type errors have been corrected, the developers should start testing the new functionalities, relying on the power of **assertions** — explained in [1.8](#) — to kill the bugs while they are still larvae. Such extensive unit and system testing, constantly interleaved with development, plays an important part in making sure that the “current demo” is trustworthy, and will eventually yield a correct and robust product.

Quality and functionality

Throughout the process, the method suggests maintaining a constant **quality** level: apply all the style rules, put in all the assertions, handle erroneous cases (rather than the all too common practice of thinking that one will “make the product robust” later on), enforce the proper architecture. This applies to all the quality factors except possibly reusability (since one may not know ahead of time how best to generalize a component, and trying to make everything fully general may conflict with solving the specific problem at hand quickly). All that varies is **functionality**: as the project progresses and clusters come into place, more and more of the final product’s intended coverage becomes available. The only question — for example to answer the more practical one “Can we ship something yet?” — is “Do we cover enough?”, never “Is it good enough?” (as in “Will it not crash?”).

Of course not everyone using Eiffel can, any more than in another approach, guarantee that the ideal just presented will always hold. But it is the theoretical scheme to which the method tends. It explains Eiffel’s emphasis on getting everything right: the grandiose and the mundane, the structure and the details. Regarding the details, the Eiffel books cited in the

bibliography include many rules, some petty at first sight, about such low-level aspects as the choice of names for classes and features (including their grammatical categories), the indentation of software texts, the style for comments (including the presence or absence of a final period), the use of spaces. Applying these rules does not, of course, guarantee quality; but they are part of a quality-oriented process, along with the more ambitious principles of design. In addition they are particularly important for the construction of quality libraries, one of the central goals of Eiffel.

Whenever they are compatible with the space constraints, the present chapter and the rest of this book apply these rules to their Eiffel examples.

1.4 HELLO WORLD

When discovering any approach to software construction, however ambitious its goals, it is reassuring to see first a small example of the big picture — a complete program to print the famous “Hello World” string. Here is how to perform this fascinating task in the Eiffel notation.

You write a class *HELLO* with a single procedure, say *make*, also serving as creation procedure. Here is a minimal version of the class:

```
class HELLO creation make feature
  make
  do io.put_string ("Hello World%N") end
end
```

In practice, however, the recommended Eiffel style rules suggest a better documented version:

```
note
  description: "Root for trivial system printing a message"
class HELLO creation
  make
feature
  make
  -- Print a simple message.
  do
    io.put_string ("Hello World")
    io.put_new_line
  end
end
```

The two versions perform identically; the following comments will cover the more complete second one.

The **note** clause does not affect execution semantics; use it to associate annotations with the class, so that browsers and other indexing and retrieval tools can help users in search of

reusable components satisfying certain properties. Here there is a single annotation entry, *description*.

The name of the class is *HELLO*. Any class may contain “features”; *HELLO* has just one, called *make*. The **creation** clause indicates that *make* is a “creation procedure”, that is to say an operation to be executed at class instantiation time. The class could have any number of creation procedures.

The definition of that creation procedure, *make*, appears in the **feature** clause. Again there can be any number of such clauses (to separate features into logical categories), and each one of them can contain any number of feature declarations. Here we have only one.

The line starting with `--` (two dash signs) is a comment; more precisely it is a “header comment”, which style rules invite software developers to write for every such feature, just after the **is**. (As will be seen in “[The short form of a class](#)”, page 40, environment tools know about this convention and use it to include the header comment in the automatically generated class documentation.)

The body of the feature is introduced by the **do** keyword and terminated by **end**. It consists of two output instructions. They both use *io*, a generally available reference to an object that provides access to standard input and output mechanisms; the notation *io.f*, for some feature *f* of the corresponding library class (*STD_FILES*), means “apply *f* to *io*”. Here we use two such features:

- *put_string* outputs a string, passed as argument, here “*Hello World*”.
- *put_new_line* terminates the line.

Rather than using a call to *put_new_line*, the first version simply integrates a new-line character, denoted as *%N*, at the end of the string. Either technique is acceptable.

To execute the software and print *Hello World*, you need to construct a small “system” (the term preferred to “program” in Eiffel, to emphasize the idea of building software by assembly of reusable components) and designate class *HELLO* as the system’s **root class**, with *make* being specified as the system’s **root procedure**. Eiffel environments provide simple ways to construct an **Ace file** that specifies the root class, the root creation procedure, and other compilation options. In ISE Eiffel, for example, an interactive tool lets you enter the names of these two elements, and builds the Ace file for you, with all compilation options initialized to the most common defaults. You then click the “Melt” (quick compile) button to compile the system; after a few seconds, you are ready to click the “Run” button, which will cause execution of the system. This outputs *Hello World* on the appropriate medium: a Console on Windows or OS/2, the starting window on Unix or VMS.

1.5 THE STATIC PICTURE: SYSTEM ORGANIZATION

We now look at the overall organization of Eiffel software.

Systems

An Eiffel system is a collection of classes, one of which is designated as the root class. One of the features of the root class, which must be one of its creation procedures, is designated as the root procedure.

To execute such a system is to create an instance of the root class (an object created according to the class description) and to execute the root procedure. In anything more significant than “Hello World” systems, this will create new objects and apply features to them, in turn triggering further creations and feature calls.

For the system to make sense, it must contain all the classes on which the root **depends** directly or indirectly. A class *B* depends on a class *A* if it is either a **client** of *A*, that is to say uses objects of type *A*, or an **heir** of *A*, that is to say extends or specializes *A*. (These two relations, client and inheritance, are described in more detail below.)

The rest of section [1.5](#) will describe the nuts and bolts of an Eiffel system and of its execution.

Classes

The notion of class is central to the Eiffel approach. A class is the description of a type of runtime data structures (*objects*), characterized by common operations (*features*) and properties. Examples of classes include:

- In a banking system, a class *ACCOUNT* may have features such as *deposit*, adding a certain amount to an account, *all_deposits*, yielding the list of deposits since the account’s opening, and *balance*, yielding the current balance, with properties stating that *deposit* must add an element to the *all_deposits* list and update *balance* by adding the sum deposited, and that the current value of *balance* must be consistent with the lists of deposits and withdrawals.
- A class *COMMAND* in an interactive system of any kind may have features such as *execute* and *undo*, as well as a feature *undoable* which indicates whether a command can be undone, with the property that *undo* is only applicable if *undoable* yields the value true.
- A class *LINKED_LIST* may have features such as *put*, which adds an element to a list and *count* yielding the number of elements in the list, with properties stating that *put* increases *count* by one and that *count* is always non-negative.

We may characterize the first of these examples as an analysis class, directly modeling objects from the application domain; the second one as a design class, describing a high-level solution; and the third as an implementation class, reused whenever possible from a library such as EiffelBase. In Eiffel, however, there is no strict distinction between these categories; it is part of the approach’s seamlessness that the same notion of class, and the associated concepts, may be used at all levels of the software development process.

Class relations

Two relations can exist between classes:

- You can define a class *C* as a **client** of a class *A* to enable the features of *C* to rely on objects of type *A*.
- You may define a class *B* as an **heir** of a class *A* to provide *B* with all the features and properties of *A*, letting *B* add its own features and properties and modify some of the inherited features if appropriate.

If *C* is a client of *A*, *A* is a **supplier** of *C*. If *B* is an heir of *A*, *A* is a **parent** of *B*. A **descendant** of *A* is either *A* itself or, recursively, a descendant of an heir of *A*; in more informal terms a descendant is a direct or indirect heir, or the class itself. To exclude *A* itself we talk of **proper descendant**. In the reverse direction the terms are **ancestor** and **proper ancestor**.

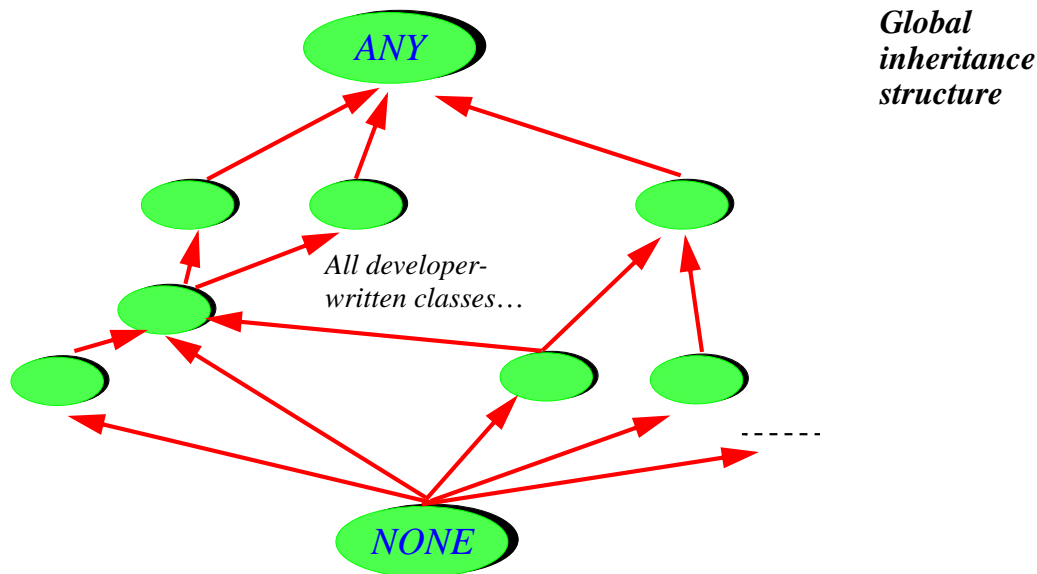
The client relation can be cyclic; an example involving a cycle would be classes *PERSON* and *HOUSE*, modeling the corresponding informal everyday “object” types and expressing the properties that every person has a home and every home has an architect. The inheritance (heir) relation may not include any cycle.

In modeling terms, client roughly represents the relation “has” and heir roughly represents “is”. For example we may use Eiffel classes to model a certain system and express that every child *has* a birth date (client relation) and *is* a person (inheritance).

Distinctive of Eiffel is the rule that classes can only be connected through these two relations. This excludes the behind-the-scenes dependencies often found in other approaches, such as the use of global variables, which jeopardize the modularity of a system. Only through a strict policy of limited and explicit inter-class relations can we achieve the goals of reusability and extendibility.

The global inheritance structure

If you write an Eiffel class, it does not come into a vacuum but fits in a preordained structure, shown in the following figure and involving two library classes: *ANY* and *NONE*.



Any class that does not explicitly inherit from another is considered to inherit from *ANY*, which introduces a number of general-purpose features useful everywhere; examples include copying, cloning and equality testing operations (page [26](#)) and default input-output mechanisms.

NONE is a fictitious class, which is considered to be an heir of any class that has no explicit heir. Since inheritance has no cycles, *NONE* cannot have proper descendants. This makes it useful, as we will see, to specify non-exported features, and to denote the type of void values.

Clusters

Classes are the only form of module in Eiffel. As will be explained in more detail, they also provide the basis for the only form of type. This module-type identification is at the heart of object technology and yields the fundamental simplicity of the Eiffel method.

Efforts to introduce a higher-level notion of module above classes are misguided, as they introduce a whole new set of issues (name space, name visibility, information hiding, separate compilation, module inclusion) to which the solutions would clash with the class-level techniques. This would also hamper reusability (by making it harder to reuse a class by itself) and extendibility.

There is a need, however, for an *organizational* concept: cluster. A cluster is a group of related classes. The cluster is a property of the method, enabling managers to organize the development into teams, but it does not require a specific Eiffel language construct. As we have already seen (section [1.3](#)) it also plays a central role in the lifecycle model.

External software

The subsequent sections will show how to write Eiffel classes with their features. In an Eiffel system, however, not everything has to be written in Eiffel: some features may be **external**, coming from external languages such as C, C++, Java, Fortran or others. For example a feature declaration may appear (in lieu of the forms seen later in this chapter) as

```
file_status (filedesc: INTEGER): INTEGER
    -- Status indicator for filedesc
external
    "C" alias "_fstat"
end
```

to indicate that it is actually an encapsulation of a C function whose original name is `_fstat` (the **alias** clause is optional, but here it is needed because the C name, starting with an underscore, is not valid as an Eiffel identifier).

Similar syntax exists in several Eiffel compilers to interface with C++ classes. ISE Eiffel includes a tool called *Legacy++* which will automatically produce, from a C++ class, an Eiffel class that encapsulates its facilities, making them available to the rest of the Eiffel software as *bona fide* Eiffel features.

These mechanisms illustrate one of the roles of Eiffel: as a system architecting and software composition tool, used at the highest level to produce systems with robust, flexible structures ready for extendibility, reusability and maintainability. In these structures not everything must be written in the Eiffel language: existing software elements and library components can play their part too, the structuring capabilities of Eiffel (classes, information hiding, inheritance, clusters and other techniques seen in this chapter) serving as the overall wrapping mechanism.

1.6 THE DYNAMIC STRUCTURE: EXECUTION MODEL

A system with a certain static structure describes a set of possible executions. The run-time model governs the structure of the data (*objects*) created during such executions.

The properties of the run-time model are not just of interest to implementers; they also involve concepts directly relevant to the needs of system modelers and analysts at the most abstract levels.

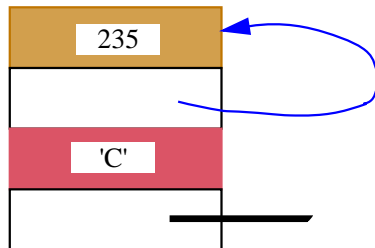
Objects, fields, values and references

A class was defined as the static description of a type of run-time data structures. The data structures described by a class are called **instances** of the class, which in turn is called their **generating class** (or just “generator”). An instance of *ACCOUNT* is a data structure representing a bank account; an instance of *LINKED_LIST* is a data structure representing a linked list.

An **object**, as may be created during the execution of a system, is an instance of some class of the system.

Classes and objects belong to different worlds: a class is an element of the software text; an object is a data structure created during execution. Although it is possible to define a class whose instances represent classes (as class *E_CLASS* in the ISE libraries, used to access properties of classes at run time), this does not eliminate the distinction between a static, compile-time notion, class, and a dynamic, run-time notion, object.

An object is either an atomic object (boolean, character, integer, real) or a composite object made of a number of **fields**, represented by adjacent rectangles on the conventional run-time diagrams:



Composite object

(with 4 fields including self-reference and void reference)

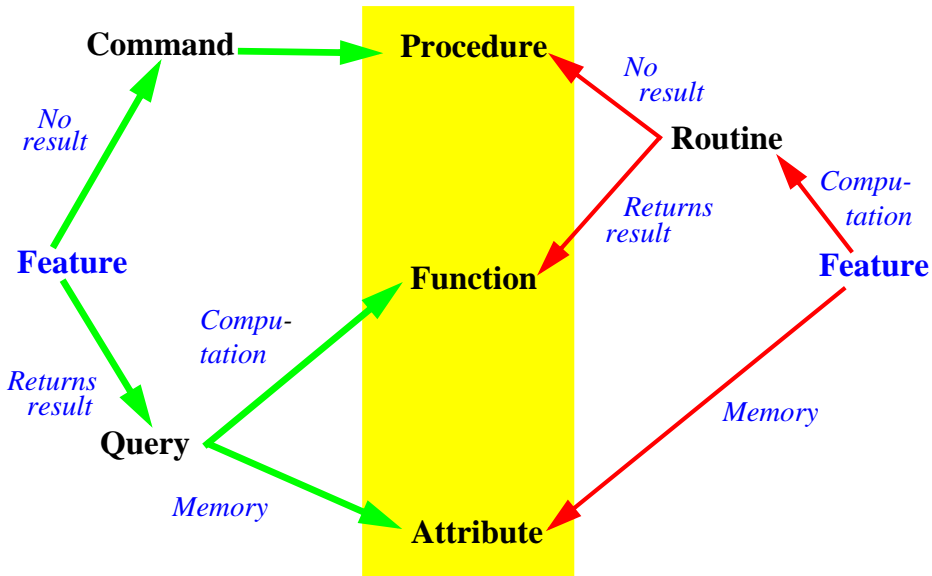
Each field is a **value**. A value can be either an object or an object reference:

- When a field is an object, it will in most cases be an atomic object, as on the figure where the first field from the top is an integer and the third a character. But a field can also be a composite object, in which case it is called a **subobject**.
- A **reference** is either void or uniquely identifies an object, to which it is said to be **attached**. In the preceding the second field from the top is a reference — attached *Feature* case, as represented by the arrow, to the enclosing object itself. The bottom field *categories* reference.

Features

A feature, as noted, is an operation available on instances of a class. A feature can be an **attribute** or a **routine**. This classification, which can be followed by starting from the figure on the figure above, is based on implementation considerations:

(Two complementary classifications)



- An attribute is a feature implemented through memory: it describes a field that will be found in all instances of the class. For example class *ACCOUNT* may have an attribute *balance*; then all instances of the class will have a corresponding field containing each account's current balance.
- A routine describes a computation applicable to all instances of the class. *ACCOUNT* may have a routine *withdraw*.

Routines are further classified into **functions**, which will return a result, and **procedures**, which will not. Routine *withdraw* will be a procedure; an example of function may be *highest_deposit*, which returns the highest deposit made so far to the account.

From the viewpoint of classes relying on a certain class (its **clients**), the more relevant classification is the one coming from the left on the preceding figure:

- **Commands** have no result, and can modify an object. They can only be procedures.
- **Queries** have a result: they return information about an object. They can be implemented as either attributes (by reserving space for the corresponding information in each instance of the class, a memory-based solution) or functions (a computation-based solution). An attribute is only possible for a query without argument, such as *balance*; a query with arguments, such as *balance_on* (*d*), returning the balance at date *d*, can only be a function.

From the outside, there is no difference between a query implemented as an attribute and one implemented as a function: to obtain the balance of an account *a*, you will always write *a.balance*.

In the implementation suggested above, *a* is an attribute, so that the notation denotes an access to the corresponding object field. But it is also possible to implement *a* as a function,

whose algorithm will explore the lists of deposits and withdrawals and compute their accumulated value. To the clients of the class, and in the official class documentation as produced by the environment tools, the difference is not visible.

This principle of **Uniform Access** supports Eiffel's goals of extendibility, reusability and maintainability: you can change the implementation without affecting clients; and you can reuse a class without having to know the details of its features' implementations.

A simple class

The following simple class text illustrates the preceding concepts

```
note
    description: "Simple bank accounts"
class
    ACCOUNT
feature -- Access
    balance: INTEGER
        -- Current balance
    deposit_count: INTEGER
        -- Number of deposits made since opening
    do
        if all_deposits /= Void then
            Result := all_deposits.count
        end
    end
feature -- Element change
    deposit (sum: INTEGER)
        -- Add sum to account.
    do
        if all_deposits = Void then
            create all_deposits
        end
        all_deposits.extend (sum)
        balance := balance + sum
    end
```



```

feature {NONE} -- Implementation
  all_deposits: DEPOSIT_LIST
    -- List of deposits since account's opening.
invariant
  consistent_balance:
    (all_deposits /= Void) implies (balance = all_deposits.total)
  zero_if_no_deposits:
    (all_deposits = Void) implies (balance = 0)
end

```

(The `{NONE}` qualifier and the `invariant` clause, used here to make the example closer to a real class, will be explained shortly. `DEPOSIT_LIST` refers to another class, which can be written separately using library classes.)

The category to which each feature belong is easy to deduce from its syntactic appearance. Here only `deposit` and `deposit_count`, which include a `do ...` clause, are routines; `balance` and `all_deposits`, which are simply declared with a type, are attributes. Note that even for attributes it is recommended to have a header comment.

Routine `deposit_count` is declared as returning a result (of type `INTEGER`); so it is a function. Routine `deposit` has no such result and hence is a procedure.

Creating and initializing objects

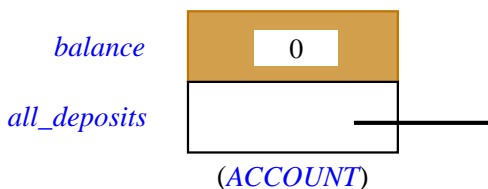
Classes, as noted, are a static notion. Objects appear at run time; they are created explicitly. The instruction that creates an object of type `ACCOUNT` and attaches it to `x` is written

```

create x

```

assuming that `x` has been declared of type `ACCOUNT`. Such an instruction must be in a routine of some class — the only place where instructions can appear — and its effect at run time will be threefold: create a new object of type `ACCOUNT`; initialize its fields to default values; and attach the value of `x` to it. Here the object will have two fields corresponding to the two attributes of the generating class: an integer for `balance`, which will be initialized to 0, and a reference for `all_deposits`, which will be initialized to a void reference:



*Instance with
fields
initialized to
defaults*

The default initialization values are specified for all possible types:

Type	Default value
<i>INTEGER, REAL</i>	Zero
<i>BOOLEAN</i>	False
<i>CHARACTER</i>	Null
Reference types (such as <i>ACCOUNT</i> and <i>DEPOSIT_LIST</i>)	Void reference
Composite expanded types (see next)	Same rules, applied recursively to all fields

It is possible to override the initialization values by providing — as in the earlier example of class *HELLO* — one or more creation procedures. For example we might change *ACCOUNT* to make sure that every account is created with an initial deposit:

```

note
  description: "Simple bank accounts, initialized with a first deposit"
class
  ACCOUNT1
creation
  make
feature -- Initialization
  make (sum: INTEGER)
    -- Initialize account with sum.
  do
    deposit (sum)
  end
  ... The rest of the class as for ACCOUNT ...
end

```

The newly added **creation** clause will list one or more (here just one) procedures of the class. In this case the original form **create** *x* is not valid any more for creating an instance of *ACCOUNT1*; a creation instruction must be of a form such as

```
create x.make (2000)
```

known as a creation call. Such a creation call will have the same effect as the original form (creation, initialization, attachment to *x*) followed by the effect of calling the selected creation procedure, which here will call *deposit* with the given argument.

Note that in this example all that *make* does is to call *deposit*. So an alternative to introducing a new procedure *make* would have been simply to introduce a creation clause of

the form **creation** *deposit*, elevating *deposit* to the status of creation procedure. Then a creation call would be of the form **create** *x.deposit* (2000).

Entities

The example assumed *x* declared of type *ACCOUNT* (or *ACCOUNTI*). Such an *x* is an example of entity, a notion generalizing the well-known concept of variable. An entity is a name that appears in a class text to represent possible run-time values (a value being, as defined earlier, an object or a reference). An entity is one of the following:

- An attribute of the enclosing class, such as *balance* and *all_deposits*.
- A formal argument of a routine, such as *sum* for *deposit* and *make*.
- A local Variable declared for the needs of a routine.
- The special entity *Result* in a function.

The third case, local variables, arises when a routine needs some auxiliary values for its computation. Here is an example of the syntax:

```
deposit (sum: INTEGER)
  -- Add sum to account.
  local
    new: AMOUNT
  do
    create new.make (sum)
    all_deposits.extend (new)
    balance := balance + sum
  end
```

This example is a variant of *deposit* for which we assume that the elements of a *DEPOSIT_LIST* such as *all_deposits* are no longer just integers, but objects, instances of a new class, *AMOUNT*. Such an object will contain an integer value, but possibly other information as well. So for the purpose of procedure *deposit* we create an instance of *AMOUNT* and insert it, using procedure *extend*, into the list *all_deposits*. The object is identified through the local Variable *new*, which is only needed within each execution of the routine (as opposed to an attribute, which yields an object field that will remain in existence for as long as the object).

The last case of entity, *Result*, serves to denote, within the body of a function, the final result to be returned by that function. This was illustrated by the function *deposits_count*, which read

```
deposits_count: INTEGER
  -- Number of deposits made since opening (provisional version)
  if all_deposits /= Void then
    Result := all_deposits.count
  end
```

The value returned by any call will be the value of the expression *all_deposits.count* (to be explained in detail shortly) for that call, unless *all_deposits* has value *Void*, denoting a void reference (*/=* is “not equal”).

The default initialization rules seen earlier for attributes (see the table on page 20) also serve to initialize all local variables and *Result* on entry to a routine. So in the last example, if *all_deposits* is void (as in the case on initialization with the class as given so far), *Result* keeps its default value of 0, which will be returned as the result of the function.

Calls

Apart from object creation, the basic computational mechanism, in the object-oriented style of computation represented by Eiffel, is feature call. In its basic form, it appears as

```
target.feature (argument1, ...)
```

where *target* is an entity (it can more generally be an expression), *feature* is a feature name, and there may be zero or more *argument* expressions. In the absence of any *argument* the part in parentheses should be removed.

We have already seen such calls. If the *feature* denotes a procedure, the call is an instruction, as in

```
all_deposits.extend (new)
```

If *feature* denotes a query (function or attribute), the call is an expression, as in the right-hand side of

```
Result := all_deposits.count
```

The principle of Uniform Access (page 18) implies that this form is the same for an attribute and for a function without arguments. (The feature used in this example, *count* from class *DEPOSIT_LIST*, could indeed be implemented in either of these two ways: we can keep a *count* field in each list, updating it for each insertion and removal; or we can compute *count*, whenever requested, by traversing the list to count the number of elements.)

In the case of a routine with arguments — procedure or function — the routine will be declared, in its class, as

```
feature (formal: TYPE1; ...)
do ... end
```

meaning that, at the time of each call, the value of each formal will be set to the corresponding actual (*formal* to *argument*1 and so on). In the routine body, it is not permitted to change the value of a formal argument, although it is possible to change the value of an attached object through a procedure call such as *formal*.*some_procedure* (...).

Infix and prefix notation

Basic types such as *INTEGER* are, as noted, part of Eiffel's uniform type system, and so are declared as classes (part of the Kernel Library). *INTEGER*, for example, is characterized by the features describing integer operations: plus, minus, times, division, less than, and so on.

With the dot notation seen so far, this would imply that simple arithmetic operations would have to be written with a syntax such as *i.plus* (*j*) instead of the usual *i + j*. This would be awkward. Infix and prefix features solve the problem, reconciling the object-oriented view of computation with common notational practices of mathematics. The addition function is declared in class *INTEGER* as

```
plus alias "+" (other: INTEGER): INTEGER
do ... end
```

Such a feature has all the properties and prerogatives of a normal “identifier” feature, except for the form of the calls, which is infix, as in *i + j*, rather than using dot notation. An infix feature must be a function, and take exactly one argument. Similarly, a function can be declared as *negated alias* “-”, with no argument, permitting calls of the form *-3* rather than *(3).negated*.

Predefined library classes covering basic types such as *INTEGER*, *CHARACTER*, *BOOLEAN*, *REAL* are known to the Eiffel compiler, so that a call of the form *i + j*, although conceptually equivalent to a routine call, can be processed just as efficiently as the corresponding arithmetic expression in an ordinary programming language. This brings the best of both worlds: conceptual simplicity, enabling Eiffel developers, when they want to, to think of integers and the like as objects; and efficiency as good as in lower-level approaches.

Infix and prefix features can be used in any class, not just predefined classes for basic types. For example a graphics class could use the name *plus alias* “|” for a function computing the distance between two points, to be used in expressions such as *point1* | *point2*.

Type declaration

Every entity in Eiffel is declared as being of a certain type, using the syntax already encountered in the above examples:

entity_name: *TYPE_NAME*

This applies to attributes, formal arguments of routines and local variables. The result type is also declared for a function, as in the earlier example

deposit_count: *INTEGER is ...*

Here the type also serves as the type implicitly declared for *Result* in the function's body.

What is a type? With the elements seen so far, every type is a **class**. *INTEGER*, used in the declaration of *deposits_count*, is, as we have seen, a library class; and the declaration *all_deposits*: *DEPOSIT_LIST* assumes the existence of a class *DEPOSIT_LIST*.

Three mechanisms introduced below — expanded types (page 24), genericity (page 32) and anchored declarations (page 70)— will generalize the notion of type slightly. But they do not change the fundamental property that **every type is based on a class**, called the type's **base class**. In the examples seen so far, each type *is* a class, serving as its own base class. An instance of a class *C* is also called an “object of type *C*”.

Type categories

It was noted above that a value is either an object or a reference. This corresponds to two kinds of type: reference types and expanded types.

If a class is declared as just

class *CLASS_NAME ...*

it defines a reference type. The entities declared of that type will denote references. So in the declaration

x: *ACCOUNT*

the possible run-time values for *x* are references, which will be either void or attached to instances of class *ACCOUNT*.

Instead of **class**, however, you can use the double keyword **expanded class**, as in the library class definition

```

note
  description: "Integer values"
expanded class
  INTEGER
feature -- Basic operations
  plus alias "+" (other: INTEGER): INTEGER
  do ... end
  ... Other feature declarations ...
end

```

In this case the value of an entity declared as *n: INTEGER* is not a reference to an object, but the object itself — in this case an atomic object, an integer value.

Note that the value of an entity of expanded type can never be void; only a reference can. Extending the earlier terminology, an expanded entity is always **attached to** an object, atomic (as in the case of *n: INTEGER*) or composite (as in *x: EC* for some expanded class *EC*).

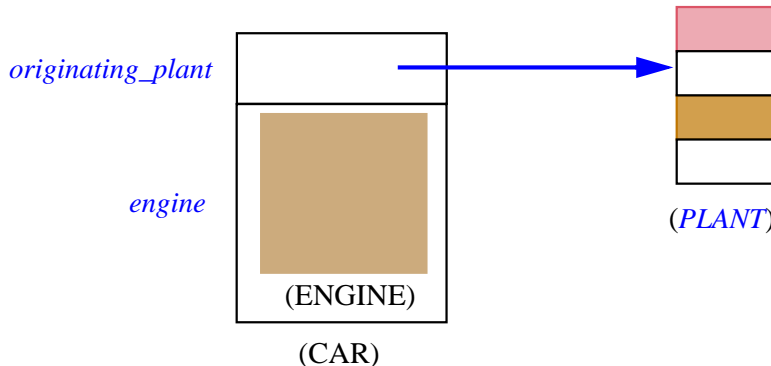
Expanded declarations make it possible to construct composite objects with subobjects, as in the following abbreviated class declaration (Notes clause and routines omitted):

```

class CAR feature
  engine: ENGINE
  originating_plant: PLANT
end

```

The following illustration shows the structure of a typical instance of *CAR*:



This example also illustrates that the distinction between expanded and reference types is important not just for system implementation purposes but for high-level system modeling as well. To understand the conceptual distinction, note that many cars will share the same *originating_plant*, but an *engine* belongs to just one car. References represent the modeling

relation “knows about”; subobjects, as permitted by expanded types, represent the *Composite* “has part”, also known as aggregation. The key difference is that sharing is possible in the former case but not in the latter.

*object with
reference and
subobject*

Basic operations

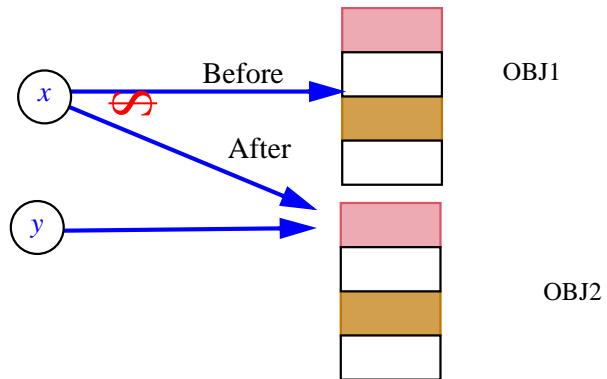
The following basic operations are available on entities and expressions.

Assignment uses the symbol `:=`. The assignment instruction

```
x := y
```

updates the value of `x` to be the same as that of `y`. This means that:

- For entities of reference types, the value of `x` will be a void reference if the value of `y` is void, and otherwise `x` will be attached to the same object OBJ2 as `y`:



*Effect of
reference
reattachment
x := y*

- For entities of expanded types, the values are objects; the object attached to `x` will be overwritten with the contents of the object attached to `y`. In the case of atomic objects, as in `n := 3` with the declaration `n: INTEGER`, this has the expected effect of assigning to `n` the integer value 3; in the case of composite objects, this overwrites the fields for `x`, one by one, with the corresponding `y` fields.

To copy an object, use `x.copy(y)` which assumes that both `x` and `y` are non-void, and copies the contents of `y`'s attached object onto those of `x`'s. Note that for expanded entities the effect is the same as that of the assignment `x := y`.

A variant of the `copy` operation is `clone`. The expression `clone(y)` produces a newly created object, initialized with a copy of the object attached to `y`, or a void value if `y` itself is void. For a reference type (the only interesting case) the returned result for non-void `y` is a reference to the new object. In other words `clone` can be viewed as a function that performs

```
create Result
Result.copy(y)
```


So in the assignment $x := \text{clone}(y)$, assuming both entities of reference types and y not void, will attach x to a **new object** identical to y 's attached object, as opposed to the assignment $x := y$ which attaches x to the **same object** as y .

To determine whether two values are equal, use the expression $x = y$. (The expression $x / = y$ will yield true if they are **not** equal.) For references, the equality comparison will yield true if the values are either both void or both attached to the same object; this is the case in the last figure in the state *after* the assignment, but not before.

As with assignment, there is also a variant that works on objects rather than references: $x.\text{is_equal}(y)$ will return true when x and y are both non-void and attached to field-by-field identical objects. This can be true even when $x = y$ is not, for example, in the figure, *before* the assignment, if the two objects shown are field-by-field equal.

A more general variant of *is_equal* is used under the form *equal* (x, y). This is always defined, even if x is void, returning true whenever *is_equal* would but also if x and y are both void. (In contrast, $x.\text{is_equal}(y)$ is not defined for void x and would, if evaluated, yield an exception as explained in [“Exception handling”, page 42](#) below.)

The predefined feature *Void* denotes a void reference. So you can make x void through the assignment $x := \text{Void}$, and test whether it is void through **if $x = \text{Void}$ then ...** The type of *Void* is *NONE*, the “least interesting” class (as seen in [“The global inheritance structure”, page 14](#)).

The features introduced in this section — *copy*, *clone*, *is_equal*, *equal*, *Void* — are not language constructs but features defined in the highest-level class *ANY* (page 14) and hence available to all classes. *Void* is of type *NONE*. Using the redefinition mechanisms to be seen in the discussion of inheritance, a class can redefine *copy* and *is_equal* to describe specific notions of copy and equality. (The assertions will ensure that the two remain compatible: after $x.\text{copy}(y)$, the property $x.\text{is_equal}(y)$ must always be true.) Redefining *copy* automatically causes *clone* to follow, and redefining *is_equal* automatically causes *equal* to follow. *Void* too should not be redefined. To avoid any mistake these features are declared as “frozen” — not redefinable. To guarantee the original, non-redefined semantics you may use the frozen variants *standard_copy*, *standard_clone*, *standard_equal* and so on.

Deep operations and persistence

Feature *clone* only duplicates one object. If some of the fields of that object are references to other objects, the references themselves will be copied, not those other objects.

It is useful, in some cases, to duplicate not just one object but an entire object structure. The expression *deep_clone* (y) achieves this goal: assuming non-void y , it will produce a duplicate not just of the object attached to y but of the entire object structure starting at that object. The mechanism of course respects all the possible details of that structure, such as cyclic reference chains. As earlier features, *deep_clone* comes from class *ANY*.

A related mechanism provides a powerful **persistence** facility. A call of the form

$x.\text{store}(\text{Some_file_or_network_connection})$

(using the conventions of ISE Eiffel) will store a copy of the entire object structure starting at x , under a suitable representation. Like *deep_clone*, procedure *store* will follow all references to the end and maintain the properties of the structure. The function *retrieved* can then be used — in the same system, or another — to recreate the structure from the stored version.

As the name suggests, *Some_file_or_network_connection* can be an external medium of various possible kinds, not just a file but possibly a database or network. ISE's EiffelNet client-server library indeed uses the *store-retrieved* mechanism to exchange object structures over a network, between compatible or different machine architectures, for example a Windows client and a Unix server.

Memory management

Reference reattachments $x := y$ of the form illustrated by the figure on page 26 can cause objects to become unreachable. This is the case for the object identified as OBJ2 on that figure (the object to which x was attached before the assignment) if no other reference was attached to it.

In all but toy systems, it is essential to reclaim the memory that has been allocated for such objects; otherwise memory usage could grow forever, as a result of creation instructions *create* $x \dots$ and calls to *clone* and the like, leading to thrashing and eventually to catastrophic termination.

Unlike some other approaches, the Eiffel method suggests that the task of detecting and reclaiming such unused object space should be handled by an automatic mechanism (part of the Eiffel run-time environment), not manually by developers (through calls to procedures such as Pascal's *dispose* and C/C++'s *free*). The arguments for this view are:

- **Convenience:** handling memory reclamation manually can add enormous complication to the software, especially when — as is often the case in object-oriented development — the system manipulates complex run-time data structures with many links and cycles.
- **Reliability:** memory management errors, such as the incorrect reclamation of an object that is still referenced by a distant part of the structure, are a notorious source of particularly dangerous and hard-to-correct bugs.

Eiffel environments have developed sophisticated **garbage collectors** which efficiently handle the automatic reclamation process, while causing no visible degradation of a system's performance and response time.

Reliance on automatic garbage collection is a key part of the Eiffel method's contribution to both ease of development and software reliability.

Information hiding and the call rule

The basic form of computation, it has been noted, is a call of the form *target.feature* (...). This is only meaningful if *feature* denotes a feature of the generating class of the object to which *target* (assumed to be non-void) is attached. The precise rule is the following:

Feature Call rule

A call of the form *target.feature* (...) appearing in a class *C* is only valid if *feature* is a feature of the base class of *target*'s type, and is available to *C*.

The first condition simply expresses that if *target* has been declared as *target: A* then *feature* must be the name of one of the features of *A*. The second condition reflects Eiffel's application of the principles of information hiding. A **feature** clause, introducing one or more feature declarations, may not just appear as

```
feature -- Comment identifying the feature category
... Feature declaration ...
... Feature declaration ...
...
```

but also include a list of classes in braces, **feature** {*A*, *B*, ...}, as was illustrated for *ACCOUNT*:

```
feature {NONE} -- Implementation
  all_deposits: DEPOSIT_LIST
  -- List of deposits since account's opening.
```

This form indicates that the features appearing in that clause are only **available** — in the sense of available for calls, as used in the Feature Call rule — to the classes listed. In the example feature *all_deposits* is only available to *NONE*. Because of the global inheritance structure (page 14) this means it is in fact available to no useful client at all, and is equivalent in practice to **feature** { } with an empty class list, but the form listing *NONE* explicitly is more visible and hence preferred.

With this specification a class text including the declaration *acc: ACCOUNT* and a call of the form

```
acc.all_deposits
```

violates the Feature Call rule and will be rejected by the Eiffel compiler.

Besides fully exported features (introduced by **feature** ... without further qualification) and fully secret ones (**feature** { } or **feature** {*NONE*}), it is possible to export features selectively to some specified classes, using the specification **feature** {*A*, *B*, ...} for arbitrary classes *A*, *B*, ... By enabling a group of related classes to provide each other with privileged

access, this Selective Export mechanism is one of the techniques that avoids the much more heavy solution of using meta-modules above the class level (see [“Clusters”, page 14](#)).

Exporting features selectively to a set of classes *A*, *B*, ... also makes them available to the descendants of these classes. So a feature clause beginning with just **feature** is equivalent to one starting with **feature** {*ANY*}.

These rules enable successive feature clauses to specify exports to different clients. In addition, the recommended style, illustrated in the examples of this chapter, suggests writing separate feature clauses — regardless of their use for specifying export privileges — to group features into separate categories. Typical categories, appearing in the order given, are: Initialization (for creation procedures); Access (for general queries); Status report; Status setting; Element change; Implementation (for selectively exported or secret features).

The Feature Call rule is the first of the rules that make Eiffel a **statically typed** approach, where the applicability of operations to objects is verified at compile time rather than during execution. Static typing is one of the principal components of Eiffel’s support for reliability in software development.

Execution scenario

The preceding elements make it possible to understand the overall scheme of an Eiffel system’s execution.

At any time during the execution of a system, one object is the **current object** of the execution, and one of the routines of the system, the **current routine**, is being executed, with the current object as its target. (We will see below how the current object and current routine are determined.) The text of a class, in particular its routines, make constant implicit references to the current object. For example in the instruction

```
balance := balance + sum
```

appearing in the body of procedure *deposit* of class *ACCOUNT*, the name of the attribute *balance*, in both occurrences, denotes the *balance* field of the current object, assumed to be an instance of *ACCOUNT*. In the same way, the procedure body that we have used for the creation procedure *make* in the *ACCOUNT1* variant

```
make (sum: INTEGER)
    -- Initialize account with sum.
    do
        deposit (sum)
    end
```

contains a call to the procedure *deposit*. Contrary to earlier calls written in dot notation as *target.feature (...)*, the call to *deposit* has no explicit target; this means its target is the current object, an instance of *ACCOUNT1*. Such a call is said to be **unqualified**; those using dot notations are **qualified** calls.

Although most uses of the current object are implicit, a class may need to name it explicitly. The predefined expression *Current* is available for that purpose. A typical use, in a routine *merge* (*other*: *ACCOUNT*) of class *ACCOUNT*, would be a test of the form

```

if other = Current then
    report_error ("Error: trying to merge an account with itself!")
else
    ... Normal processing (merging two different accounts) ...
end

```

With these notions it is not hard to define precisely the overall scenario of a system execution by defining what object and routine will, at each instant, be the current object and the current routine:

- Starting a system execution, as we have seen, consists in creating an instance of the root class, the root object, and executing a designated creation procedure, the root procedure, with the root object as its target. The root object is the initial current object, and the root procedure is the initial current procedure.
- From then on only two events can change the current object and current procedure: a qualified routine call; and the termination of a routine.
- In a call of the form *target.routine* (...), *target* denotes a certain object TC. (If *target* is an attribute, TC is the object attached to the corresponding field of the current object, which must be non-void for the call to proceed.) Then TC becomes the new current object. The generating class of TC must, as per the Feature Call rule, contain a routine of name *routine*, which becomes the new current routine.
- When a routine execution terminates, the target object and routine of the most recent non-terminated call (which just before the terminated call were the current object and the current routine) assume again the role of current object and current routine. This does not apply, of course, to the termination of the original root procedure call; in this case the entire execution terminates and there is nothing more to say about it.

Abstraction

The description of assignments stated that in $x := y$ the target x must be an entity. More precisely it must be a **variable** entity; this excludes formal routine arguments: as noted, a routine r (*arg*: *SOME_TYPE*) cannot assign to *arg* (reattaching it to a different object), although it can change the attached objects through calls of the form *arg.procedure* (...).

The restriction to an entity precludes in particular assignments of the form *obj.some_attribute* := *some_value*, since the left-hand side *obj.some_attribute* is an expression (a feature call), not an entity, and you can no more assign to *obj.some_attribute* than to, say, $a + b$ — another expression which is also, formally, a feature call.

To obtain the intended effect of the invalid assignment you may use a procedure call of the form *obj.set_attribute (some_value)*, where the base class of *obj*'s type has defined the procedure

```

set_attribute (v: VALUE_TYPE)
    -- Set value of attribute to v.
do
    attribute := v
end

```

This rule is essential to enforcing the method. Permitting direct assignments to an object's fields would violate all the tenets of information hiding by circumventing the interface carefully crafted by the author of the supplier class. It is the responsibility of each class author to define the exact privileges that the class gives to each of its clients, in particular field modification rights. A field can be totally hidden (when the corresponding attribute is exported to *NONE*); it can be exported in read-only mode (when the attribute is exported, but no procedure that modifies it); it can be exported in free-write mode (as with *set_attribute* if the class exports this procedure); but it can also be exported in restricted-write mode, as with procedure *deposit* of class *ACCOUNT*, which will allow addition of a certain amount to the balance field, but not direct setting of the balance. In such a case, the exported procedures may, thanks to the assertion mechanism reviewed later (1.8), place some further restrictions on the permitted modifications, for example by requiring the withdrawn amount to be positive.

The more general view is that each class describes a well-understood abstraction, for which the class designer decides exactly what operations are permitted. The class documentation (the *short form*, see page 40) makes this view clear to client authors; no violation of that interface is permitted, as it would make a mockery of the principles of object technology. This approach also paves the way for future generalization (page 8) of the most promising components and their inclusion into reusable libraries.

1.7 GENERICITY

Some of the classes that we will need, particularly (but not solely) in libraries, are **container** classes, describing data structures made of a number of objects of the same type, or compatible types. Examples of containers include arrays, stacks and lists. The class *DEPOSIT_LIST* posited in earlier examples describes containers.

It is not hard, with the mechanisms seen so far, to write a class such as *DEPOSIT_LIST*, which would include such features as *count* (query returning the number of elements) and *put* (command to insert a new element).

Most of the operations, however, would be the same for lists of objects other than deposits. To avoid undue replication of efforts and promote reuse, we need a way to describe **generic** container classes, which can be applied to describe containers of elements of many different types.

The notation

```
class C [G] ... The rest as for any other class declaration ...
```

introduces such a generic class. A name such as *G* appearing in brackets after the class name is known as a **formal generic parameter**; it represents an arbitrary type.

Within the class text, feature declarations can freely use *G* even though it is not known what type *G* stands for. Class *LIST* of ISE's EiffelBase libraries, for example, includes features

```
irst: G
  -- Value of first list element

extend (val: G)
  -- Add a new element of value val at end of list
  ...
```

The operations available on an entity such as *first* and *x*, whose type is a formal generic parameter, are the operations available on all types: use as source *y* of an assignment *x* := *y*, use as target *x* of such an assignment (although not for *val*, which as a formal routine argument is not variable), use in equality comparisons *x* = *y* or *x* /= *y*, and application of universal features from *ANY* such as *clone*, *equal* and *copy*.

To use a generic class such as list, a client will provide a type name as **actual generic parameter**; for example instead of using *DEPOSIT_LIST* the class *ACCOUNT* could include the declaration

```
all_deposits: LIST [DEPOSIT]
```

using *LIST* as a generic class and *DEPOSIT* as the actual generic parameter. Then all features declared in *LIST* as working on entities of type *G* will work, when called on the target *all_deposits*, on entities of type *DEPOSIT*. With the target

```
all_accounts: LIST [ACCOUNT]
```

these features would work on entities of type *ACCOUNT*.

A note of terminology: to avoid confusion, Eiffel literature uses the word **argument** for routine arguments, reserving **parameter** for the generic parameters of classes.

Genericity reconciles extendibility and reusability with the static type checking demanded by reliability. A typical error, such as confusing an account and a deposit, will be detected immediately at compile time, since the call *all_accounts.extend(dep)* is invalid for *dep* declared of type *DEPOSIT*. (What is valid is something like *all_accounts.extend(acc)* for *acc* of type *ACCOUNT*.) In other approaches, the same effect might require costly run-time checks (as in Java or Smalltalk), with the risk of run-time errors.

Further flexibility will be provided by providing a **constrained** form of genericity that allows assuming other operations, on a formal generic parameter, than just those of *ANY*. This will be seen in the discussion of inheritance.

An example of generic class from the Kernel Library is *ARRAY [G]*, which describes direct-access arrays. Features include:

- *put* to replace an element's value, as in *my_array.put (val, 25)* which replaces by *val* the value of the array entry at index 25.
- *item* to access an entry, as in *my_array.item (25)* yielding the entry at index 25. A synonym is---- FIX ---- "@", so that the same result can be obtained more tersely as *my_array @ 25*.
- *lower*, *upper* and *count*: queries yielding the bounds and the number of entries.
- The creation procedure *make*, as in **create** *my_array.make (1, 50)* which creates an array with the given index bounds. It is also possible to resize an array through *resize*, keeping of course the old elements. In general, the Eiffel method shuns built-in limits and favors automatically resizable structures.

The comment made about *INTEGER* and other basic classes applies to *ARRAY* too: Eiffel compilers know about this class, and will be able to process expressions of the form *my_array.put (val, 25)* and *my_array @ 25* in essentially the same way as a C or Fortran array access (*my_array [25]* in C). But it is consistent and practical to let developers treat *ARRAY* as a class and arrays as objects; many library classes in EiffelBase, for example, inherit from *ARRAY*. Once again the idea is to get the best of both worlds: the convenience and uniformity of the object-oriented way of thinking; and the efficiency of traditional approaches. A similar technique applies to another Kernel Library class, that one not generic: *STRING*, describing character strings with a rich set of string manipulation features.

The introduction of genericity brings up a small difference between classes and types. A generic class *C* is not directly a type since you cannot declare an entity as being of type *C* (you must use for some actual generic parameter *T* — itself a type). Rather, *C* is a type pattern. To obtain an actual type *C [T]*, you must provide an actual generic parameter *T*. This is known as a **generic derivation**. (*T* itself is, recursively, a type — either a non-generic class or again a generically derived type *D [U]* for some *D* and *U*, as in *LIST [ARRAY [INTEGER]]*.)

It remains true, however, that every type is based on a class. The base class of a generically derived type *C [T]* is *C*.

1.8 DESIGN BY CONTRACT, ASSERTIONS, EXCEPTIONS

Eiffel directly implements the ideas of Design by Contract, which enhance software reliability and provide a sound basis for software specification, documentation and testing, as well as exception handling and the proper use of inheritance.

Design by Contract basics

A system — a software system in particular, but the ideas are more general — is made of a number of cooperating components. Design by Contract states that their cooperation should be based on precise specifications — *contracts* — describing each party’s expectations and guarantees.

An Eiffel contract is similar to a real-life contract between two people or two companies, which it is convenient to express in the form of tables listing the expectations and guarantees. Here for example is how we could sketch the contract between a homeowner and the telephone company:

<i>provide_service</i>	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition: Pay bill	(From postcondition: Get telephone service
Supplier	(Satisfy postcondition: Provide telephone service	(From precondition: No need to provide anything if bill not paid

Note how the obligation for each of the parties maps onto a benefit for the other. This will be a general pattern.

The client’s obligation, which protects the supplier, is called a **precondition**. It states what the client must satisfy before requesting a certain service. The client’s benefit, which describes what the supplier must do (assuming the precondition was satisfied), is called a **postcondition**.

In addition to preconditions and postconditions, there will also be **invariants** applying to a class as a whole. More precisely a class invariant must be ensured by every creation procedure (or by the default initialization if there is no creation procedure), and maintained by every exported routine of the class.

Expressing assertions

Eiffel provides syntax for expressing preconditions (require), postconditions (ensure) and class invariants (invariant), as well as other assertion constructs studied later (see [“Instructions”, page 75](#)): loop invariants and variants, check instructions.

Here is a partial update of class *ACCOUNT* with more assertions:

```

note
  description: "Simple bank accounts"
class
  ACCOUNT
feature -- Access
  balance: INTEGER
    -- Current balance
  deposit_count: INTEGER
    -- Number of deposits made since opening
  do
    ... As before ...
  end
feature -- Element change
  deposit (sum: INTEGER)
    -- Add sum to account.
  require
    non_negative: sum >= 0
  do
    ... As before ...
  ensure
    one_more_deposit:
      deposit_count = old deposit_count + 1
    updated: balance = old balance + sum
  end
feature {NONE} -- Implementation
  all_deposits: DEPOSIT_LIST
    -- List of deposits since account's opening.
invariant
  consistent_balance: (all_deposits /= Void) implies
    (balance = all_deposits.total)
  zero_if_no_deposits: (all_deposits = Void) implies
    (balance = 0)
end

```

Each assertion is made of one or more subclauses, each of them a boolean expression (with the additional possibility of the **old** construct). If there is more than one subclause, as in the postcondition of *deposit* and in the invariant, they are treated as if they were connected by an **and**. Each clause may be preceded by a label, such as *consistent_balance* in the invariant, and a colon; the label is optional and does not affect the assertion's semantics, except for error reporting as explained in the next section, but including it systematically is part of the recommended style. The boolean expression *a implies b* is true if *a* is false, and otherwise if both *a* and *b* are true.

Because assertions benefit from the full power of boolean expressions, they may include function calls. This makes it possible to express sophisticated consistency conditions, such as “*the graph contains no cycle*”, which would not be otherwise expressible through simple expressions, or even through first-order predicate calculus, but which are easy to implement as Eiffel functions returning boolean results.

The precondition of a routine expresses conditions that the routine is imposing on its clients. Here a call to *deposit* is correct if and only if the value of the argument is non-negative. The routine does not guarantee anything for a call that does not satisfy the precondition. It is in fact part of the Eiffel method that a routine body should **never** test for the precondition, since it is the client’s responsibility to ensure it. (An apparent paradox of Design by Contract, which is reflected in the bottom-right entries of the preceding and following contract tables, and should not be a paradox any more at the end of this discussion, is that one can get *more* reliable software by having *fewer* explicit checks in the software text.)

The postcondition of a routine expresses what the routine does guarantee to its clients for calls satisfying the precondition. The notation **old expression**, valid in postconditions (**ensure** clauses) only, denotes the value that *expression* had on entry to the routine.

The precondition and postcondition state the terms of the contract between the routine and its clients, similar to the earlier example of a human contract:

<i>deposit</i>	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Use a non-negative argument.	(From postcondition:) Get deposits list and balance updated.
Supplier	(Satisfy postcondition:) Update deposits list and balance.	(From precondition:) No need to handle negative arguments.

The class invariant, as noted, applies to all features. It must be satisfied on exit by any creation procedures, and is implicitly added to both the precondition and postcondition of every exported routine. In this respect it is both good news and bad news for the routine implementer: good news because it guarantees that the object will initially be in a stable state, averting the need in the above example to check that the total of *all_deposits* is compatible with the *balance*; bad news because, in addition to its official contract as expressed by its specific postcondition, every routine must take care of restoring the invariant on exit.

A requirement on meaningful contracts is that they should be in good faith: satisfiable by an honest partner. This implies a consistency rule: if a routine is exported to a client (either generally or selectively), any feature appearing in its precondition must also be available to that client. Otherwise — for example if the precondition included **require** $n > 0$, where n is a secret attribute — the supplier would be making demands that a good-faith client cannot possibly check for.

It should be noted in this respect that ensuring a precondition does not necessarily mean testing for it explicitly. Assuming n is indeed exported, a client can make a correct call as

```
if  $x.n > 0$  then  $x.r$  end
```

possibly with an **else** part, but this is not the only possible form: if n is known to be positive, perhaps because some preceding call set it to the sum of two squares, then there is no need for protection by an **if** or equivalent. In such a case, a **check** instruction as introduced later (“[Instructions](#)”, [page 75](#)) is recommended if the reason for omitting the test is non-trivial.

Using assertions for built-in reliability

The first use of assertions is purely methodological. By applying a discipline of expressing, as precisely as possible, the logical assumptions behind software elements, one can write software whose reliability is built-in: software that is developed hand-in-hand with the rationale for its correctness.

This simple observation — usually not clear to people until they have practiced Design by Contract thoroughly on a large-scale project — brings as much change to software practices and quality as the rest of object technology.

Run-time assertion monitoring

Assertions in Eiffel are not just wishful thinking. They can be monitored at run time under the control of compilation options.

It should be clear from the preceding discussion that assertions are not a mechanism to test for special conditions, for example erroneous user input. For that purpose, the usual control structures (**if** $deposit_sum \geq 0$ **then** ...) are available, complemented in applicable cases by the exception handling mechanism reviewed next. An assertion is instead a **correctness condition** governing the relationship between two software modules (not a software module and a human, or a software module and an external device). If sum is negative on entry to *deposit*, violating the precondition, the culprit is some other software element, whose author was not careful enough to observe the terms of the deal. Bluntly:

Assertion Violation rule

A run-time assertion violation is the manifestation of a bug.

To be more precise:

- A precondition violation signals a bug in the client, which did not observe its part of the deal.
- A postcondition (or invariant) violation signals a bug in the supplier — the routine — which did not do its job.

That violations indicate bugs explains why it is possible to enable or disable assertion monitoring through mere compilation options: for a correct system — one without bugs —

assertions will always hold, so the compilation option makes no difference to the semantics of the system.

But of course for an incorrect system the best way to find out where the bug is — or just that there is a bug — is often to check the assertions. Hence the presence of the compilation options, which Eiffel environments typically provide at several levels (here as supported in ISE Eiffel, which makes them settable separately for each class, with defaults at the system and cluster levels):

- **no**: assertions have no run-time effect.
- **require**: check preconditions only, on routine entry.
- **ensure**: preconditions on entry, postconditions on exit.
- **invariant**: as **ensure**, plus class invariant on both entry and exit for qualified calls.
- **all**: as **invariant**, plus **check** instructions, loop invariants and loop variants ([“Instructions”, page 75](#)).

An assertion violation, if detected at run time under one of these options other than the first, will cause an exception ([“Exception handling”, page 42](#)). Unless the software has an explicit “retry” plan as explained below, the violation will cause production of an exception trace and termination (or, in development environment such as EiffelBench, a return to the browsing and debugging facilities of the environment at the point of failure). If present, the label of the violated subclause will be displayed, serving to identify the cause precisely.

The default is **require**. This is particularly interesting in connection with the Eiffel method’s insistence on using libraries: with libraries such as EiffelBase that are richly equipped with preconditions expressing terms of use, an error in the **client software** will often lead, for example through an incorrect argument, to violating one of these preconditions. A somewhat paradoxical consequence is that even an application developer who does not apply the method too well (out of carelessness, haste, indifference or ignorance) will still benefit from the presence of assertions in someone else’s library code.

During development and testing, assertion monitoring should be turned on at the highest possible level. Combined with static typing and the immediate feedback of compilation techniques such as the Melting Ice Technology, this permits the development process mentioned in the section [“Quality and functionality”, page 9](#), where errors are exterminated at birth. No one who has not practiced the method in a real project can imagine how many mistakes are found in this way; surprisingly often, a violation will turn out to affect an assertion that was just included for goodness’ sake, the developer being convinced that it could not “possibly” fail to be satisfied.

By providing a precise reference (the description of what the software is supposed to do) against which to assess the reality (what the software actually does), Design by Contract profoundly transforms the activities of debugging, testing and quality assurance.

When releasing the final version of a system, it is usually appropriate to turn off assertion monitoring, at least down to the **require** level. The exact policy depends on the circumstances; it is a tradeoff between efficiency considerations, the potential cost of mistakes, and how much the developers and quality assurance team trust the product. When developing the

software, however, one should always assume that monitoring will be turned off in the end (so as to avoid loosening one's guard).

The short form of a class

Another application of assertions regards documentation. Environment mechanisms — such as clicking the **short** button of a Class Tool in ISE's EiffelBench — will produce, from a class text, an abstracted version, the short form, which only includes the information relevant for client authors. Here is the short form of class *ACCOUNT* in its latest version:

```

note
  description: "Simple bank accounts"
class interface
  ACCOUNT
feature -- Access
  balance: INTEGER
    -- Current balance
  deposit_count: INTEGER
    -- Number of deposits made since opening
feature -- Element change
  deposit (sum: INTEGER)
    -- Add sum to account.
    require
      non_negative: sum >= 0
    ensure
      one_more_deposit: deposit_count = old deposit_count + 1
      updated: balance = old balance + sum
invariant
  consistent_balance: balance = all_deposits.total
end

```

The words **class interface** are used instead of just **class** to avoid any confusion with actual Eiffel text, since this is documentation, not executable software. (It is in fact possible to generate a compilable variant of the short form in the form of a deferred class, a notion defined later in this chapter.)

Compared to the full text, the short form keeps all the elements that are part of the abstract interface relevant to client authors:

- Names and signatures (argument and result type information) for the exported features.
- Header comments of these features, which carry informal descriptions of their purpose. (Hence the importance, mentioned in section [1.4](#), of always including such comments and writing them carefully.)

- Preconditions and postconditions of these routines (at least the subclauses involving only exported features, which may exclude certain postcondition subclauses).
- Class invariant (same observation).

The following elements are removed, however: any information about non-exported features; all the routine bodies (**do** clauses, or the **external** and **once** variants seen in “[External software](#)”, [page 15](#) above and “[Once routines, shared objects, smart initialization and on-demand execution](#)”, [page 73](#) below); assertion subclauses involving non-exported features; and some keywords not useful in the documentation, such as **is** for a routine.

In accordance with the Uniform Access principle ([page 18](#)), the short form does not distinguish between attributes and argument-less queries. In the above example, *balance* could be one or the other, as it makes no difference to clients, except possibly for performance.

The short form is the fundamental tool for using supplier classes in the Eiffel method. It protects client authors from the need to read the source code of software on which they rely. This is a crucial requirement in large-scale industrial developments.

The result is also particularly interesting because it satisfies the property that should always be required of good software documentation:

- It is truly abstract, free from the implementation details of what it describes but concentrating on its functionality.
- Instead of being produced separately — an unrealistic requirement, hard to impose on developers initially and becoming impossible in practice if we expect the documentation to remain up to date as the software evolves — the documentation is extracted from the software itself. It is not a separate product but a different view of the same product. This prolongs the **single product** principle that lies at the basis of Eiffel’s seamless development model ([1.3](#)).

Other views are possible. For example the EiffelCase tool of ISE’s environment offers graphical “bubble-and-arrow” diagrams representations of system structures, showing classes and their relations — client, inheritance — according to the conventions of BON (the Business Object Notation) with, in the first case, the possibility to edit these diagrams and generate updated Eiffel text in accordance with the principles of seamlessness and reversibility.

The short form — or its variant the flat-short form, which takes account of inheritance (“[Flat and flat-short forms](#)”, [page 64](#)) are the standard form of library documentation, used extensively, for example, in the book *Reusable Software* (see bibliography). Assertions play a central role in such documentation by expressing the terms of the contract. As demonstrated *a contrario* by the widely publicized \$500-million crash of the Ariane-5 rocket launcher in June of 1996, due to the incorrect reuse of a software module from the Ariane-4 project, **reuse without a contract documentation** is the path to disaster. Non-reuse would, in fact, be preferable.

Exception handling

Another application of Design by Contract governs the handling of unexpected cases. The vagueness of many discussions of this topic follows from the lack of a precise definition of terms such as “exception”. With Design by Contract we are in a position to be specific:

- Any routine has a contract to achieve.
- Its body defines a strategy to achieve it — a sequence (or other control structure) involving instructions. Some of these operations are themselves routines, with their own contracts; but even an atomic operation, such as the computation of an arithmetic operation, has an implicit contract, stating that the **result** will be representable.
- Any one of these operations may **fail**, that is to say be unable to meet its contract; for example an arithmetic operation may produce an overflow (non-representable result).
- Failure of one of these operations is an **exception** for the routine.
- As a result the routine may fail too — causing an exception in its own caller.

Note how the two basic concepts, failure and exception, are defined precisely. Although failure is the more basic concept — since it is defined for atomic, non-routine operations — the definitions are mutually recursive, since an exception may cause a failure of the recipient routine, and a routine’s failure causes an exception in its own caller.

Why only the observation that an exception “may” cause a failure? The reason is that a routine may have planned for the exception and defined a **rescue** policy. This is done through a clause with the corresponding keyword, as in:

```

read_next_character (f: FILE)
    -- Make next character available in last_character;
    -- if impossible, set failed to True.
require
    readable: file.readable
local
    impossible: BOOLEAN
do
    if impossible then
        failed := True
    else
        last_character := low_level_read_function (f)
    end
rescue
    impossible := True
retry
end

```

This example includes the only two constructs needed for exception handling: **rescue** and **retry**. The **retry** instruction is only permitted in a rescue clause; its effect is to start again the

execution of the routine, without repeating the initialization of local variables (such as *impossible* in the example, which was initialized to *False* on first entry). Features *failed* and *last_character* are assumed to be attributes of the enclosing class.

This example is typical of the use of exceptions: as a last resort, for situations that should not occur. The routine has a precondition, *file.readable*, which ascertains that the file exists and is accessible for reading characters. So clients should check that everything is fine before calling the routine. Although this check is almost always a guarantee of success, a rare combination of circumstances could cause a change of file status (because a user or some other system is manipulating the file) between the check for *readable* and the call to *low_level_read_function*. If we assume this latter function will fail if the file is not readable, we must catch the exception.

A variant would be

```
local
  attempts: INTEGER
do
  if attempts < Max_attempts then
    last_character := low_level_read_function (f)
  else
    failed := True
  end
rescue
  attempts := attempts + 1
  retry
end
```

which would try again up to *Max_attempts* times before giving up.

The above routine, in either variant, never fails: it always fulfills its contract, which states that it should either read a character or set *failed* to record its inability to do so. In contrast, consider the variant

```
local
  attempts: INTEGER
do
  last_character := low_level_read_function (f)
rescue
  attempts := attempts + 1
  if attempts < Max_attempts then
    retry
  end
end
```

with no more role for *failed*. In this case, after *Max_attempts* unsuccessful attempts, the routine will execute its **rescue** clause to the end, with no **retry** (the **if** having no **else** clause). This is how a routine fails. It will, as noted, pass on the exception to its caller.

Such a rescue clause should, before returning, restore the invariant of the class so that the caller and possible subsequent **retry** attempts from higher up find the objects in a consistent state. As a result, the rule for an absent **rescue** clause (the case, of course, for the vast majority of routines in most systems) is that it is equivalent to

```
rescue
  default_rescue
```

where procedure *default_rescue* comes from *ANY*, where it is defined to do nothing; in a system built for robustness, classes subject to non-explicitly-**rescued** exceptions should redefine *default_rescue* (perhaps using a creation procedure, which is bound by the same formal requirement) so that it will always restore the invariant.

Behind Eiffel's exception handling scheme lies the principle — at first an apparent platitude, but violated by many existing mechanisms — that a routine should **either succeed or fail**. This is all a result of the contract notion: succeeding means being able to fulfil the contract (possibly after one or more **retry**); failure is the other case, which must always trigger an exception in the caller. Without this principle it would be possible for a routine to miss its contract and yet to return to its caller in a seemingly normal state. That is the worst possible way to handle an exception.

Concretely, exceptions result from the following events:

- A routine failure (**rescue** clause executed to the end with no **retry**), as just seen.
- Assertion violation, if they are monitored.
- Attempt to call a feature on a void reference: *x.f* (...), the fundamental computational mechanism, can only work if *x* is attached to an object, and will cause an exception otherwise.
- Developer exception, as seen next.
- Operating system signal: no memory available for a requested creation or clone (even after garbage collection has rummaged everything to find some space), arithmetic overflow. (But no C/C++-like “wrong pointer address”, which cannot occur thanks to the statically typed nature of Eiffel.)

It may in some cases be useful, when handling exceptions in **rescue** clauses, to ascertain the exact nature of the exception that got the execution there. For this it suffices to inherit from the Kernel Library class *EXCEPTIONS*, which provides queries such as *exception*, giving the code for the last exception, and symbolic names (“[Constant attributes](#)”, [page 75](#)) for all such codes, such as *No_more_memory* and so on. Then by testing *exception* against various possibilities one can have specific exception handling. The method strongly suggests, however, that exception handling code should remain simple; a complicated algorithm in a **rescue** clause is probably a sign of abuse.

Class *EXCEPTIONS* also provides various facilities for fine-tuning the exception facilities, such as a procedure *raise* that will explicitly trigger a “developer exception” with a code than can then be detected and processed.

Exception handling makes it possible to produce Eiffel software that is not just correct but robust, by planning for cases that should *not* normally arise, but might out of Murphy’s law, and ensuring they do not affect the software’s basic safety and simplicity.

Other applications of Design by Contract

The Design by Contract ideas pervade the Eiffel method. In addition to the applications just mentioned, they have two particularly important consequences:

- They make it possible to use Eiffel for analysis and design. At a high level of abstraction, it is necessary to be precise too. With the exception of BON, object-oriented analysis and design methods tend to favor abstraction over precision. Thanks to assertions, it is possible to express precise properties of a system (“At what speed should the alarm start sounding?”) without making any commitment to implementation. The discussion of deferred classes (“[Applications of deferred classes](#)”, [page 54](#)) will show how to write a purely descriptive, non-software model in Eiffel, using assertions to describe the essential properties of a system without any computer or software aspect.
- Assertions also serve to control the power of inheritance-related mechanisms — redeclaration, polymorphism, dynamic binding — and channel them to correct uses by assigning the proper semantic limits. See “[Inheritance and contracts](#)”, [page 59](#).

1.9 THE INHERITANCE MECHANISM

Inheritance is a powerful and attractive technique. A look at either the practice or literature shows, however, that it is not always well applied. Eiffel has made a particular effort to tame inheritance for the benefit of modelers and software developers. Many of the techniques are original with Eiffel. Paul Dubois has written (*comp.lang.python* Usenet newsgroup, 23 March 1997): *there are two things that [Eiffel] got right that nobody else got right anywhere else: support for design by contract, and multiple inheritance. Everyone should understand these “correct answers” if only to understand how to work around the limitations in other languages.*

Basic inheritance structure

To make a class inherit from another, simply use an **inherit** clause:

```
note ... class D creation ... inherit
  A
  B
  ...
feature
  ...
```

This makes *D* an heir of *A*, *B* and any other class listed. Eiffel supports **multiple inheritance**: a class can have as many parents as it needs. Later sections ([“Multiple inheritance and feature renaming”, page 57](#) and [“Repeated inheritance and selection”, page 65](#)) will explain how to handle the possible conflicts created by parent features.

By default *D* will simply include all the original features of *A*, *B*, ..., to which it may add its own through its **feature** clauses if any. But the inheritance mechanism is more flexible, allowing *D* to adapt the inherited features in many ways. Each parent name — *A*, *B*, ... in the example — can be followed by a Feature Adaptation clause, with subclauses, all optional, introduced by keywords **rename**, **export**, **undefine**, **redefine** and **select**, enabling the author *A* to make the best use of the inheritance mechanism by tuning the inherited features to the precise needs of *D*. This makes inheritance a principal tool in the Eiffel process, mentioned earlier, of carefully crafting each individual class for the benefit of its clients. The various Feature Adaptation subclauses will be reviewed in the following sections.

Redefinition

The first form of feature adaptation is the ability to change the implementation of an inherited feature. Assume a class *SAVINGS_ACCOUNT* that specializes the notion of account. It is probably appropriate to define it as an heir to class *ACCOUNT*, to benefit from all the features of *ACCOUNT* still applicable to savings accounts, and express the conceptual relationship that every savings account “is” an account (apart from its own specific properties). But we may need to produce a different effect for procedure *deposit* so that besides recording the deposit and update the balance it also updates the interest (say).

This example is typical of the form of reuse promoted by inheritance and crucial to effective reusability in software: the case of *reuse with adaptation*. Traditional forms of reuse are all-or-nothing: either you take a component exactly as it is, or you build your own. Inheritance will get us out of this “reuse or redo” dilemma by allowing us to reuse *and* redo. The mechanism is feature redefinition:

```

note
    description: "Savings accounts"
class
    SAVINGS_ACCOUNT
inherit
    ACCOUNT
        redefine deposit end
feature -- Element change
    deposit (sum: INTEGER)
        -- Add sum to account.
        do
            ... New implementation (see below) ...
        end
    ... Other features ...
end

```

Without the **redefine** subclause, the declaration of *deposit* would be invalid, yielding two features of the same name, the inherited one and the new one. The subclause makes this valid by specifying that the new declaration will override the old one.

In a redefinition, the original version — such as the *ACCOUNT* implementation of *deposit* in this example — is called the **precursor** of the new version. It is common for a redefinition to rely on the precursor's algorithm and add some other actions; the reserved word *Precursor* helps achieve this goal simply. Permitted only in a routine redefinition, it denotes the parent routine being redefined. So here the body of the new *deposit* could be of the form

```
Precursor (sum)    -- Apply the algorithm of ACCOUNT's version of deposit
... Instructions to update the interest ...
```

Besides changing the implementation of a routine, a redefinition can turn an argument-less function into an attribute; for example a proper descendant of *ACCOUNT* could redefine *deposits_count*, originally a function, as an attribute. The principle of Uniform Access (page 18) guarantees that the redefinition makes no change for clients, which will continue to use the feature under the form *acc.deposits_count*.

Polymorphism

The inheritance mechanism is relevant to both roles of classes: module and type. Its application as a mechanism to reuse, adapt and extend features from one class to another, as just seen, covers the module role. But inheritance is also a **subtyping** mechanism. To say that *D* is an heir of *A*, or more generally a descendant of *A*, is to express that instances of *D* can be viewed as instances of *A*.

The mechanism that supports this idea is **polymorphic assignment**. In an assignment $x := y$, the types of *y* do not, thanks to inheritance, have to be identical; the rule is that the type of *y* must simply **conform** to another. A class *D* conforms to a class *A* if and only if it is a descendant (which of course includes the case in which *A* and *D* are the same class); if these classes are generic, conformance of *D* [*U*] to *C* [*T*] requires in addition that type *U* conform to type *T* (through the recursive application of the same rules).

So with the inheritance relations suggested earlier, the declarations

```
acc: ACCOUNT; sav: SAVINGS_ACCOUNT
```

make it valid to write the assignment

```
acc := sav
```

which will assign to *acc* a reference attached (if not void) to a direct instance of type *SAVINGS_ACCOUNT*, not *ACCOUNT*.

Such an assignment, where the source and target types are different, is said to be polymorphic. An entity such as *acc*, which as a result of such assignments may become

attached at run time to objects of types other than the one declared for it, is itself called a polymorphic entity.

For polymorphism to respect the reliability requirements of Eiffel, it must be controlled by the type system and enable static type checking. We certainly do not want an entity of type *ACCOUNT* to become attached to an object of type *DEPOSIT*. Hence the second typing rule:

Type Conformance rule

An assignment $x := y$, or the use of y as actual argument corresponding to the formal argument x in a routine call, is only valid if the type of y conforms to the type of x .

The second case is that of a call such as *target.routine* (... , y , ...) where the corresponding routine declaration is of the form routine (... , x : *SOME_TYPE*, ...). The rules governing the setting of x to the value of y at the beginning of the call are exactly the same as those of an assignment $x := y$: not just the type rule, as expressed by Type Conformance (the type of y must conform to *SOME_TYPE*), but also the actual run-time effect which, as for assignments, will be either a reference attachment or, for expanded types, a copy.

The ability to accept the assignment $x := \text{Void}$ for x of any reference type (“[Basic operations](#)”, page 26) is a consequence of the Type Conformance rule, since Void is of type NONE which by construction (“[The global inheritance structure](#)”, page 14) conforms to all types.

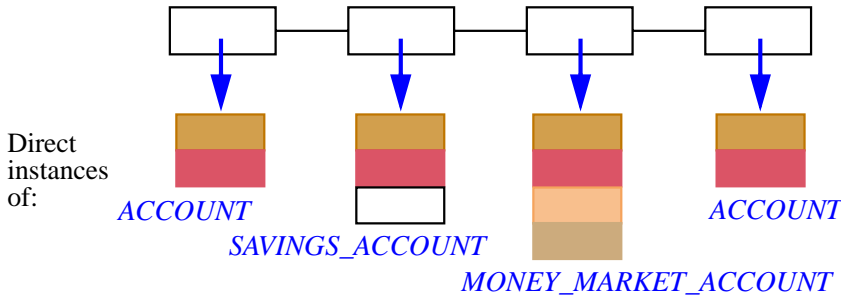
Polymorphism also yields a more precise definition of “instance”. A **direct instance** of a type A is an object created from the exact pattern defined by the declaration of A 's base class, with one field for each of the class attributes; you will obtain it through a creation instruction of the form **create** x ..., for x of type A , or by cloning an existing direct instance. An **instance** of A is a direct instance of any type conforming to A : A itself, but also many types based on descendant classes. So an instance of *SAVINGS_ACCOUNT* is also an instance, although not a direct instance, of *ACCOUNT*.

A consequence of polymorphism is the ability to define **polymorphic data structures**. With a declaration such as

accounts: LIST [*ACCOUNT*]

the procedure call *accounts.extend* (*acc*), because it uses a procedure *extend* which in this case expects an argument of any type conforming to *ACCOUNT*, will be valid not only if *acc* is of type *ACCOUNT* but also if it is of a descendant type such as *SAVINGS_ACCOUNT*.

Successive calls of this kind make it possible to construct a data structure that, at run-time, might contain objects of several types, all conforming to *ACCOUNT*:



*Polymorphic
data structure*

Such polymorphic data structures combine the flexibility and safety of genericity and inheritance. They can be more or less general depending on the type, here *ACCOUNT*, chosen as actual generic parameter; static typing is again precious, prohibiting for example a mistaken insertion of the form *accounts.extend(dep)* where *dep* is of type *DEPOSIT*, which does not conform to *ACCOUNT*.

It remains possible to produce unrestrictedly polymorphic data structures, such as a *general_list*: *LIST [ANY]* which makes the call *general_list.extend(x)* valid for any *x*. The price to pay is that the operations applicable to an element retrieved from such a list are only the most general ones (assignment, clone, equality comparison and the like) — although assignment attempt, studied below, will make it possible to apply more specific operations after checking that a retrieved object is of the appropriate type.

Dynamic binding

The complement of polymorphism and dynamic binding, which is the answer to the question “What version of a feature will be applied in a call whose target is polymorphic?”. For example, if *acc* is of type *ACCOUNT*, the attached objects may now, thanks to polymorphism, be direct instances not just of *ACCOUNT* but also *SAVINGS_ACCOUNT* or other descendants. Some of these descendants, indeed *SAVINGS_ACCOUNT* among them, redefine features such as *deposit*. Then we have to ask what the effect will be for a call of the form

acc.deposit(some_value)?

Dynamic binding is the clearly correct answer: the call will execute the version of *deposit* from the generating class of the object attached to *acc* at run time. If *acc* is attached to a direct instance of *ACCOUNT*, execution will use the original *ACCOUNT* version; if *acc* is attached to a direct instance of *SAVINGS_ACCOUNT*, the call will execute the version redefined in that class.

This is a clear correctness requirement. A policy of *static binding* (as available for example by default in C++ or Borland’s Delphi) would take the declaration of *acc* as an *ACCOUNT* literally. But that declaration is only meant to ensure generality, to enable the use

of a single name *acc* in many different cases: what counts at execution time is the object that *acc* represents. Applying the *ACCOUNT* version to a *SAVINGS_ACCOUNT* object would be wrong, possibly leading in particular to objects that violate the invariant of their own generating class (since there is no reason a routine of *ACCOUNT* will preserve the specific invariant of a proper descendant such as *SAVINGS_ACCOUNT*, which it does not even know about).

Note that in some cases the choice between static and dynamic binding does not matter: this is the case for example if a call's target is not polymorphic, or if the feature of the call is redefined nowhere in the system. In such cases the use of static binding permits slightly faster calls (since the feature is known at compile time). This application of static binding should, however, be treated as a **compiler optimization**. Good Eiffel compilers detect such cases and process them accordingly — unlike approaches that make developers responsible for specifying what should be static and what dynamic (a tedious and error-prone task, especially delicate because a minute change in the software can make a static call, far away in another module of a large system, suddenly become dynamic). Eiffel developers are protected from such concerns; they can rely on the semantics of dynamic binding in all cases, with the knowledge that an optimizing compiler will apply static binding when safe and desirable.

Even in cases that require dynamic binding, the design of Eiffel, in particular the typing rules, enable compilers to make the penalty over the static-binding calls of traditional approaches very small and, most importantly, **constant-bounded**: it does not grow with the depth or complexity of the inheritance structure. The discovery in 1985 of a technique for constant-time dynamic binding calls, even in the presence of multiple and repeated inheritance, was the event that gave the green light to the development of Eiffel.

Dynamic binding is particularly interesting for polymorphic data structures. If we iterate over the list of accounts of various kinds, *accounts: LIST [ACCOUNT]*, illustrated in the last figure, and at each step let *acc* represent the current list element, we can repeatedly apply

acc.deposit (...)

to have the appropriate variant of the *deposit* operation triggered for each element.

The benefit of such techniques appears clearly if we compare them with the traditional way to address such needs: using multi-branch discriminating instructions of the form **if** “*Account is a savings account*” **then** ... **elseif** “*It is a money market account*” **then** ... and so on, or the corresponding **case** ... **of** ... or **inspect** instructions. Apart from their heaviness and complexity, such solutions cause many components of a software system to rely on the knowledge of the exact set of variants available for a certain notion, such as bank account. Then any addition, change or removal of variants can cause a ripple of changes throughout the architecture. This is one of the major obstacles to extendibility and reusability in traditional approaches. In contrast, using the combination of inheritance, redefinition, polymorphism and dynamic binding makes it possible to have a **point of single choice** — a unique location in the system which knows the exhaustive list of variants. Every client then manipulates entities of the most general type, *ACCOUNT*, through dynamically bound calls of the form *acc.some_account_feature (...)*.

These observations make dynamic binding appear for what it is: not an implementation mechanism, but an **architectural technique** that plays a key role (along with information hiding, which it extends, and Design by Contract, to which it is linked through the assertion redefinition rules seen below) in providing the modular system architectures of Eiffel, the basis for the method's approach to reusability and extendibility. These properties apply as early as analysis and modeling, and continue to be useful throughout the subsequent steps.

Deferred features and classes

In the above examples of dynamic binding, all classes were assumed to be fully implemented, and dynamically bound features had a version in every relevant class, including the most general ones such as *ACCOUNT*.

It is also useful to define classes that leave the implementation of some of their features entirely to proper descendants. Such an abstract class is known as **deferred**; so are its unimplemented features. The reverse of deferred is **effective**, meaning fully implemented.

LIST is a typical example of deferred class. As it describes the general notion of list, it should not favor any particular implementation; that will be the task of its effective descendants, such as *LINKED_LIST* (linked implementation), *TWO_WAY_LIST* (linked both ways), *ARRAYED_LIST* (implementation by an array), all effective, and all indeed to be found in ISE's EiffelBase libraries.

At the level of the deferred class *LIST*, some features such as *extend* (add an item at the end of the list) will have no implementation and hence will be declared as deferred. Here is the corresponding form, illustrating the syntax for both deferred classes and their deferred features:

```

note
  description: "Sequential finite lists, without a commitment%[
    to a representation%]"
deferred class
  LIST [G]
feature -- Access
  count: INTEGER
    -- Number of items in list
  do
    ... See below; this feature can be effective ...
  end

```

```

feature -- Element change
  extend (x: G)
    -- Add x at end of list.
    require
      space_available: not full
    deferred
    ensure
      one_more: count = old count + 1
    end

... Other feature declarations and invariant ...
end

```

A deferred feature (considered to be a routine, although it can yield an attribute in a proper descendant) has the single keyword **deferred** in lieu of the **do** *Instructions* clause of an effective routine. A deferred class — defined as a class that has at least one deferred feature — must be introduced by **deferred class** instead of just **class**.

As the example of *extend* shows, a deferred feature, although it has no implementation, can be equipped with assertions. They will be binding on implementations in descendants, in a way to be explained below.

Deferred classes do not have to be fully deferred. They can contain some effective features along with their deferred ones. Here, for example, we may express *count* as a function:

```

count: INTEGER
  -- Number of items in list
  do
    from start until after loop
      Result := Result + 1; forth
    end
  end

```

This implementation relies on the loop construct described below (**from** introduces the loop initialization) and on a set of deferred features of the class which allow traversal of a list based on moving a fictitious cursor: *start* to bring the cursor to the first element if any, *after* to

find out whether all relevant elements have been seen, and forth (with precondition **not** after) to advance the cursor to the next element. For example, forth appears as

```

forth
    -- Advance cursor by one position
require
    not_after: not after
deferred
ensure
    moved_right: index = old index + 1
end

```

where *index* — another deferred feature of the class — is the integer position of the cursor.

Although the above version of feature count is time-consuming — it implies a whole traversal just for the purpose of determining the number of elements — it has the advantage of being applicable to all variants, without any commitment to a choice of implementation, as would follow for example if we decided to treat count as an attribute. Proper descendants can always redefine count for more efficiency.

Function count illustrates one of the most important contributions of the method to reusability: the ability to define **behavior classes** that capture common behaviors (such as count) while leaving the details of the behaviors (such as start, after, forth) open to many variants. As noted earlier, traditional approaches to reusability provide closed reusable components. A component such as *LIST*, although equipped with directly usable behaviors such as count, is open to many variations, to be provided by proper descendants.

A class *B* inheriting from a deferred class *A* may provide implementations — effective declarations — for the features inherited in deferred form. In this case there is no need for the equivalent of a **redefine** subclause; the effective versions simply replace the inherited versions. The class is said to **effect** the corresponding features. If after this process there remain any deferred features, *B* is still considered deferred, even if it introduces no deferred features of its own, and must be declared as **deferred class**.

In the example, classes such as *LINKED_LIST* and *ARRAYED_LIST* will effect all the deferred features they inherit from *LIST* — *extend*, *start* etc. — and hence be effective.

Note that — except in some applications restricted to pure system modeling — deferred classes and features only make sense thanks to polymorphism and dynamic binding. Because *extend* has no implementation in class *LIST*, a call of the form *my_list.extend* (°) with *my_list* of type *LIST [T]* for some *T* can only be executed if *my_list* is attached to a direct instance of an effective proper descendant of *LIST*, such as *LINKED_LIST*; then it will use the corresponding version of *extend*. Static binding would not even be meaningful here.

Even an effective feature of *LIST* such as count may depend on deferred features (start and so on), so that a call of the form *my_list.count* can only be executed in the context of an effective descendant.

All this indicates that a deferred class must have **no direct instance** (it will have instances, the direct instances of its effective descendants). If it had any, we could call deferred features on them, leading to execution-time impossibility. The rule which achieves this goal is simple: if the base type of x is a deferred class, no creation instruction of target x , of the form **create** $x \dots$, is permitted.

Applications of deferred classes

Deferred classes cover abstract notions with many possible variants. They are widely used in Eiffel where they cover various needs:

- Capturing high-level classes, with common behaviors.
- Defining the higher levels of a general taxonomy, especially in the inheritance structure of a library.
- Defining the components of an architecture during system design, without commitment to a final implementation.
- Describing domain-specific concepts in analysis and modeling.

As the reader will have noted, these applications make deferred classes a central tool of the Eiffel method's support for seamlessness and reversibility. The last one in particular uses deferred classes and features to model objects from an application domain, without any commitment to implementation, design, or even software (and computers). Deferred classes are the ideal tool here: they express the properties of the domain's abstractions, without any temptation of implementation bias, yet with the precision afforded by type declarations, inheritance structures (to record classifications of the domain concepts), and assertions to express the abstract properties of the objects being described.

Unlike approaches using a separate object-oriented analysis and design method and notation (Booch, OMT, UML...), this technique integrates seamlessly with the subsequent phases (assuming the decision is indeed taken to develop a software system): it suffices to develop the deferred classes progressively by introducing effective elements, either by modifying the classes themselves or by introducing design- and implementation-oriented descendants. In the resulting system, the classes that played an important role for analysis, and are the most meaningful for customers, will remain important; as we have seen ("[Seamlessness and reversibility](#)", [page 8](#)) this *direct mapping* property is a great help for extendibility.

The following sketch (from the book [Object-Oriented Software Construction](#)) illustrates these ideas on the example of scheduling the programs of a TV station. This is pure modeling of an application domain; no computers or software are involved yet. The class describes the notion of program segment.

Note the use of assertions to define semantic properties of the class, its instances and its features. Although often presented as high-level, most object-oriented analysis methods (with the exception of Waldén's and Nerson's Business Object Notation) have no support for the

expression of such properties, limiting themselves instead to the description of broad structural relationships.

note

description: "Individual fragments of a broadcasting schedule"

deferred class

SEGMENT

feature -- Access

schedule: SCHEDULE is deferred end

-- Schedule to which segment belongs

index: INTEGER is deferred end

-- Position of segment in its schedule

starting_time, ending_time: INTEGER is deferred end

-- Beginning and end of scheduled air time

next: SEGMENT is deferred end

-- Segment to be played next, if any

sponsor: COMPANY is deferred end

-- Segment's principal sponsor

rating: INTEGER is deferred end

-- Segment's rating (for children's viewing etc.)

Minimum_duration: INTEGER is 30

-- Minimum length of segments, in seconds

Maximum_interval: INTEGER is 2

-- Maximum time between two successive segments, in seconds

feature -- Element change

set_sponsor (s: SPONSOR)

require

not_void: s /= Void

deferred

ensure

sponsor_set: sponsor = s

end

... change_next, set_rating omitted ...

invariant

in_list: $(1 \leq \text{index})$ **and** $(\text{index} \leq \text{schedule.segments.count})$

in_schedule: $\text{schedule.segments.item}(\text{index}) = \text{Current}$

next_in_list: $(\text{next} \neq \text{Void})$ **implies** $(\text{schedule.segments.item}(\text{index} + 1) = \text{next})$

no_next_iff_last: $(\text{next} = \text{Void}) = (\text{index} = \text{schedule.segments.count})$

non_negative_rating: $\text{rating} \geq 0$

positive_times: $(\text{starting_time} > 0)$ **and** $(\text{ending_time} > 0)$

sufficient_duration: $\text{ending_time} - \text{starting_time} \geq \text{Minimum_duration}$

decent_interval: $(\text{next.starting_time}) - \text{ending_time} \leq \text{Maximum_interval}$

end

Structural property classes

An interesting category of deferred classes includes classes whose purpose is to describe a structural property, which may be useful to the description of many other classes. Typical examples are covered by classes of the Kernel Library:

- *NUMERIC* describes objects on which arithmetic operations $+$, $-$, $*$, $/$ are available, with the properties of a ring (associativity, distributivity, zero elements etc.). Kernel Library classes such as *INTEGER* and *REAL* — but not, for example, *STRING* — are descendants of *NUMERIC*. An application that defines a class *MATRIX* may also make it a descendant of *NUMERIC*.
- *COMPARABLE* describes objects on which comparison operations $<$, \leq , $>$, \geq are available, with the properties of a total preorder (transitivity, irreflexivity). Kernel Library classes such as *CHARACTER*, *STRING* and *INTEGER* — but not out *MATRIX* example — are descendants of *NUMERIC*.

For such classes it is again essential to permit the inclusion of effective features in a deferred class, and to include assertions. For example class *COMPARABLE* will declare *plus alias* " $<$ " as deferred, and express the other features effectively in terms of it. (The type *like* *Current* is explained in "[Covariance and anchored declarations](#)", page 70 below; it may be considered equivalent, in the following class text, to the type *COMPARABLE*.)

note

description: "Objects that can be compared according to a total preorder relation"

deferred class

COMPARABLE

```

feature -- Comparison
  is_less alias "<" (other: like Current): BOOLEAN
    -- Is current object less than other?
  require
    other_exists: other /= Void
  deferred
  ensure
    asymmetric: Result implies not (other < Current)
  end

  is_less_equal alias "<=" (other: like Current): BOOLEAN
    -- Is current object less than or equal to other?
  require
    other_exists: other /= Void
  do
    Result := (Current < other) or is_equal (other)
  ensure
    definition: Result = (Current < other) or is_equal (other)
  end
... Other features: is_greater alias ">", min, max, ...
invariant
  irreflexive: not (Current < Current)
end

```

Note how \leq is defined in terms of $<$, and \geq in terms of \leq .

Multiple inheritance and feature renaming

It is often necessary to define a new class in terms of several existing ones. For example:

- The Kernel Library classes *INTEGER* and *REAL* must inherit from both *NUMERIC* and *COMPARABLE*.
- A class *TENNIS_PLAYER*, in a system for keeping track of player ranking, will inherit from *COMPARABLE*, as well as from other domain-specific classes.
- A class *COMPANY_PLANE* may inherit from both *PLANE* and *ASSET*.
- Class *ARRAYED_LIST*, describing an implementation of lists through arrays, may inherit from both *LIST* and *ARRAY*.

In all such cases multiple inheritance provides the answer.

Multiple inheritance can cause **name clashes**: two parents can include a feature with the same name. This would conflict with the ban on name overloading within a class — the rule that no two features of a class may have the same name. Eiffel provides a simple way to remove the name clash at the point of inheritance through the **rename** subclause, as in

```

note
    description: "Sequential finite lists implemented as arrays"
class
    ARRAYED_LIST [G]
inherit
    LIST [G]
    ARRAY [G]
    rename
        count as capacity, item as array_item
    end
feature
    ...
end

```

Here both *LIST* and *ARRAY* have features called *count* and *item*. To make the new class valid, we give new names to the features inherited from *ARRAY*, which will be known within *ARRAYED_LIST* as *capacity* and *array_item*. Of course we could have renamed the *LIST* versions instead, or renamed along both inheritance branches.

Every feature of a class has a **final name**: for a feature introduced in the class itself (“immediate” feature) it is the name appearing in the declaration; for an inherited feature that is not renamed, it is the feature’s name in the parent; for a renamed feature, it is the name resulting from the renaming. This definition yields a precise statement of the rule against in-class overloading:

Final Name rule

Two different features of a class may not have the same final name.

It is interesting to compare renaming and redefinition. The important distinction is between features and feature names. Renaming keeps a feature, but changes its name. Redefinition keeps the name, but changes the feature. In some cases, it is of course appropriate to do both.

Renaming is interesting even in the absence of name clashes. A class may inherit from a parent a feature which it finds useful for its purposes, but whose name, appropriate for the context of the parent, is not consistent with the context of the heir. This is the case with *ARRAY*’s feature *count* in the last example: the feature that defines the number of items in an array — the total number of available entries — becomes, for an arrayed list, the *maximum* number of list items; the truly interesting indication of the number of items is the count of how many items have been inserted in the list, as given by feature *count* from *LIST*. But even if we did not have a name clash because of the two inherited *count* features we should rename *ARRAY*’s *count* as *capacity* to maintain the consistency of the local feature terminology.

The **rename** subclause appears before all the other feature adaptation subclauses — **redefine** already seen, and the remaining ones **export**, **undefine** and **select** — since an

inherited feature that has been renamed sheds its earlier identity once and for all: within the class, and to its own clients and descendants, it will be known solely through the new name. The original name has simply disappeared from the name space. This is essential to the view of classes presented earlier: self-contained, consistent abstractions prepared carefully for the greatest enjoyment of clients and descendants.

Inheritance and contracts

A proper understanding of inheritance requires looking at the mechanism in the framework of Design by Contract, where it will appear as a form of *subcontracting*.

The first rule is that invariants accumulate down an inheritance structure:

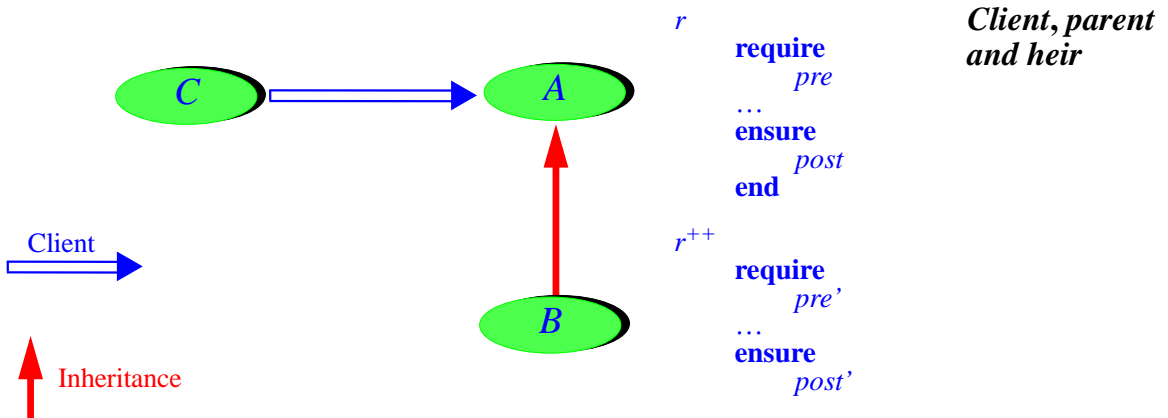
Invariant Accumulation rule

The invariants of all the parents of a class apply to the class itself.

The invariant of a class is automatically considered to include — in the sense of logical “and” — the invariants of all its parents. This is a consequence of the view of inheritance as an “is” relation: if we may consider every instance of B as an instance of A , then every consistency constraint on instances of A must also apply to instances of B .

Next we consider routine preconditions and postconditions. The rule here will follow from an examination of what contracts mean in the presence of polymorphism and dynamic binding.

Consider a parent A and a proper descendant B (a direct heir on the following figure), which redefines a routine r inherited from A .



As a result of dynamic binding, a call $al.r$ from a client C may be serviced not by A 's version of r but by B 's version if al , although declared of type A , becomes at run time attached to an instance of B . This shows the combination of inheritance, redefinition, polymorphism and dynamic binding as providing a form of **subcontracting**; A subcontracts certain calls to B .

The problem is to keep subcontractors honest. Assuming preconditions and postconditions as shown on the last, a call in C of the form

if $al.pre$ then $al.r$ end

or just $al.q; al.r$ where the postcondition of q implies the precondition pre of r , satisfies the terms of the contract and hence is entitled to being handled correctly — to terminate in a state satisfying $al.post$. But if we let the subcontractor B redefine the assertions to arbitrary pre' and $post'$, this is not necessarily the case: pre' could be stronger than pre , enabling B not to process correctly certain calls that are correct from A 's perspective; and $post'$ could be weaker than $post$, enabling B to do less of a job than advertized for r in the short form of A , the only official reference for authors of client classes such as C . (An assertion p is stronger than or equal to an assertion q if p implies q in the sense of boolean implication.)

The rule, then, is that for the redefinition to be correct the new precondition pre' must be weaker than or equal to the original pre , and the new postcondition $post'$ must be stronger than or equal to the original $post$.

Because it is impossible to check simply that an assertion is weaker or stronger than another, the language rule relies on new variants of the assertion constructs: **require else** and **ensure then**, relying on the mathematical property that, for any assertions p and q , p implies (p or q), and (p and q) implies p . For a precondition, using **require else** with a new assertion will perform an **or**, which can only weaken the original; for a postcondition, **ensure then** will perform an **and**, which can only strengthen the original. Hence the rule:

Assertion Redeclaration rule

In the redeclared version of a routine, it is not permitted to use a **require** or **ensure** clause. Instead you may:

- Use a clause introduced by **require else**, to be or-ed with the original precondition.
- Use a clause introduced by **ensure then**, to be and-ed with the original postcondition.

In the absence of such a clause, the original assertion is retained.

The last case — retaining the original — is frequent but by no means universal.

The Assertion Redeclaration rule applies to **redeclarations**. This terms covers not just redefinition but also effecting (the implementation, by a class, of a feature that it inherits deferred). The rules — not just for assertions but also, as reviewed below, for typing — are indeed the same in both cases. Without the Assertion Redeclaration rule, assertions on deferred features, such as those on *extend*, *count* and *forth* in [“Deferred features and classes”, page 51](#), would be almost useless — wishful thinking; the rule makes them binding on all effectings in descendants.

From the Assertion Redeclaration rule follows an interesting technique: abstract preconditions. What needs to be weakened for a precondition (or strengthened for a

postcondition) is not the assertion's concrete semantics but its abstract specification as seen by the client. A descendant can change the *implementation* of that specification as it pleases, even to the effect of strengthening the concrete precondition, as long as the abstract form is kept or weakened. The precondition of procedure *extend* in the deferred class *LIST* provided an example. We wrote the routine (page 52) as

```

extend (x: G)
    -- Add x at end of list.
    require
        space_available: not full
    deferred
    ensure
        one_more: count = old count + 1
    end

```

The precondition expresses that it is only possible to add an item to a list if the representation is not full. We may well consider — in line with the Eiffel principle that whenever possible structures should be of unbounded capacity — that *LIST* should by default make *full* always return false:

```

full: BOOLEAN
    -- Is representation full?
    -- (Default: no)
    do
        Result := False
    end

```

Now a class *BOUNDED_LIST* that implements bounded-size lists (inheriting, like the earlier *ARRAYED_LIST*, from both *LIST* and *ARRAY*) may redefine *full*:

```

full: BOOLEAN
    -- Is representation full?
    -- (Answer: if and only if number of items is capacity)
    do
        Result := (count = capacity)
    end

```

Procedure *extend* remains applicable as before; any client that used it properly with *LIST* can rely polymorphically on the *FIXED_LIST* implementation. The abstract precondition of *extend* has not changed, even though the concrete implementation of that precondition has in fact been strengthened.

Note that a class such as *BOUNDED_LIST*, the likes of which indeed appear in EiffelBase, is not a violation of the Eiffel advice to stay away from fixed-size structures. The corresponding structures are bounded, but the bounds are changeable. Although *extend*

requires **not full**, another feature, called *force* by convention, will be available to work in all cases, resizing (and possibly reallocating) the structure if necessary. Even arrays in Eiffel are not fixed-size, and have a procedure *force* with no precondition, accepting any index position.

The Assertion Redeclaration rule, together with the Invariant Accumulation rule, provides the right methodological perspective for understanding inheritance and the associated mechanisms. Defining a class as inheriting from another is a strong commitment; it means inheriting not only the features but the logical constraints. Redeclaring a routine is a committing decision: it means that you are providing a new implementation (or, for an effecting, a first implementation) of a previously defined semantics, as expressed by the original contract. Usually you have a wide margin for choosing your implementation, since the contract only defines a range of possible behaviors (rather than just one behavior), but you **must** remain within that range. Otherwise you would be perverting the goals of redeclaration, using this mechanism as a sort of late-stage hacking to override bugs in ancestor classes.

Join and uneffecting

It is not an error to inherit two deferred features from different parents under the same name, provided they have the same signature (number and types of arguments and result). In that case a process of **feature join** takes place: the features are merged into just one — with their preconditions and postconditions, if any, respectively or-ed and and-ed.

More generally, it is permitted to have any number of deferred features and *one* effective feature that share the same name: the effective version will apply to all the others.

All this is not a violation of the Final Name rule (page 58), since the name clashes prohibited by the rule involve two *different* features having the same final name; here the result is just *one* feature, resulting from the join of all the inherited versions.

Sometimes we may want to join *effective* features inherited from different parents, assuming again the features have compatible signatures. One way is to redefine them all into a new version; then they again become one feature, with no name clash in the sense of the Final Name rule. But in other cases we may simply want one of the inherited implementations to take over the others. The solution is to revert to the preceding case by **uneffecting** the other features; uneffecting an inherited effective feature makes it deferred (this is the reverse of

effecting, which turns an inherited deferred feature into an effective one). The syntax uses the **undefine** subclause:

```

class D inherit
  A
    rename
      g as f      -- g was effective in A
    undefine
      f
    end
  B
    undefine f end  -- f was effective in B
  C
    -- C also has an effective feature f, which will serve as implementation
    -- for the result of the join.
feature
  ...

```

Again what counts, to determine if there is an invalid name clash, is the final name of the features. In this example two of the joined features were originally called *f*; the one from *A* was called *g*, but in *D* it is renamed as *f*, so without the undefinition it would cause an invalid name clash.

Feature joining is the most common application of uneffecting. In some non-joining cases, however, it may be useful to forget the original implementation of a feature and let it start a new life devoid of any burden from the past.

Changing the export status

Another Feature Adaptation subclause makes it possible to change the export status of an inherited feature. By default — covering the behavior desired in the vast majority of practical cases — an inherited feature keeps its original export status (exported, secret, selectively exported). In some cases, however, this is not appropriate:

- A feature may have played a purely implementation-oriented role in the parent, but become interesting to clients of the heir. Its status will change from secret to exported.
- In implementation inheritance (for example *ARRAYED_LIST* inheriting from *ARRAY*) an exported feature of the parent may not be suitable for direct use by clients of the heir. The change of status in this case is the reverse of the previous one.

You can achieve either of these goals by writing

```
class D inherit
  A
  export {X, Y, ...} feature1, feature2, ... end
  ...
```

This gives a new export status to the features listed (under their final names since, as noted, **export** like all other subclauses comes after **rename** if present): they become exported to the classes listed. In most cases this list of classes, *X, Y, ...*, consists of just *ANY*, to re-export a previously secret feature, or *NONE*, to hide a previously exported feature. It is also possible, in lieu of the feature list, to use the keyword **all** to apply the new status to all features inherited from the listed parent. Then there can be more than one class-feature list, as in

```
class ARRAYED_LIST [G] inherit
  ARRAY [G]
  rename
    count as capacity, item as array_item, put as array_put
  export
    {NONE} all
    {ANY} capacity
  end
  ...
```

where any explicit listing of a feature, such as *capacity*, takes precedence over the export status specified for **all**. Here most features of *ARRAY* are secret in *ARRAYED_LIST*, because the clients should not be permitted to manipulate array entries directly — they will manipulate them indirectly through list features such as *extend* and *item*, whose implementation relies on *array_item* and *array_put* —; but *ARRAY*'s feature *count* remains useful, under the name *capacity*, to the clients of *ARRAYED_LIST*.

Flat and flat-short forms

Thanks to inheritance, it is possible to write a concise class text which achieves a lot, relying on all the features inherited from direct and indirect ancestors.

This is part of the power of the object-oriented form of reuse, but can create a comprehension and documentation problem when the inheritance structures become deep: how does one understand such a class, either as client author or as maintainer? For clients, the short form, which only considers the class text, does not tell the full story about available features; and for maintainers, much of the information must be sought in proper ancestors.

These observations suggest a need for mechanisms that will produce, from a class text, a version that is equivalent feature-wise and assertion-wise, but has no inheritance dependency at all. This is called the **flat form** of the class. It is a class text that has no inheritance clause and includes all the features of the class, immediate (declared in the class itself) as well as

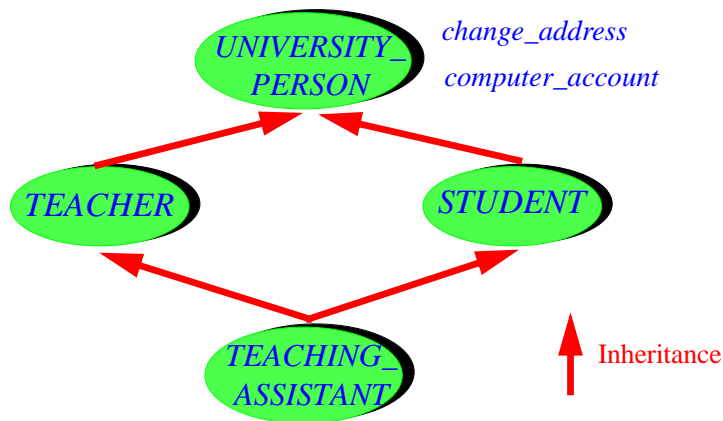
inherited. For the inherited features, the flat form must of course take account of all the feature adaptation mechanisms: renaming (each feature must appear under its final name), redefinition, effecting, uneffecting and export status change. For redeclared features, **require else** clauses are or-ed with the precursors' preconditions, and **ensure then** clauses are and-ed with precursors' postconditions. For invariants, all the ancestors' clauses are concatenated. As a result, the flat form yields a view of the class, its features and its assertions that conforms exactly to the view offered to clients and (except for polymorphic uses) heirs.

An Eiffel environment should provide tools to produce the flat form of a class. (In the ISE environment users will click on the “Flat” button in the “Class Tool” to get it.)

The short form (“[The short form of a class](#)”, page 40) of the flat form of a class, known as its flat-short form, is the complete interface specification, documenting all exported features and assertions — immediate or inherited — and hiding implementation aspects. It is the appropriate documentation for a class.

Repeated inheritance and selection

An inheritance mechanism, following from multiple inheritance, remains to be seen. Through multiple inheritance, a class can be a proper descendant of another through more than one path. This is called repeated inheritance and can be indirect, as in the following figure, or even direct, when a class *D* lists a class *A* twice in its **inherit** clause.



This particular example is in fact often used by introductory presentations of *multiple* inheritance, which is a grave pedagogical mistake: simple multiple inheritance examples (such as *INTEGER* inheriting from *NUMERIC* and *COMPARABLE*, *COMPANY_PLANE* from *ASSET* and *PLANE* and so on) should involve the combination of **separate abstractions**. Repeated inheritance is an advanced technique; although precious, it does not arise in elementary uses and requires a little more care.

In fact there is only one non-trivial issue in repeated inheritance: what does a feature of the repeated ancestor, such as *change_address* and *computer_account*, mean for the repeated

descendant, here *TEACHING_ASSISTANT*? (The example features chosen involve a *Indirect* and an attribute; the basic rules will be the same.)

repeated inheritance

There are two possibilities: sharing (the repeatedly inherited feature yields just one in the repeated descendant) and duplication (it yields two). Examination of various shows quickly that a fixed policy, or one that would apply to all the features of a class be inappropriate. A feature such as *change_address* calls for sharing: as a teaching you may be both teacher and student, but you are just one person and have just one domicile. But if there are different computers and accounts for students doing course work and for faculty, you will probably have two accounts, one as a student and one as a teacher.

The Eiffel rule enables, once again, the software developer to craft the resulting class so as to tune it to the exact requirements. Not surprisingly, it is based on names. In accordance with the Final Name rule (no in-class overloading):

Repeated Inheritance rule

- A feature inherited multiply under one name will be shared: it is considered to be just one feature in the repeated descendant.
- A feature inherited multiply under different names will be replicated, yielding as many variants as names.

So to tune the repeated descendant, feature by feature, for sharing and replication it suffices to use renaming. If you do nothing, you will obtain sharing, which is indeed in most cases the desired policy (especially for those cases of unintended repeated inheritance: making *D* inherit from *A* even though it also inherits from *B*, which you forgot is already a descendant of *A*). If you use renaming somewhere along the way, so that the final names are different, you will obtain two separate features. Note that it does not matter where the renaming occurs; all that counts is whether in the common descendant, *TEACHING_ASSISTANT* in the last figure, the names are the same or different. So you can use renaming at that last stage to cause replication; but if the features have been renamed higher you can also use last-minute renaming to *avoid* replication, by bringing them back to a single name.

The Repeated Inheritance rule gives the desired flexibility to disambiguate the meaning of repeatedly inherited features. There remains a problem in case of redeclaration and polymorphism. Assume that somewhere along the inheritance paths one or both of two replicated versions of a feature *f*, such as *computer_account* in the example, has been redeclared; we need to define the effect of a call *a.f(a.computer_account* in the example) if *a* is of the repeated ancestor type, here *UNIVERSITY_PERSON*, and has become attached as a result of polymorphism to an instance of the repeated descendant, here *TEACHING_ASSISTANT*. If one or more of the intermediate ancestors has redefined its version of the feature, the dynamically-bound call has two or more versions to choose from.

The ambiguity is resolved here through a **select** clause, as in

```
class TEACHING_ASSISTANT inherit
  TEACHER
  rename
    computer_account as faculty_account
  select
    faculty_account
  end
  STUDENT
  rename
    computer_account as student_account
  end
  ...
```

The assumption here is that no other renaming has occurred — *TEACHING_ASSISTANT* takes care of the renaming to ensure replication — but that one of the two parents has redefined *computer_account*, for example *TEACHER* to express the special privileges of faculty accounts. In such a case the rule is that one (and exactly one) of the two parent clauses in *TEACHING_ASSISTANT* must **select** the corresponding version. Note that no problem arises for an entity declared as

```
ta: TEACHING_ASSISTANT
```

since the valid calls are of the form *ta.faculty_account* and *ta.student_account*, neither of them ambiguous (the call *ta.computer_account* would be invalid, since after the renamings class *TEACHING_ASSISTANT* has no feature of that name). The **select** only applies to a call

```
up.computer_account
```

with *up* of type *UNIVERSITY_PERSON*, dynamically attached to an instance of *TEACHING_ASSISTANT*; then the **select** resolves the ambiguity by causing the call to use the version from *TEACHER*. For example if we traverse a data structure of the form *computer_users: LIST [UNIVERSITY_PERSON]* to print some information about the computer account of each element in the list, the account used for a teaching assistant is the faculty account, not the student account. (Note that we can, if desired, redefine *faculty_account* in class *TEACHING_ASSISTANT*, using *student_account* if necessary, to take into consideration the existence of another account. But in all cases we need a precise disambiguation of what *computer_account* means for a *TEACHING_ASSISTANT* object known only through a *UNIVERSITY_PERSON* entity.)

The **select** is only needed in case of replication. If the Repeated Inheritance rule would imply sharing, as with *change_address*, and one or both of the shared versions has been redeclared, the Final Name rule makes the class invalid, since it now has **two different features** with the same name. (This is only a problem if both versions are effective; if one or both are deferred there is no conflict but a mere case of feature joining as explained in [“Join and uneffecting”](#), page 62.) The two possible solutions follow from the previous discussions:

- If you do want sharing, one of the two versions must take precedence over the other. It suffices to **undefine** the other, and everything gets back to order. Alternatively, you can redefine both into a new version, which takes precedence over both.
- If you want to keep both versions, switch from sharing to replication: rename one or both of the features so that they will have different names; then you must **select** one of them.

Constrained genericity

Eiffel's inheritance mechanism has an important application to extending the flexibility of the **genericity** mechanism. In a class *SOME_CONTAINER* [*G*], as noted ([1.7](#)), the only operations available on entities of type *G*, the formal generic parameter, are those applicable to entities of all types. A generic class may, however, need to assume more about the generic parameter, as with a class *SORTABLE_ARRAY* [*G*...] which will have a procedure *sort* that needs, at some stage, to perform tests of the form

```
if item (i) < item (j) then ...
```

where *item* (*i*) and *item* (*j*) are of type *G*. But this requires the availability of a feature *plus alias* "<" in all types that may serve as actual generic parameters corresponding to *G*. Using the type *SORTABLE_ARRAY* [*INTEGER*] should be permitted, because *INTEGER* has such a feature; but not *SORTABLE_ARRAY* [*MATRIX*] if there is no total order relation on *MATRIX*.

To cover such cases, declare the class as

```
class SORTABLE_ARRAY [G -> COMPARABLE] ... The rest as before ...
```

making it **constrained generic**. The symbol \rightarrow recalls the arrow of inheritance diagrams; what follows it is a type, known as the generic constraint. Such a declaration means that:

- Within the class, all features of the generic constraint — here all features of *COMPARABLE*: *is_less alias* "<", *is_less_equal alias* "<=" etc. — may be applied to entities of type *G*.
- A generic derivation is only valid if the chosen actual generic parameter conforms to the constraint. Here we can use *SORTABLE_ARRAY* [*INTEGER*] since *INTEGER* is a descendant of *COMPARABLE*, but not *SORTABLE_ARRAY* [*INTEGER*] if *MATRIX* is not a descendant of *COMPARABLE*.

A class can have a mix of constrained and unconstrained generic parameters, as in the EiffelBase class *HASH_TABLE* [*G*, *H* -> *HASHABLE*] whose first parameter represents the types of objects stored in a hash table, the second representing the types of the keys used to store them, which must be *HASHABLE*. As these examples suggest, structural property classes such as *COMPARABLE*, *NUMERIC* and *HASHABLE* are the most common choice for generic constraints.

Unconstrained genericity, as in *C* [*G*], is defined as equivalent to *C* [*G* -> *ANY*].

Assignment attempt

The Type Conformance rule (“[Polymorphism](#)”, [page 47](#)) ensures type safety by requiring all assignments to be from a more specific source to a more general target.

In some cases, the type of the target object cannot be known for sure. This happens for example when the target comes from the outside — a file, a database, a network. The persistence storage mechanism (“[Deep operations and persistence](#)”, [page 27](#)) includes, along with the procedure *store* seen there, the reverse operation, a function *retrieved* which yields an object structure retrieved from a file or network, to which it was sent using *store*. But *retrieved* as declared in the corresponding class *STORABLE* of EiffelBase can only return the most general type, *ANY*; the exact type can only be ascertained at execution time, since the corresponding objects are not under the control of the retrieving system, and might even have been corrupted by some external agent.

In such cases we cannot trust the declared type but must check it against the type of an actual run-time object. Eiffel introduces for this purpose the **assignment attempt** operation, written

```
x ?= y
```

with the following effect (only applicable if *x* is a variable entity of reference type):

- If *y* is attached, at the time of the instruction’s execution to an object whose type conforms to the type of *x*, perform a normal reference assignment.
- Otherwise (if *y* is void, or attached to a non-conforming object), make *x* void.

Using this mechanism, a typical object structure retrieval will be of the form

```
x ?= retrieved
if x = Void then
    “We did not get what we expected”
else
    “Proceed with normal computation, which will typically involve
    calls of the form x.some_feature”
end
```

As another application, assume we have a *LIST [ACCOUNT]* and class *SAVINGS_ACCOUNT*, a descendant of *ACCOUNT*, has a feature *interest_rate* which was not

in *ACCOUNT*. We want to find the maximum interest rate for savings accounts in the list. Assignment attempt easily solves the problem:

```

local
  s: SAVINGS_ACCOUNT
do
  from account_list.start until account_list.after loop
    s ?= acc_list.item
      -- item from LIST yields the element at cursor position
    if s /= Void and then s.interest_rate > Result then
      -- Using and then (rather than and) ensures that
      -- s.interest_rate is not evaluated if s = Void is true.
      Result := s.interest_rate
    end
  account_list.forth
end
end

```

Note that if there is no savings account at all in the list the assignment attempt will always yield void, so that the result of the enclosing function will be 0, the default initialization.

Assignment attempt is useful in the cases cited — access to external objects beyond the software’s own control, and access to specific properties in a polymorphic data structure. The form of the instruction precisely serves these purposes; not being a general type comparison (but only a verification of a specific expected type) it does not carry the risk of encouraging developers to revert to multi-branch instruction structures, for which Eiffel provides the usually preferable alternative of polymorphic, dynamically-bound feature calls.

Covariance and anchored declarations

The final property of Eiffel inheritance involves the rules for adapting not only the implementation of inherited features (through redeclaration of either kind, redeclaration and redefinition, as seen so far) and their contracts (through the Assertion Redeclaration rule), but also their types. More general than type is the notion of a feature’s **signature**, defined by the number of its arguments, their types, the indication of whether it has a result (that is to say, is a function or attribute rather than a procedure) and, if so, the type of the result.

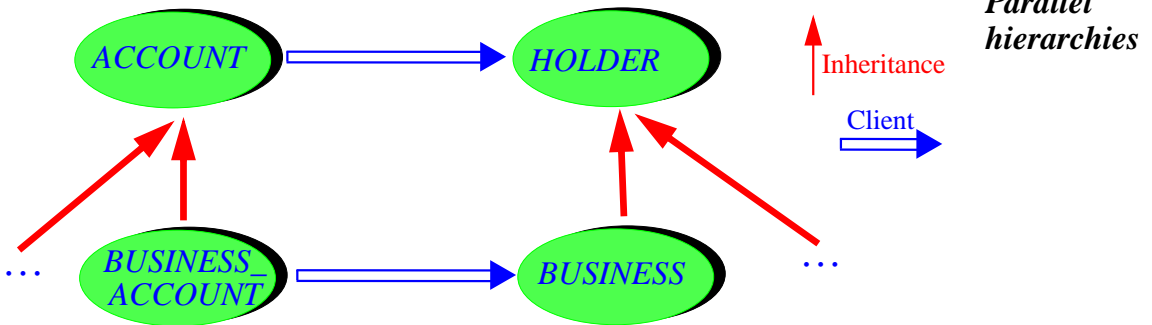
In many cases the signature of a redeclared feature remains the same as the original’s. But in some cases we may want to adapt it to the new class. For example if we assume that class *ACCOUNT* has features

```

owner: HOLDER
set_owner (h: HOLDER)
  -- Make h the account owner.
require
  not_void: h /= Void
do
  owner := h
end

```

Assume we introduce an heir *BUSINESS_ACCOUNT* of *ACCOUNT* to represent special business accounts, corresponding to class *BUSINESS* inheriting from *HOLDER*:



Clearly, *owner* must be redefined in class *BUSINESS_ACCOUNT* to yield a result of type *BUSINESS*; the same signature redefinition must be applied to the argument of *set_owner*. This case is fully typical of the general scheme of signature redefinition: in a descendant, you may need to redefine both query results and routine arguments to types conforming to the originals. This is reflected by a language rule:

Covariance rule

In a feature redeclaration, both the result type if the feature is a query (attribute or function) and the type of any argument if it is a routine (procedure or function) must conform to the original type as declared in the precursor version.

The term “covariance” reflects the property that all types — those of arguments and those of results — vary together in the same direction as the inheritance structure.

If a feature such as *set_owner* has to be redefined for more than its signature — to update its implementation or assertions — explicit signature redefinition is acceptable. For example *set_owner* could do more for business owners than it does for ordinary owners. Then the redefinition will be of the form

But in many cases the body will be exactly the same as in the precursor. Then explicit redefinition would be unbearably tedious, implying constant text duplication. The mechanism

```

set_owner (b: BUSINESS)
    -- Make b the account owner.
    ... New routine body ...
end

```

of **anchored redeclaration** solves this problem. The original declaration of *set_owner* in *ACCOUNT* should be of the form

```

set_owner (h: like Current)
    -- Make h the account owner.
    -- The rest as before:
require
    not_void: h /= Void
do
    owner := h
end

```

A **like anchor** type, known as an anchored type, may appear in any context in which *anchor* has a well-defined type *T*: *anchor* can be an attribute or function of the enclosing class, or an argument of the enclosing routine. Then in the class in which it appears type **like anchor** means the same as *T*; for example, in *set_owner* above, the declaration of *h* has the same effect as if *h* had been declared of type *HOLDER*, the type of the anchor *owner* in class *ACCOUNT*. The difference comes in proper descendants: if a type redefinition changes the type of the anchor, any entity declared **like** the anchor will be considered to have been redefined too. So this is a form of implicit type redeclaration.

In the example, class *BUSINESS_ACCOUNT* only needs to redefine the type of *owner* (to *BUSINESS*). Then there is no need to redefine *set_owner* — except if we want to change its implementation or assertions.

It is possible to use *Current* as anchor; the declaration **like Current** denotes a type based on the current class (with the same generic parameters if any). This is in fact a common case; we saw in [“Structural property classes”, page 56](#), that it applies in class *COMPARABLE* to features such as

```

is_less alias "<" (other: like Current): BOOLEAN is ...

```

since we only want to compare two comparable elements of compatible types — but not, for example, integer and strings, even if both types conform to *COMPARABLE*. (A “balancing rule” makes it possible, however, to mix the various arithmetic types, consistently with mathematical traditions, in arithmetic expressions such as $3 + 45.82$ or boolean expressions such as $3 < 45.82$.)

Similarly, procedure *copy* is declared in class *ANY* as

```

copy (other: like Current) is ...

```

with both the argument anchored to the current object. Function *clone*, for its part, has signature *clone* (*other*: *ANY*): *like other*, showing a result anchored to the argument, so that the type of *clone* (x) for any x is the same as the type of x.

A final, more application-oriented example of anchoring to *Current* would be the feature *merge* posited in an earlier example (page 31) with the signature *merge* (*other*: *ACCOUNT*). By using instead *merge* (*other*: *like Current*) we can ensure that in any descendant class — *BUSINESS_ACCOUNT*, *SAVINGS_ACCOUNT*, *MINOR_ACCOUNT*... — an account will only be mergeable with another of a compatible type.

Covariance complicates somewhat the static type checking mechanism; mechanisms of “system validity” and “catcalls” address the problem, which is discussed in detail in the book [Object-Oriented Software Construction](#) (see the bibliography).

1.10 OTHER IMPORTANT MECHANISMS

We now examine a few supplementary mechanisms that complement the preceding picture: shared objects; constants; instructions; and lexical conventions.

Once routines, shared objects, smart initialization and on-demand execution

The Eiffel’s method obsession with extendibility, reusability and maintainability yields, as has been seen, highly modular and decentralized architectures, where inter-module coupling is limited to the strictly necessary, interfaces are clearly delimited, and all the temptations to introduce obscure dependencies, in particular global variables, have been removed. There is a need, however, to let various components of a system have access to common objects, without requiring their routines to pass these objects around as arguments (which would only be slightly better than global variables). For example various classes may need to perform output to a common “console window”, represented by a shared object.

Eiffel addresses this need through an original mechanism that also takes care of another important issue, poorly addressed by many design and programming approaches: initialization. The basic idea is very simple: if instead of **do** the implementation of an effective routine is introduced by the keyword **once**, it will only be executed the first time the routine is called during a system execution (or, in a multithreaded environment, the first time in each thread), regardless of what the caller was. Subsequent calls from the same caller or others will have no effect; if the routine is a function, it will always return the result computed by the first call — object if an expanded type, reference otherwise.

In the case of procedures, this provides a convenient initialization mechanism. A delicate problem in the absence of a **once** mechanism is how to provide the users of a library with a set of routines which they can call in any order, but which all need, to function properly, the guarantee that some context had been properly set up. Asking the library clients to precede the first call with a call to an initialization procedure *setup* is not only user-unfriendly but silly: in a well-engineered system we will want to check set-up in every of the routines, and report an error if necessary; but then if we were able to detect improper set-up we might as well shut up and set up (by calling *setup*) ourselves instead. This is not easy, however, since the object

on which we call *setup* must itself be properly initialized, so we are only pushing the problem further. Making *setup* a **once** procedure solves it: we can simply include a call

```
setup
```

at the beginning of each affected routine; the first one to come in will perform the needed initializations; subsequent calls will have, as desired, no effect. We may call this mechanism **smart initialization**.

Once functions will give us shared objects. A common scheme is

```
console: WINDOW
    -- Shared console window
once
    create Result.make (...)
end
```

The first call will create the appropriate window and return a reference to it. Subsequent calls, from anywhere in the system, will return that same reference. The simplest way to make this function available to a set of classes is to include it in a class *SHARED_STRUCTURES* which the classes needing a set of related shared objects will simply inherit.

For the classes using it, *console*, although a function, looks very much as if it was an attribute — only one referring to a shared object.

The “Hello World” system at the beginning of this chapter (1.4) used an output instruction of the form *io.put_string* (“*Some string*”). This is another example of the general scheme illustrated by *console*. Feature *io*, declared in *ANY* and hence usable by all classes, is a once function that returns an object of type *STANDARD_FILES* (another Kernel Library class) providing access to basic input and output mechanisms. Procedure *put_string* is one of them. Because basic input and output must all work on the same files, *io* should clearly be a once function, shared by all classes that need these mechanisms.

By default, “once” means “once in this execution”. The mechanism actually provides further versatility. You may include a *once key*, as in **once** (“*THREAD*”); this specifies, in a multi-threaded setup, that the routine will be executed once for each thread of execution. Another predefined once key is “*OBJECT*”, specifying that you want the routine to be executed once for each applicable object. For example, the function

```
historical_data: LARGE_AMOUNT_OF_INFORMATION
    -- Collection of supplementary pieces of information on this object
once (“OBJECT”)
    create Result.make (...)
    “Load in all the necessary information”
end
```


will retrieve a possibly large amount of information, but only once for each object, and only if needed (since retrieve the information for every instance would take up unrealistic space). This provides a convenient form of **on-demand execution**, which without the once mechanism might require tedious and repetitive programming.

As further refinements of the mechanism, you may select an arbitrary string as once key; a routine may list more than one once key, as in `once ("KEY1", "KEY2")`; and you may execute a call `onces.refresh ("SOME_KEY")`, using feature `onces` from class `ANY`, to reset all routines that have listed `"SOME_KEY"` as a once key, that is to say, cause them to execute their bodies again the next time they are called. To reset all once routines, use `onces.refresh_all`.

Constant attributes

The attributes studied earlier were variable: each represents a field present in each instance of the class and changeable by its routines.

It is also possible to declare constant attributes, as in

```
Solar_system_planet_count: INTEGER is 9
```

These will have the same value for every instance and hence do not need to occupy any space in objects at execution time. (In other approaches similar needs would be addressed by constants, as in Pascal or Ada, or macros, as in C.)

What comes after the `is` is a manifest constant: a self-denoting value of the appropriate type. Manifest constants are available for integers, reals, booleans (`True` and `False`), characters (in single quotes, as `'A'`, with special characters expressed using a percent sign as in `%N` for new line, `%B` for backspace and `%U` for null).

Manifest constants are also available for strings, using double quotes as in

```
User_friendly_error_message: INTEGER is "Go get a life!"
```

with special characters again using the `%` codes. It is also possible to declare manifest arrays using double angle brackets:

```
<<1, 2, 3, 5, 7, 11, 13, 17, 19>>
```

which is an expression of type `ARRAY [INTEGER]`. Manifest arrays and strings are not atomic, but denote instances of the Kernel Library classes `STRING` and `ARRAY`, as can be produced by once functions.

Instructions

Eiffel has a remarkably small set of instructions. The basic computational instructions have been seen: creation, assignment, assignment attempt, procedure call, **retry**. They are complemented by control structures: conditional, multi-branch, loop, as well as **debug** and **check**.

A conditional instruction has the form **if** ... **then** ... **elseif** ... **else** ... **end**. The **elseif** ... part (of which there may be more than one) and the **else** ... part are optional. After the **if** comes a boolean expression; after **then**, **elseif** and **else** come zero or more instructions.

A multi-branch instruction has the form

```
inspect
  exp
when 0 then
  inst1
when -1, 1..10 then
  inst2
...
else
  inst0
end
```

where the **else** *inst*₀ part is optional, *exp* is an expression of type *INTEGER* (as here), *CHARACTER* or *STRING*, the values listed in the **when** parts are constants of the corresponding type (or constant intervals such as *1..10*), listing all different values, and *inst*₀, *inst*₁, *inst*₂, ... are sequences of zero or more instructions.

The effect of such a multi-branch instruction, if the value of *exp* is one of the values listed or belongs to one of the intervals, is to execute the corresponding *inst*₁. If none of the *v*₁ matches, the instruction executes *inst*₀, unless there is no **else** part, in which case it triggers an exception (the appropriate behavior, since an absent **else** part is an explicit statement that the *exp* must have one of the values listed, unlike a conditional **if** *c* **then** *inst* **end** with no **else** part, which does nothing in the absence of an **else** part).

A variant of the multi-branch instruction lets you discriminate on the basis of the type of the run-time object attached to the value of an expression:

```
inspect
  exp
when {TYPE1} then
  inst1
when {TYPE2}, {TYPE3} then
  inst2
else
  inst0
end
```

This is useful when the various branches do not use the specific type of *exp*, as when discriminating between various kinds of exception in a **rescue** clause. For more sophisticated discrimination structures, dynamic binding usually provides a better solution.

The loop construct has the form

```
from
    initialization
until
    exit
invariant
    inv
variant
    var
loop
    body
end
```

The **invariant** *inv* and **variant** *var* parts are optional, the others required. *initialization* and *body* are sequences of zero or more instructions; *exit* and *inv* are boolean expressions (more precisely, *inv* is an assertion); *var* is an integer expression.

The effect is to execute *initialization*, then, zero or more times until *exit* is satisfied, to execute *body*. (If after *initialization* the value of *exit* is already true, *body* will not be executed at all.) Note that the syntax of loops always includes an initialization, as most loops require some preparation. If not, just leave *initialization* empty, while including the **from** since it is a required component.

The assertion *inv*, if present, expresses a **loop invariant** (not to be confused with class invariants). For the loop to be correct, *initialization* must ensure *inv*, and then every iteration of *body* executed when *exit* is false must preserve the invariant; so the effect of the loop is to yield a state in which both *inv* and *exit* are true. The loop must terminate after a finite number of iterations, of course; this can be guaranteed by using a **loop variant** *var*. It must be an integer expression whose value is non-negative after execution of *initialization*, and decreased by at least one, while remain non-negative, by any execution of *body* when *exit* is false; since a non-negative integer cannot be decreased forever, this ensures termination. The full-assertion-monitoring mode will check these properties of the invariant and variant after initialization and after each loop iteration, triggering an exception if the invariant does not hold or the variant is negative or does not decrease.

An occasionally useful instruction is **debug** (*Debug_key*, ...) *instructions* **end** where *instructions* is a sequence of zero or more instructions and the part in parentheses is optional, containing if present one or more strings (debug keys). Compilation options of the environment (specifying explicit debug keys, or just yes or no to govern the effect of debug instructions with no keys) make it possible to treat this instruction as executing the *instructions*, or doing nothing at all. The obvious use is for instructions that should be part of the system but executed only in some circumstances, for example to provide extra debugging information.

The final instruction is connected with Design by Contract. The instruction **check Assertions end**, where *Assertions* is a sequence of zero or more assertions, will have

no effect unless assertion monitoring is turned on at the Check level or higher. In that case it will evaluate all the assertions listed, having no further effect if they are all satisfied; if any one of them does not hold, however, the instruction will trigger an exception.

This instruction serves to state properties that are expected to be satisfied at some stages of the computation — other than the specific stages, such as routine entry and exit, already covered by the other assertion mechanisms such as preconditions, postconditions and invariants. A recommended use of **check** involves calling a routine with a precondition, where the call, for good reason, does not explicitly test for the precondition. Consider a routine of the form

```
r (ref: SOME_REFERENCE_TYPE)
  require
    not_void: r /= Void
  do
    r.some_feature
    ...
  end
```

Because of the call to *some_feature*, the routine will not work unless its precondition is satisfied. A call *a.r* (*x*) can appear as **if** *x* /= *Void* **then** *a.r* (*x*) **end**, but this is not the only possible scheme; for example if the preceding instruction is **create** *x* then we know *x* is not void and do not need to protect the call at all. In some cases, however, the argument showing that *x* is not void might be less obvious; for example *x* could have been obtained, in a non-adjacent part of the algorithm, as *clone* (*y*) for some *y* that we know is not void. It is good practice in this case to write the call as

```
  check
    x_not_void: x /= Void end
    -- Because x was obtained as a clone of y,
    -- and y is not void because [etc.]
  end
a.r (x)
```

Note the recommended convention: extra indentation of the **check** part to separate it from the algorithm proper; and inclusion of a comment listing the rationale behind the developer's decision not to check explicitly for the precondition.

In production mode with assertion monitoring turned off, this instruction will have no effect. But it will be precious for a maintainer of the software who is trying to figure out what it does, and in the process to reconstruct the original developer's reasoning. (The maintainer might of course be the same person as the developer, six months later.) And if the rationale is wrong somewhere, turning assertion checking on will immediately uncover the bug.

Lexical conventions

Eiffel software texts are free-format: distribution into lines is not semantically significant, and any number of successive space and line-return characters is equivalent to just one space. The style rules suggest indenting software texts as illustrated by the examples in this chapter.

About 65 names — all unabbreviated single English words, except for **elseif** which is made of two words — are reserved, meaning that they cannot be used to declare new entities. Most of them are keywords, serving only as syntactic markers, and conventionally written in boldface in texts such as the present one: **class**, **feature**, **inherit** etc. Other reserved words, such as *Current*, directly carry a semantic denotation.

Except in manifest character constants (appearing in single quotes, such as 'A') and character strings (appearing in double quotes, such as "*lower and UPPER*"), letter case is not significant, to avoid errors due to subtle differences in writing an identifier. The style rules are again quite strict: they suggest writing class names in upper case, as *ACCOUNT*, non-constant feature names and keywords in lower case, as *balance* and **class**, constant features and predefined entities and expressions with an initial lower case, as *Avogadro* and *Current*.

Successive declarations or instructions may be separated by semicolons. Eiffel's syntax has been so designed, however, that (except in rare cases) **the semicolon is optional**. Omitting semicolons for elements appearing on separate lines lightens text and is the recommended practice. For clarity, however, successive elements appearing on a single line should always be separated by semicolons.

1.11 CONCURRENCY AND FURTHER DEVELOPMENTS

Recent work has resulted in advanced mechanisms being made available to the Eiffel community in the area of concurrency, Internet development, multithreading, CORBA.

SCOOP

Many proposals have been made to make Eiffel support concurrent programming; an extensive bibliography may be found at <http://www.eiffel.com>. The most developed of these proposals is known as SCOOP — Simple Concurrent Object-Oriented Programming — and is the result of work performed between 1991 and 1996.

The key word in SCOOP is the first: “Simple”. SCOOP represents a minimal extension to Eiffel — one keyword, **separate** — and takes full advantage of the existing sequential Eiffel mechanisms, remaining fully compatible with the spirit of the method which it prolongs to its natural concurrent counterparts. In spite of its simplicity, it is extremely general, covering all known forms of concurrency, from multiple processes to Internet programming, multithreading and distributed computation (all implemented or being implemented in ISE's environment at the time of writing). The following summary is drawn from the chapter on concurrency in [Object-Oriented Software Construction](#).

We use the fundamental scheme of O-O computation: feature call, $x.f(a)$, executed on behalf of some object O1 and calling f on the object O2 attached to x , with the argument a .

But instead of a single processor that handles operations on all objects, we may now rely on different processors for O1 and O2 — so that the computation on O1 can move ahead without waiting for the call to terminate, since another processor handles it.

Because the effect of a call now depends on whether the objects are handled by the same processor or different ones, the software text must tell us unambiguously what the intent is for any x . Hence the need for the single new keyword: rather than just x : *SOME_TYPE*, we declare x : **separate** *SOME_TYPE* to indicate that x is handled by a different processor, so that calls of target x can proceed in parallel with the rest of the computation. With such a declaration, any creation instruction **create** x .*make* (...) will spawn off a new processor — a new thread of control — to handle future calls on x .

Nowhere in the software text should we have to specify *which* processor to use. All we state, through the *separate* declaration, is that two objects are handled by different processors, since this radically affects the system’s semantics. Actual processor assignment can wait until run time. Nor do we settle too early on the exact nature of processors: a processor can be implemented by a piece of hardware (a computer), but just as well by a task (process) of the operating system, or, on a multithreaded OS, just a thread of such a task. Viewed by the software, “processor” is an abstract concept; you can execute the same concurrent application on widely different architectures (time-sharing on one computer, distributed network with many computers, threads within one Unix or Windows task...) without any change to its source text. All you will change is a “Concurrency Configuration File” which specifies the last-minute mapping of abstract processors to physical resources.

We need to specify synchronization constraints. The conventions are straightforward:

- No special mechanism is required for a client to resynchronize with its supplier after a separate call $x.f(a)$ has gone off in parallel. The client will wait when and if it needs to: when it requests information on the object through a query call, as in $value := x.some_query$. This automatic mechanism is called *wait by necessity*.
- To obtain exclusive access to a separate object O2, it suffices to use the attached entity a as an argument to the corresponding call, as in $r(a)$.
- A routine precondition involving a separate argument such as a causes the client to wait until the precondition holds.
- To guarantee that we can control our software and predict the result (in particular, rest assured that class invariants will be maintained), we must allow the processor in charge of an object to execute at most one routine at any given time.
- We may, however, need to *interrupt* the execution of a routine to let a new, high-priority client take over. This will cause an exception, so that the spurned client can take the appropriate corrective measures — most likely retrying after a while.

This covers most of the mechanism, which will enable us to build the most advanced concurrent and distributed applications through the full extent of Eiffel techniques reviewed in this chapter, from multiple inheritance and behavior classes to static typing, dynamic binding and Design by Contract.

Other developments

(--- REWRITE OR REMOVE---) As part of the growth of Eiffel usage in large projects and its openness to the rest of the software world, a number of important developments, some already in the form of products, others in progress, have recently occurred:

- Development of multithreading libraries (which may be used without the SCOOP extensions of the preceding section for users using sequential Eiffel, or in conjunction with SCOOP).
- CORBA interfaces, as a result of a cooperation between ISE and ICL Ltd., and of a multi-vendor effort leading to a proposed official Eiffel binding for IDL, the Interface Definition Language of CORBA.
- Interfaces to Microsoft's OLE 2 and Active X.
- Interfaces to relational and object-oriented databases.
- Libraries or reusable components in many different areas (such as ISE's EiffelMath for scientific and financial applications).
- Java and Java bytecode generation, Java interfaces.
- Interfaces to many other industry-standard products.

PART II: LANGUAGE DESCRIPTION

This second part of the book presents the full description of the Eiffel language. It presents the formal elements (syntax, validity, semantics) interspersed with detailed explanations and examples.

For an extract containing only the formal elements, see part [VI](#).

Syntax, validity and semantics

2.1 OVERVIEW

To study the details of Eiffel, you will need a few conventions and basic rules. In particular, you will need to understand the role of the three levels of language description:

- **Syntax**, defining the textual structure of Eiffel software.
- **Validity**, defining when a syntactically well-formed software element has a meaning.
- **Semantics**, specifying what that meaning is, in terms of its effect on the software’s execution.

Each of these levels conditions the next: validity is only defined for a syntactically legal element, and semantics only for a valid element. This chapter defines the three levels more precisely and introduces the notations used, in the rest of the book, to describe the syntax, validity and semantics of Eiffel constructs. It also offers an overview of *correctness*, a part of semantics.

Before proceeding, you should have read the note about the language description style [after the Preface](#).

← “[About the language description](#)”, page xv.

2.2 SYNTAX: COMPONENTS, SPECIMENS, CONSTRUCTS

Eiffel’s syntax defines the structure of class texts.

The structure only: to express further limitations on legal texts, we need validity constraints; and to describe the effect of these texts, we need semantic rules.

→ See below [2.7](#) and [2.8](#), starting on page [96](#), about validity constraints, and [2.9](#) about semantics.

Syntax, BNF-E

Syntax is the set of rules describing the structure of software texts. The notation used to define Eiffel’s syntax is called **BNF-E**.

“BNF” is *Backus-Naur Form*, a traditional technique for describing the syntax of a certain category of formalisms (“*context-free languages*”), originally introduced for the description of Algol 60. BNF-E adds a few conventions — one production per construct, a simple notation for repetitions with separators — to make descriptions clearer. The range of formalisms that can be described by BNF-E is the same as for traditional BNF.

Here are the key syntax notions:



Component, construct, specimen

Any class text, or syntactically meaningful part of it, such as an instruction, an expression or an identifier, is called a **component**. The structure of any kind of components is described by a **construct**. A component of a kind described by a certain construct is called a **specimen** of that construct.

For example, any particular class text, built according to the rules given in this language description, is a *component*. The *construct* `Class` describes the structure of class texts; any class text is a *specimen* of that construct. At the other end of the complexity spectrum, an identifier such as `your_variable` is a specimen of the construct `Identifier`.

Although we could use the term “instance” in lieu of “specimen”, it could cause confusion with the instances of an Eiffel class — the run-time objects built according to the class specification.

An important convention will simplify the discussions:

Construct Specimen convention

The phrase “an **X**”, where **X** is the name of a construct, serves as a shorthand for “a specimen of **X**”.

For example, “a `Class`” means “a specimen of construct `Class`”: a text built according to the syntactical specification of the construct `Class`.

This example illustrates another convention

Construct Name convention

Every construct has a name starting with an upper-case letter and continuing with lower-case letters, possibly with underscores (to separate parts of the name if it uses several English words).

Besides `Class`, examples of construct names include `Parenthesized` and `Unlabeled_assertion_clause`. Every non-terminal appears in the index with the page of its syntactical definition. An appendix gives the full list.

→ [Appendix K, Syntax in alphabetical order](#)

Typesetting conventions complement the Construct Name convention: construct names, such as `Class`, always appear in Roman and in Green — distinguishing them from the blue of Eiffel text, as in `Result := x`.

→ [“Textual conventions”, page 94](#).

2.3 TERMINALS, NON-TERMINALS AND TOKENS

Every construct is either a “terminal” or a “non-terminal”:



Terminal, non-terminal, token

Specimens of a **terminal construct** have no further syntactical structure. Examples include:

- Reserved words such as **if** and **Result**.
- Manifest constants such as the integer *234*; symbols such as **;** (semicolon) and **+** (plus sign).
- Identifiers (used to denote classes, features, entities) such as *LINKED_LIST* and *put*.

The specimens of terminal constructs are called **tokens**.

In contrast, the specimens of a **non-terminal** construct are defined in terms of other constructs.

Chapter 32, about the lexical structure, explains the various kinds of tokens.

Tokens (also called **lexical components**) form the basic vocabulary of Eiffel texts. By starting with tokens and applying the rules of syntax you may build more complex components — specimens of non-terminals.

An example of non-terminal is **Conditional**, whose specimens are “conditional instructions” such as

```
if a > b then put (a) else put (b) end
```

Such a non-terminal construct specimen includes further syntactical components, here the expression *a > b* and the instructions *put (a)* and *put (b)* — themselves specimens of non-terminals, with further sub-components. We may represent the full structure as a “syntax tree” as illustrated on the next page.

2.4 THE LEXICAL LEVEL

As the figure shows, constructs defining keywords and symbols, such as **if** and **;**, do not have construct names, since they each have a single specimen (**if** etc.) which you can use directly to refer to the construct.

Other terminal constructs such as **Identifier** represent many possible specimens (an infinity of them in the case of **Identifier**). So, like non-terminal constructs, they need a name and a description of how to obtain their specimens. They are called **lexical** constructs. Other examples of lexical constructs include **Integer**, denoting unsigned integer constants, such as *598*, and **String**, denoting sequences of arbitrary characters. As noted earlier, the specimens of a lexical construct, such as individual integers or strings, are called *tokens*.



Production

A **production** is a formal description of the structure of all specimens of a non-terminal construct. It has the form

Construct \triangleq right-side
where **right-side** describes how to obtain specimens of the **Construct**.

The symbol \triangleq may be read aloud as “is defined as”.

BNF-E uses exactly one production for each non-terminal. The reason for this convention is explained below.

→ “*One production per non-terminal*”, page 93.



We need three kinds of right side:

Kinds of production

A production is of one of the following three kinds, distinguished by the form of the **right-side**:

- **Aggregate**, describing a construct whose specimens are made of a fixed sequence of parts, some of which may be optional.
- **Choice**, describing a construct having a set of given variants.
- **Repetition**, describing a construct whose specimens are made of a variable number of parts, all specimens of a given construct.

The rest of this section explores them in turn.

Aggregate productions

The **right-side** of an Aggregate production lists one or more constructs, some of which may be in square brackets to indicate optional parts. This specifies that, to obtain a specimen of the left-hand side construct, you simply provide a **succession** (“*aggregation*”) of specimens of the listed constructs, in the order given. So the production

→ Page 481.



Conditional \triangleq **if** Then_part_list [Else_part] **end**

indicates that a **Conditional** is made of the keyword **if**, followed by a **Then_part_list** (that is to say, a specimen of the **Then_part_list** construct), possibly followed by an **Else_part** — “possibly” because brackets indicate an optional component —, followed by the keyword **end**. More generally:

DEFINITION

Aggregate production

An **aggregate** right side is of the form $C_1 C_2 \dots C_n$ ($n > 0$), where every one of the C_i is a construct and any contiguous subsequence may appear in square brackets as $[C_i \dots C_j]$ for $1 \leq i \leq j \leq n$.

Every specimen of the corresponding construct consists of a specimen of C_1 , followed by a specimen of C_2 , ..., followed by a specimen of C_n , with the provision that for any subsequence in brackets the corresponding specimens may be absent.

Choice productions

A **Choice** production also lists a number of constructs, but with a different purpose: we want to state that a specimen of the left-hand side construct is a specimen of **one** — any one — of the listed constructs (rather than specimens of *all* the non-optional specimens, as in the aggregate case).

We separate the alternatives by vertical bars | to suggest exclusive choice. For example:

→ Page [328](#).

SYNTAX

$$\text{Type} \triangleq \text{Class_or_tuple_type} \mid \text{Formal_generic_name} \mid \text{Anchored}$$

This specifies that a **Type** is one of: a **Class_or_tuple_type**, a **Formal_generic_name**, an **Anchored**. More generally:

DEFINITION

Choice production

A **choice** right side is of the form $C_1 \mid C_2 \mid \dots \mid C_n$ ($n > 1$), where every one of the C_i is a construct.

Every specimen of the corresponding construct consists of exactly **one** specimen of one of the C_i .

Repetition productions

Use a Repetition production for a certain construct to express that its specimens are made of **any number** of specimens of some given construct, separated, if more than one, by a specified separator. The production will use braces, and three dots ... to suggest repetition; it will also list either an asterisk* if “any number” means “zero or more”, or a plus sign + to specify “one or more”.

So a right side of the form $\{C \ \$ \ \dots\}^+$ means “one or more specimens of C , to be separated by $\$$ ”, where $\$$ is a separator symbol or, more generally a construct. Replace $^+$ by $*$ for “zero or more”, allowing constructs with empty specimens.

For example:

→ Page [481](#).



$$\text{Then_part_list} \triangleq \{\text{Then_part} \ \mathbf{elseif} \ \dots\}^+$$

means that a specimen of **Then_part_list** — representing the “then part” of a conditional instruction, as specified by the production for **Conditional** shown earlier as an example of aggregate — consists of one or more **Then_part** clauses, separated, if more than one, by the keyword **elseif**. So typical specimens of **Then_part_list** are of the forms



$$\begin{aligned} &t1 \\ &t1 \ \mathbf{elseif} \ t2 \\ &t1 \ \mathbf{elseif} \ t2 \ \mathbf{elseif} \ t3 \end{aligned}$$

and so on, where $t1, t2, t3 \dots$ are specimens of **Then_part**.

If the production had used an asterisk $*$ instead of a plus $^+$, the empty text would also have been an acceptable specimen of **Then_part_list**.

More generally:



Repetition production, separator

A **repetition** right side is of one of the two forms

$$\begin{aligned} &\{C \ \$ \ \dots\}^* \\ &\{C \ \$ \ \dots\}^+ \end{aligned}$$

where C and $\$$ (the **separator**) are constructs.

Every specimen of the corresponding construct consists of zero or more (one or more in the second form) specimens of C , each separated from the next, if any, by a specimen of $\$$.

The following abbreviations may be used if the separator is empty:

$$\begin{aligned} &C^* \\ &C^+ \end{aligned}$$

The last two cases are not common, since most repetitions involve a separator, but for those that don't the simpler notation suffices.

Using recursive productions



You will note that many productions appear to define constructs recursively (that is to say, in terms of themselves). For example the grammar includes the following three productions:



$$\begin{aligned} \text{Instruction} &\triangleq \dots \text{ Other choices } \dots \mid \text{Conditional} \\ \text{Conditional} &\triangleq \text{if Then_part_list [Else_part] end} \\ \text{Else_part} &\triangleq \text{else Compound} \\ \text{Compound} &\triangleq \{ \text{Instruction } ";" \dots \}^+ \end{aligned}$$

which, taken together, show that **Instruction** is defined in terms of **Conditional**, defined in terms of **Else_part**, defined in terms of **Compound**, defined in terms of **Instruction**. This may seem strange (if you haven't seen syntax descriptions before), but in fact may make perfect sense.

Such recursive chains, to be useful, must always include a Choice production, with at least one branch leading to a construct entirely defined from terminals. Although the mathematical theory falls beyond the scope of this book, the general idea is that if the mutually recursive productions involving a construct **A** are



$$\begin{aligned} A &\triangleq T1 B T2 \\ B &\triangleq A \mid T3 \end{aligned}$$

where **T1**, **T2** and **T3** are terminal constructs, then the possible specimens of **A** are of the form

$$\begin{aligned} t1 t3 t2 \\ t1 t1 t3 t2 t2 \\ t1 t1 t1 t3 t2 t2 t2 \\ \dots \text{ and so on } \dots \end{aligned}$$

where *t1* is a specimen of **T1** etc.

Informally, you may view a set of mutually recursive productions as an equation $\mathbf{N} = \mathbf{T} + \mathbf{A} * \mathbf{N}$, where **N** is the vector of non-terminals (**A** and **B** in the last example), **T** is a vector of terminals, **A** is a matrix of terminals, + is choice (the same as |), and * is concatenation. Then the solution of the equation is $\mathbf{N} = \mathbf{T} + \mathbf{A} * \mathbf{T} + \mathbf{A}^2 * \mathbf{T} + \mathbf{A}^3 * \mathbf{T} + \dots$

For a detailed discussion of the theory of fix-points, which underlies these comments, see the book "Introduction to the Theory of Programming Languages".

The language definition makes only moderate use of recursion thanks to the availability of Repetition productions: when the purpose is simply to describe a construct whose specimens may contain successive specimens of another construct, a Repetition generally gives a clearer picture; see for example the definition of **Compound** as a repetition of **Instruction**. Recursion remains necessary to describe constructs with unbounded nesting possibilities, such as **Conditional** and **Loop**.

One production per non-terminal

The conventions of BNF-E ensure a property mentioned earlier:

Basic syntax description rule

Every non-terminal construct is defined by exactly one production.

Unlike in most BNF variants, every BNF-E production always uses exactly one of Aggregate, Choice and Repetition, *never* mixing them in the right sides. This convention yields a considerably clearer grammar, even if it has a few more productions (which in the end is good since they give a more accurate image of the language's complexity).

We do *not*, for example, define

$$\text{Type} \triangleq \dots \text{Other Choices} \dots \mid \text{like Anchor}$$

with the last Choice branch involving an Aggregate. Instead, we use two productions, one Choice and one Aggregate:

$$\begin{aligned} \text{Type} &\triangleq \dots \text{Other Choices} \dots \mid \text{Anchored} \\ \text{Anchored} &\triangleq \text{like Anchor} \end{aligned}$$

Non-production syntax rules

BNF-E and other BNF variants only cover a certain category of grammatical structures, known as “context-free”. Not all properties of interest are context-free; in addition some could in principle be described by context-free productions, but not easily.

To capture such properties we must use any applicable description technique, often just plain English (but as usual with much care and precision). We call such excursions from BNF “non-production” rules:

Non-production syntax rule

A **non-production syntax rule**, marked “(*non-production*)”, is a syntax property expressed outside of the BNF-E formalism.

Unlike validity rules, non-production syntax rules belong to the syntax, that is to say the description of the structure of Eiffel texts, but they capture properties that are not expressible, or not conveniently expressible, through a context-free grammar.

For example the BNF-E Aggregate productions allow successive right-side components to be separated by an arbitrary break — any sequence of spaces, tabs and “new line” characters. In a few cases, for example in an *Alias* declaration such as **alias** "+", it is convenient to use BNF-E — with a right-side listing the keyword **alias**, a double quote, an *Operator* and again a double quote — but we need to *prohibit* breaks between either double quote and the operator. We still use BNF-E to specify such constructs, but add a non-production syntax rule stating the supplementary constraints.

2.6 REPRESENTING TERMINALS

As shown by the preceding examples, the right sides of productions often list some terminals. This raises a problem for reserved words, which might be mistaken for construct names — consider for example the keyword **class** and the construct *Class* — and special symbols, some of which, such as {, [and +, are also used as symbols of the syntax notation.

The following conventions remove any ambiguity.

Textual conventions

The syntax (BNF-E) productions and other rules of the Standard apply the following conventions:

- 1 • Symbols of BNF-E itself, such as the vertical bars | signaling a choice production, appear in black (non-bold, non-italic).

- 2 • Any construct name appears in **dark green** (non-bold, non-italic), with a first letter in upper case, as **Class**.
 - 3 • Any component (Eiffel text element) appears in **blue**.
 - 4 • The double quote, one of Eiffel's special symbols, appears in productions as `""`: a double quote character (blue like other Eiffel text) enclosed in two single quote characters (black since they belong to BNF-E, not Eiffel).
 - 5 • All other special symbols appear in double quotes, for example a comma as `","`, an assignment symbol as `":="`, a single quote as `"""` (double quotes black, single quote blue).
 - 6 • Keywords and other reserved words, such as **class** and **Result**, appear in **bold** (blue like other Eiffel text). They do not require quotes since the conventions avoid ambiguity with construct names: **Class** is the name of a construct, **class** a keyword.
 - 7 • Examples of Eiffel comment text appear in non-bold, non-italic (and in blue), as `-- A comment`.
 - 8 • Other elements of Eiffel text, such as entities and feature names (including in comments) appear in non-bold *italic* (blue).
- The color-related parts of these conventions do not affect the language definition, which remains unambiguous under black-and-white printing (thanks to the letter-case and font parts of the conventions). Color printing is recommended for readability.

As an example, here is the syntactic definition of the construct **Compound**, given by a repetition production. A specimen of **Compound** is formed of zero or more specimens of **Instruction**, separated by semicolons:

Because of the difference between cases [1](#) and [3](#), `{` denotes the opening brace as it might appear in an Eiffel class text, whereas `{` is a symbol of the syntax description, used in repetition productions.

In case [2](#) the use of an upper-case first letter is a consequence of the “Construct Name convention”. ← Page [86](#).

Special symbols are normally enclosed in double quotes (case [5](#)), except for the double quote itself which, to avoid any confusion, appears enclosed in single quotes (case [4](#)). In either variant, the enclosing quotes — double or single respectively — are not part of the symbol.

In some contexts, such as the table of all such symbols, special symbols (cases [4](#) and [5](#)) appear in bold for emphasis. → Pages [890](#), [1154](#).

In application of cases [7](#) and [8](#), occurrences of Eiffel entities or feature names in comments appear in italics, to avoid confusion with other comment text, as in a comment

-- Update the value of *value*.

where the last word denotes a query of name *value* in the enclosing class.

Compound \triangleq {Instruction ";" ... } *

2.7 VALIDITY

The productions and other elements labeled SYNTAX, as described so far, specify the structure of constructs. In many cases, however, adherence to the structural requirements does not suffice to guarantee that a specimen of a construct will be meaningful.



For example, the following **Assignment** is built according to the syntactical specification of the corresponding construct:

$x := f. func(a + b, x)$

→ The specification of **Assignment** is on page [589](#). The right side in this example is a specimen of qualified **Call**, whose syntax appears on page [626](#).

But this does not mean that the **Assignment** will be acceptable in every possible context. It must also satisfy certain rules regarding the types of the components involved, the number of arguments passed to a routine such as *func* etc. Such supplementary requirements are called validity constraints



Validity constraint

A **validity constraint** on a construct is a requirement that every syntactically well-formed specimen of the construct must satisfy to be acceptable as part of a software text.

Validity constraints come in addition to syntactic rules, and are in fact defined only for what the definition calls “syntactically well-formed specimens”. In the **Assignment** example, the validity constraint is:



Assignment rule

VBAR

An **Assignment** is valid if and only if its source expression conforms to its target entity.

→ See a full discussion of this constraint on page [590](#).

(The “target entity” is the left side, x in the example; the “source expression” is the right side.)

Such validity constraints are introduced by the VALIDITY road sign as shown. Every constraint has a four-character code, here *VBAR*, uniquely identifying it. You do not need to pay any attention to these codes as you are first reading this book; but implementors of language processing tools, especially compilers, should include the appropriate code in any error message that reports a constraint violation. Then, if you get one of these error messages during system development, you will be able to look up the code in the index of this book, where they all appear under the heading “validity codes”, directing you to the detailed explanation of the language rule that you may have violated.

Some compilers, such as Eiffel Software’s EiffelStudio environment, give you the exact validity constraint, out of this book, as part of the error message.

The first letter of a validity code is always *V* (for “Validity”), the second one identifies the chapter, such as *B* for this chapter; the last two are a mnemonics for the constraint, for example *AR* for Assignment Rule.

A number of the validity rules have been reorganized from the previous editions. The appendix on changes gives the list of differences.

→ “[CHANGES IN VALIDITY CONSTRAINTS AND CONFORMANCE RULES](#)”.

Many constraints, such as the Feature Declaration rule, *VFFD*, list several conditions, each identified with a number. Error messages in this case should include not just the constraint code but also the number of the particular condition which was violated, for example *VFFD (2)*.

→ Page [162](#).



Valid

A construct specimen, built according to the syntax structure defined by the construct’s production, is said to be **valid**, and will be accepted by the language processing tools of any Eiffel environment, if and only if it satisfies the validity constraints, if any, applying to the construct.

2.8 INTERPRETING THE CONSTRAINTS

To avoid confusion, use the language properly, and benefit from the diagnostics of compilers and other tools, you must understand the precise nature of constraints and the conventions governing their interpretation.

Almost all the constraints listed in this book are *necessary and sufficient conditions*. This is not the usual style for other programming language descriptions, which commonly tell you that specimens of a certain construct *must* satisfy a certain property, or *may not* be of a certain form. Constraints in this book tell you instead that specimens of a certain construct will be valid **if and only if** they meet a specified set of requirements.

As discussed in the [Preface](#), such a form is preferable, since it allows you not just to detect that certain specific components are *not* valid, but also to ascertain without doubt whether an arbitrary component *is* valid. ← See “[FORMALITY](#)”, page [xvi](#).

This style requires a general convention. When reading the Assignment rule, **VBAR**, used in the previous section to illustrate the notion of constraint, it may have struck you that the rule cannot possibly suffice to ensure the validity of the example assignment: what about the validity of the right side, $f.\text{func}(a + b, x)$, which must satisfy all the validity constraints on function calls (*func* a properly defined and exported function applicable to objects of *f*'s type, with exactly two formal arguments of types matching the actual arguments given)?

Spelling out all such conditions on the components of a construct would lead to needlessly complex and repetitive validity constraints. Instead, all validity discussions rely on a universal interpretation rule:



General Validity rule

VBGV

Every validity constraint relative to a construct is considered to include an implicit supplementary condition stating that every component of the construct satisfies every validity constraint applicable to the component.

In the [Assignment](#) case, this means that constraint **VBAR** is considered to be automatically extended with the condition

“... and x satisfies all validity constraints on specimens of Variable, and y satisfies all validity constraints on specimens of Expression”

→ The constraint on [Variable](#) is the [Entity](#) rule, page [513](#); for the constraints on [Expression](#) see chapter [28](#).

so that, for the example [Assignment](#) above, the Assignment rule implicitly requires that $f.\text{func}(a + b, x)$ be a valid function call.

2.9 SEMANTICS

Lexical, syntactic and validity rules are only there to help us ensure that our software makes sense. The next question — even more important — is: what *is* that sense? The task of semantics is to answer that question.



Semantics

The **semantics** of a construct specimen that is syntactically legal and valid is the construct's effect on the execution of a system that includes the specimen.

The “effect” may include executing actions, producing a value, or both. It is defined by a rule marked SEMANTICS. For specimens having subcomponents, the rule will recursively refer to the semantics of the subcomponents.



The definition of “semantics” above explicitly assumes that the construct is syntactically legal and valid. When reading the SEMANTICS paragraphs, remember that they only apply to valid specimens. In many cases, the semantic rules would not even make sense otherwise; attempting to describe the effect of an invalid component is useless.



Most construct presentations will cover first syntax, then validity, then semantics. This is the expected order: first how to build language components of a certain kind; then what restrictions may exist on their parts; finally, what the result means. In a few cases, the semantics comes before the validity; such departures from the normal sequence occur when the best way to understand the reason for a constraint is to look first at the construct's effect in valid cases, and then find out what is required for that semantics to make sense. The change of order in such cases is, of course, only a pedagogical device; as everywhere else, the semantic specification is meaningless for invalid components.

2.10 CORRECTNESS

Validity is only a structural property; execution of valid Eiffel software may produce undesired results, or not terminate, or produce an exception that lead to failure.

→ “Failure” is a technical term defined in the discussion of exceptions in chapter 26.



The `loop from until False loop end` is a valid instruction, but, if executed, will never terminate.

Even for a valid component, then, we need a more advanced criterion: the component's ability to operate properly at run-time. This is called **correctness** and is a more elusive aim than validity, since it involves semantic properties.

This loop has an empty initialization (**from**), an empty loop body → Assertions, specifications and correctness are studied in chapter 9.

Ascertaining the correctness of an executable software component requires two pieces of information: what the component does (its implementation), but also what it is expected to do (its specification, or

contract). Eiffel supports both aspects: along with the executable elements of a class (the bodies of its routines, made of executable instructions), you may provide **assertions**, which state the contracts.

A class will be said to be correct if its features are guaranteed to perform according to their contracts.

2.11 TWO-TIER DEFINITION AND UNFOLDED FORMS

A number of Eiffel mechanisms provide high-level idioms for program schemes that could also be addressed — less concisely, or less elegantly — through other constructs. As a simple example, for a **Multi_branch** instruction dealing with character constants, you may use character intervals rather than listing individual characters; instead of writing

→ “*MULTI-BRANCH CHOICE*”, 17.4, page 482.



```
inspect
  char
when 'a', 'b', 'c', 'd', 'e' then [1]
  case1
else
  case2
end
```

you may replace the consecutive character choices by a single interval:



```
when 'a'.. 'e' then -- The rest as above [2]
```

The effect is the same but the text is simpler. We may call such language mechanisms *second-tier*, where the first tier would contain the constructs that cannot easily be expressed in terms of any others. The presence of second-tier constructs doesn’t contradict the Eiffel language design principle that “the language should provide *one* good way to do anything useful”, since the intention of the second-tier mechanisms is to provide a significantly better means of expression in applicable cases.

For such mechanisms the language definition often relies on the technique of **unfolded forms**. The idea is simply to define the properties of second-tier variants in terms of the more basic constructs; then it suffices to define the validity constraints, semantic specification, or both, for these basic forms. In the discussion of multi-branch instructions, the definition of “Unfolded form of a **Multi_branch**” reduces any variant of the instruction to one without intervals, so that the unfolded form of variant [2] above is [1]. After that, the validity and semantic definition for **Multi_branch** only address (through a number of intermediate definitions) the case of unfolded forms.

→ Page 486.

We will find unfolded forms useful for specifying the following constructs:

- Multiple declarations. → *Unfolded form definition: : page 159.*
- **Inheritance** parts, to ensure conformance of all types to **ANY**. → *Page 174.*
- **Only** clauses in postconditions. → *Page 244.*
- Assertions. → *Page 254.*
- **Precursor**. → *Page 306.*
- Anchored declarations. → *Page 344.*
- Formal generic parameters → *Page 353.*
- Tuples, through the notion of “anonymous class”.
- Conversion → *Page 416.*
- **Multi_branch** choice instruction and associated **Interval** definitions. → *Pages 486 and 486.*
- **Creators** part of a class. → *Page 548.*
- Creation instruction. → *Page 552.*
- Assigner call. → *Page 600.*
- Non-object call. → *Pages 630.*
- Once routine in the case of a “fresh” call. → *Pages 646.*
- Operator expressions, which we unfold into steps: first through the “Parenthesized Form” to remove potential ambiguities thanks to operator precedence; then through the “Equivalent Dot Form” reduce every expression to a **Call**. → *Pages 769 and 780.*

2.12 THE CONTEXT OF EXECUTING SYSTEMS

As explained in the next chapter, the executable units of Eiffel software are called “systems” (although many people also use the more traditional term “program”). The following terminology will serve to discuss the context of system execution:



Execution terminology

- **Run time** is the period during which a system is executed.
- The **machine** is the combination of hardware (one or more computers) and operating system through which you can execute systems.
- The machine type, that is to say a certain combination of computer type and operating system, is called a **platform**.
- **Language processing tools** serve to build, manipulate, explore and execute the text of an Eiffel system on a machine.

The most obvious example of a language processing tool is an Eiffel compiler or interpreter, which you can use to execute a system. But many other tools can manipulate Eiffel texts: Eiffel-aware editors, browsers to explore systems and their properties, documentation tools, debuggers, configuration management systems. Hence the generality of the term “*language processing tool*”.

2.13 TEXTUAL CONVENTIONS

Eiffel texts are written in “free format”: the only purpose of separating them into lines and including extra “white space” (space or tab characters) in these lines is to improve the readability of class texts, according to the style rules of a later chapter. → *Style guidelines are the topic of appendix 34.*



A language processing tool treats any sequence of line separations, spaces and tabs between lexical elements of the language as a single “break”, as explained in the chapter on the lexical structure. For such a tool the only relevant information is the presence of a break, not its precise makeup — for example its use of a line return rather than a space — which is interesting only for human readers. → “BREAKS”, 32.5, page 881

Eiffel is case-insensitive:

→ “Letter Case rule”, page 886

Case Insensitivity principle

In writing the letters of an **Identifier** serving as name for a class, feature or entity, or a reserved word, using the upper-case or lower-case versions has no effect on the semantics.

So you can write a class or feature name as *DOCUMENT*, *document* and even *dOcUmEnT* with exactly the same meaning.

Hence the definitions:



Upper name, lower name

The **upper name** of an **Identifier** or **Operator** *i* is *i* written with all letters in upper case; its **lower name**, *i* with all letters in lower case.



In the example the lower name is *document* and the upper name *DOCUMENT*.

The definition is mostly useful for identifiers, but the names of some operators, such as **and** and other boolean operators, also contain letters.

The reason for not letting letter case stand in the way of semantic interpretation is that it is simply too risky to let the meaning of a software text hang on fine nuances of writing, such as changing a letter into its upper-case variant; this can only cause confusion and errors. Different things should, in reliable and maintainable software, have clearly different names.

Letter case is of course significant in “manifest strings”, denoting texts to be taken verbatim, such as error messages or file names.

This letter case policy goes with strong rules on **style**:

- Classes and types should always use the upper name, as with a class *DOCUMENT*.
- Non-constant features and entities should always use the lower name, as with an attribute *document*.
- Constants and “once” functions should use the lower name with the first letter changed to upper, as with a constant attribute *Document*.



These rules are detailed in the corresponding chapter. They are for the benefit of your fellow human readers; language processing tools such as compilers will ignore them, except if they include an option for enforcing style standards. Do apply these standards: one of the attractions of Eiffel is its readability; consistency of Eiffel style, from Saõ Paulo to Sakhalin, makes it even better. → *Chapter 34*.

Another convention that greatly facilitates the writing and maintenance of Eiffel systems is the optional nature of semicolons:

Syntax (non-production): Semicolon Optionality rule

In writing specimens of **any** construct defined by a Repetition production specifying the semicolon ";" as separator, it is permitted, without any effect on the syntax structure, validity and semantics of the software, to omit any of the semicolons, or to add a semicolon after the last element.

This rule applies to instructions, declarations, successive groups of formal arguments, and many other Repetition constructs. It does not rely on the *layout* of the software: Eiffel's syntax is free-format, so that a return to the next line has the same effect as one or more spaces or any other "break". Rather than relying on line returns, the Semicolon Optionality rule is ensured by the syntax design of the language, which guarantees that omitting a semicolon never creates an ambiguity.

The rule also guarantees that an extra semicolon at the end, as in *a; b;* instead of just *a; b* is harmless.

The style guidelines suggest omitting semicolons (which would only obscure reading) for successive elements appearing on separate lines, as is usually the case for instructions and declarations, and including them to separate elements on a given line.

Because the semicolon is still formally in the grammar, programmers used to languages where the semicolon is an instruction *terminator*, who may then out of habit add a semicolon after every instruction, will not suffer any adverse effect, and will get the expected meaning.

The architecture of Eiffel software

3.1 OVERVIEW

The constituents of Eiffel software are called **classes**. To keep your classes and your development organized, it is convenient to group classes into **clusters**. By combining classes from one or more clusters, you may build executable **systems**.

These three concepts provide the basis for structuring Eiffel software:

- A *class* is a modular unit.
- A *cluster* is a logical grouping of classes.
- A *system* results from the assembly of one or more classes to produce an executable unit.

Of these, only “class”, describing the basic building blocks, corresponds directly to a construct of the language. To build clusters and systems out of classes, you will use not a language mechanism, but tools of the supporting environment.

Clusters provide an intermediate level between classes and systems, indispensable as soon as your systems grow beyond the trivial:

- At one extreme, a cluster may be a simple group of a few classes.
- At the other end, a system as a whole is simply a cluster that you have made executable (by selecting a *root class* and a *root procedure*).
- In-between, a cluster may be a library consisting of several subclusters, or an existing system that you wish to integrate as a subcluster into a larger system.

Clusters also serve to store and group classes using the facilities of the underlying operating system, such as files, folders and directories.

After the basic definitions, the language description will concentrate on classes, indeed the most important concept in the Eiffel method, which views software construction as an industrial production activity: combining components, not writing one-of-a-kind applications.

The present chapter introduces the overall structure of Eiffel software by discussing in turn the notions of class, system and cluster.

3.2 CLASSES

Classes are not just the modular units of software decomposition: they also serve as a basis for the types of Eiffel.

“Object-Oriented Software Construction” discusses the practical and theoretical roles of classes.

This dual view is essential to understanding the notion of class and, more generally, the principles of object-oriented software construction:

- As a decomposition unit, a class is a module, that is to say a group of related **services** packaged together into a named unit.
- As a type, a class is the description of similar run-time data elements, or *objects*, called the **instances** of the class.

→ Chapter 19 explains the precise nature of objects.

Although these two roles may at first seem rather different, it is in fact useful to support them through a single concept — the class — on the basis of an important observation (the starting point of the theory of *Abstract Data Types*): a good way of describing a set of similar objects without describing their implementation is to list the operations applicable to them. But then if the objects are all instances of the same class, we can define that class, viewed as a module, so that the services it offers are precisely the operations available on the instances of the class, viewed as a type.

This identification of services on modules with operations on instances is what makes it possible to merge the module and type views into the single concept of class. The **features** of a class are these services-operations.



For example, a document processing system could have classes such as *DOCUMENT*, *PARAGRAPH*, *FONT*, *TEXT_DISPLAY*. These are the modular units of the system; their texts can be processed by an Eiffel language processing tool, such as a compiler. They also describe possible run-time objects: documents, paragraphs, fonts, displayable views of text. Systems that include the given classes will be able to create such objects, modify them, and access their properties.

Each of these classes will contain features; for example, *PARAGRAPH* may include features *indent*, describing an operation that indents a paragraph, and *line_count*, to determine the number of lines of a paragraph.

To create an instance of a class, you may use a **creation instruction**; a typical form is

→ Chapter 20.

```
create x.cp (...)
```

where *x* is the name of the entity that will denote the newly created object, and *cp* is one of the features of the class, which must have been designated as a **creation procedure**. This creates an object, makes it accessible through the name *x*, and applies *cp* to initialize it. For example you might create an instance of class *DOCUMENT* through



```
create new_text.make ("Isabelle", 250)
```


assuming *DOCUMENT* has a creation procedure *make* with two arguments: a string for the author's name, an integer for the expected number of pages.

A bit of more precise terminology is useful here. An instance of a class *C* resulting from a creation instruction on a target of the corresponding type is called a **direct instance** of *C*; in the last example, *new_text* will be attached to a direct instance of *DOCUMENT*. The reason for this term is that with the introduction of inheritance we will consider direct instances of *proper descendants* of *C* also as instances (not direct) of *C*.

3.3 CLASS TEXTS AND CLASS NAMES

Every class has a class name, such as *DOCUMENT* or *PARAGRAPH*, and a class text describing the features of the class and its other properties. → *Chapter 4* presents the structure of class texts.

As you know, letter case is not significant in identifiers, so that you can write a class name as *doCumEnt* if you really want to. But this is strongly discouraged. The standard style is to write all class names using their upper names, such as *DOCUMENT*. → *Appendix 34* presents style rules.
← *The upper name is the name all in upper case. See "TEXTUAL CONVENTIONS", 2.13, page 102.*



When you want to display a class in EiffelStudio, you may type its name in any mix of lower and upper case (lower case is usually more convenient); the tools will display the upper name.

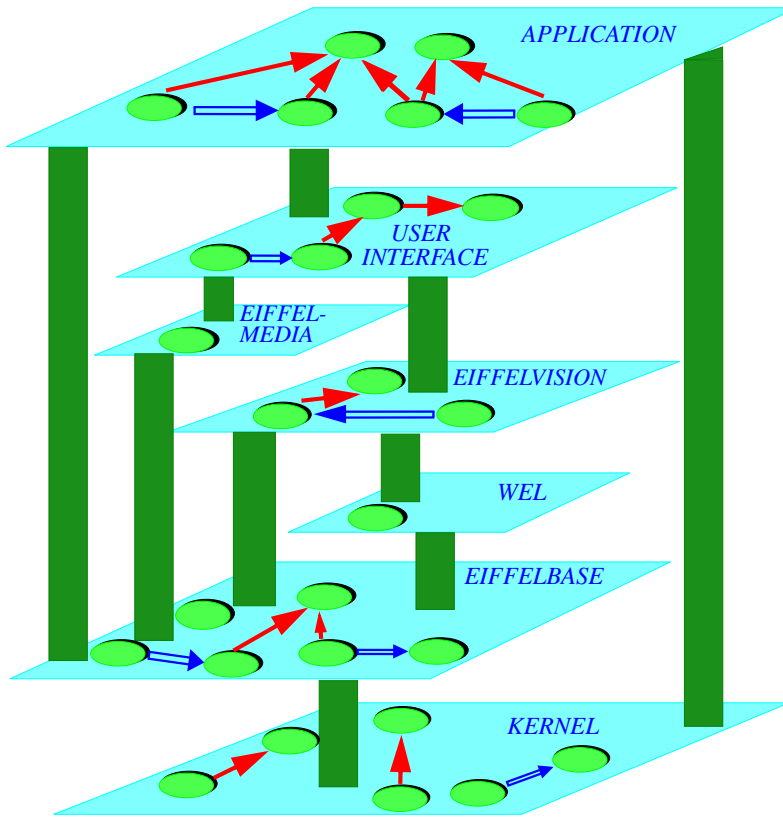
The classes of a system must all, as discussed below, have different names. → *"Class Name rule", page 111.*

3.4 CLUSTERS

As the number of classes in your systems grows, you will need to arrange these classes into groups, called clusters.

Clusters correspond to the major divisions of a system. For example, a compiling system may include a lexical cluster, a parsing cluster, a semantic analysis cluster, an optimization cluster, a generation cluster. A cluster may encompass a library, such as EiffelBase or EiffelVision; or it may be an application cluster, encompassing a logically significant subset of a system's specific classes.

The figure on the next page illustrates a typical system structure as a set of layers, each representing a cluster. Every cluster of this example except *KERNEL* relies on others through pillars, representing the dependency relations, client and inheritance, between the clusters' classes. The lower clusters, which normally should be built first, provide the basic capabilities; the higher clusters are more specialized, including *APPLICATION* which is assumed to cover the application-specific facilities of the system. In practice, of course, a system may include several application clusters. *The analogy with a physical construction works only to a point; the author and publisher decline any responsibility should you build your house with the architecture shown.*



A possible cluster structure

Here some of the clusters are Eiffel Software Libraries.

In each cluster, some classes are shown, with possible inheritance (single arrow) and client (double arrow) links.

You may nest clusters; a cluster included in another is called a **subcluster**. So we may represent a structure of classes and clusters as a tree, as shown at the top of the facing page. With this structure, a system as a whole is a cluster; a library is a cluster; and if you want to embed an existing system (itself having such a nested structure) as a subsystem in a larger system, you'll make it one of its subclusters. Such arbitrary nesting is part of the Eiffel method's support for software reuse and composition:

It is useful to define these notions precisely:



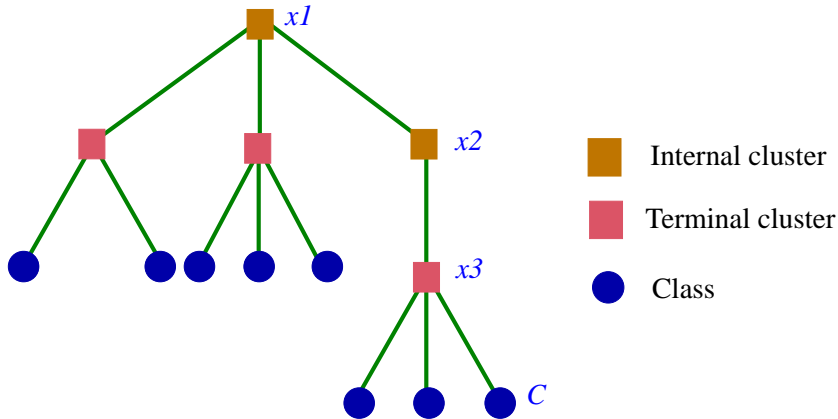
Cluster, subcluster, contains directly, contains

A **cluster** is a collection of classes, (recursively) other clusters called its **subclusters**, or both. The cluster is said to **contain directly** these classes and subclusters.

A cluster **contains** a class *C* if it contains directly either *C* or a cluster that (recursively) contains *C*.

In the presence of subclusters, several clusters may contain a class, but exactly one contains it directly.

Clusters, subclusters and classes



In the figure clusters $x1$, $x2$ and $x3$ all contain class C ; the one that contains C directly is $x3$. These observations lead us to define two kinds of cluster:



Terminal cluster, internal cluster

A cluster is **terminal** if it contains directly at least one class.

A cluster is **internal** if it contains at least one subcluster.



From these definitions, it is possible for a cluster to be both terminal and internal.

This is not the recommended style, however; the *methodological advice* is to keep the two cases separate, so that terminal clusters will contain only classes and internal clusters will contain directly only subclusters. This is the case in the example of the last figure.



Beyond this advice, there is no absolute rule on how to group classes into clusters. It is usually wise to observe the following informal criteria:

- The classes in a cluster should be conceptually related.
- In most cases, the number of classes in a terminal cluster should not exceed 20. You should consider splitting a terminal cluster into subclusters if it reaches that size, unless you feel that the classes are strongly connected and the cluster has a “flat” structure with no obvious criterion for splitting it.
- Cycles in the client relation should, in general, only involve classes that all belong to the same terminal cluster, avoiding cases in which A is a client of B and B a client of A with A and B in different clusters.
- For any terminal cluster, there should be at least one person who understands the cluster in its entirety.

→ “Client” is a relation between classes, studied in chapter 7. Cycles in the relation are explicitly permitted; see “SIMPLE CLIENTS”, 7.4, page 193.



Do not look, however, for a cluster construct in Eiffel. The highest-level construct is the class; clusters, although essential for organizing Eiffel software and managing its development, do not require language support. This is because such support would in most cases be redundant with the facilities provided by operating systems. If, as may be expected, classes are kept in files, then clusters will use the operating system mechanisms available to support the grouping of related files: folders (the Windows/Macintosh term) and directories (Unix/Linux). If, as suggested above, you make a clear separation between terminal and internal clusters, then some cluster folders will only contain class files, and the others will only contain subfolders. In the figure on the preceding page, squares then represent folders and the circles represent class files.

Eiffel tools, unlike the Eiffel language, should support clusters. The notion of cluster is also prominent in the [Lace control language](#).

→ *Lace is a control language for assembling, compiling and executing Eiffel systems, covered by appendix B.*

3.5 SYSTEMS

By themselves, classes are only building blocks. To obtain an executable software element, you must assemble one or more classes into a **system** and designate one of them as the “root”. Here are the precise definitions.

We start with a “universe” of classes:



Universe
A **universe** is a set of classes.

The universe provides a reference from which to draw classes of interest for a particular system. Any Eiffel environment will provide a way to specify a universe.



For example, in the EiffelStudio environment, you may define a universe by specifying (through the control language Lace, or through the graphical interface) a set of *directories* (folders), each defining a **cluster**. The cluster is a set of classes; by default, any file in that directory with a name ending with **.e** — for example, *your_class.e* —, called a **class file**, is expected to contain an Eiffel class. The class texts contained in the class files of the specified clusters then make up the universe.

Each class is given by its name:



Class names

Class_name \triangleq **Identifier**

This is indeed a unique identification within a universe:



Class Name rule

VSCN

It is valid for a universe to include a class if and only if no other class of the universe has the same upper name.

Eiffel expressly does not include a notion of “namespace” as present in some other languages. Experience with these mechanisms shows that they suffer from two limitations:

- They only push forward the problem of class name clashes, turning it into a problem of namespace clashes.
- Even more seriously, they tie a class to a particular context, making it impossible to reorganize (“*refactor*”) the software later without breaking existing code, and hence defeating some of the principal benefits of object technology and modern software engineering.

Name clashes, in the current Eiffel view, should be handled by *tools* of the development environment, enabling application writers to combine classes from many different sources, some possibly with clashing names, and resolving these clashes automatically (with the possibility of registering user preferences and remembering them from one release of an acquired external set of classes to the next) while maintaining clarity, reusability and extensibility.

The preceding validity rule leads to the rule giving the *meaning* of a class name:



Class name semantics

A **Class_name** *C* appearing in the text of a class *D* denotes the class called *C* in the enclosing universe.

As usual, the semantic rule only makes sense if the validity rule holds.

A system will be drawn from a universe; to do this we need to designate a particular type and a particular procedure as “roots”:



System, root type name, root procedure name

A **system** is defined by the combination of:

- 1 • A universe.
- 2 • A type name, called the **root type name**.
- 3 • A feature name, called the **root procedure name**.

The names that you choose for root type and root procedure should correspond to suitable types and procedures in the system. To state this rule we need a notion of dependency between types:



Type dependency

A type T **depends** on a type R if any of the following holds:

- 1 • R is a parent of the base class C of T .
- 2 • T is a client of R .
- 3 • (Recursively) there is a type S such that T depends on S and S depends on R .

This states that C depends on A if it is connected to A directly or indirectly through some combination of the basic relations between types and classes — inheritance and client — studied later. Case 1 relies on the property that every type derives from a class, called its “base class”; for example a generically derived type such as $LIST [INTEGER]$ has base class $LIST$. Case 3 gives us indirect forms of dependency, derived from the other cases.

→ *Inheritance: chapter 6; client: chapter 7 (general definition on page 192); base class: chapter 11.*

This makes it possible to define what’s a proper choice of root type:



Root Type rule

VSRT

It is valid to designate a type TN as root type of a system of universe U if and only if it satisfies the following conditions:

- 1 • TN is the name of a stand-alone type T .
- 2 • T only involves classes in U .
- 3 • T 's base class is not deferred.
- 4 • The base class of any type on which T depends is in U .

These conditions make it possible to create the root object:

- A type is “*stand-alone*” if it only involves class names; this excludes “anchored” types (like *some_entity*) and formal generic parameters, which only mean something in the context of a particular class text. Clearly, if we want to use a type as root for a system, it must have an absolute meaning, independent of any specific context. “Stand-alone type” is defined at the end of the discussion of types.
 - A deferred class is not fully implemented, and so cannot have any direct instances. It wouldn’t work as base class here, since the very purpose of a root type is to be instantiated, as the first event of system execution. → “*DEFERRED FEATURES*”, 10.11, page 272
 - To be able to assemble the system, we must ensure that any class to which the root refers directly or indirectly is also part of the universe. → “*Types and classes involved in a type*”, page 343.
- In condition 2, a type T “involves” a class C if it is defined in terms of C , meaning that C is the base class of T or of any of its generic parameters: $U [V, X [Y, Z]]$ involves U, V, X, Y and Z . If T is a non-generic class used as a type, T “involves” only itself.

To complement the conditions on the root type we need one on the root procedure (the procedure that will start the system’s execution):



Root Procedure rule

VSRP

It is valid to specify a name pn as root procedure name for a system S if and only if it satisfies the following conditions:

- 1 • pn is the name of a creation procedure p of S ’s root type.
- 2 • p has no formal argument.
- 3 • p is precondition-free.

A routine is *precondition-free* (condition 3) if it has no precondition, or a precondition that evaluates to true. A routine can impose preconditions on its callers if these callers are other routines; but it makes no sense to impose a precondition on the external agent (person, hardware device, other program...) that triggers an entire system execution, since there is no way to ascertain that such an agent, beyond the system’s control, will observe the precondition. Hence the last condition of the rule.

Regarding condition 1, note that a non-deferred class that doesn’t explicitly list any creation procedures is understood to have a single one, procedure *default create*, which does nothing by default but may be redefined in any class to carry out specific initializations.

→ “*OMITTING THE CREATION PROCEDURE*”, 20.4, page 527.



Another condition on the root procedure is that it must be effective (non-deferred): a deferred procedure has no implementation, and hence cannot be used to start system execution. But we don't need such a condition in the Root Procedure rule, because it follows from the Root Class rule: if the root class contained a deferred procedure, it would itself have to be declared as deferred (as a result of a rule to be seen in a later chapter), and we have already precluded that through condition 3 of the Root Class rule.

→ "Class Header rule", page 126.

Thanks to the Root Type and Root Procedure rules we no longer have to talk about type and procedure *names*, but can directly refer to the root type, the root procedure and the root class of a system:

Root type, root procedure, root class

In a system *S* of root type name *TN* and root procedure name *pn*, the **root type** is the type of name *TN*, the **root class** is the base class of that root type, and the **root procedure** is the procedure of name *pn* in that class.

Any language processing tool used to assemble and execute systems must enable you to perform the following tasks:

- 1 • Selecting a root class.
- 2 • If the root class has two or more creation procedures, selecting one of them — the **root procedure** — for the system's execution.



Techniques to perform these selections fall beyond the scope of Eiffel proper, relying instead on tools of the environment. One possibility is to use **Lace** (Language for Assembling Classes in Eiffel), a simple Eiffel-like notation for specifying how to build and process a system. You may find a detailed description of Lace in an appendix.

→ Appendix B.

The root type and root procedure are needed to *execute* the system:



System execution

To **execute** a system on a machine means to cause the machine to apply a creation instruction to the system's root type.

If a routine is a creation procedure of a type used as root of a system, its execution will usually create other objects and call other features on these objects. In other words, the execution of any system is a chain of explosions — creations and calls — each one firing off the next, and the root procedure is the spark that detonates the first step.

Classes

4.1 OVERVIEW

Classes are the components used to build Eiffel software.

Classes serve two complementary purposes: they are the modular units of software decomposition; they also provide the basis for the type system of Eiffel.

This chapter explores the role of classes and the structure of class texts.

4.2 OBJECTS



Viewed as a type, a class describes (as noted in the previous chapter) the properties of a set of possible data structures, or **objects**, which may exist during the execution of a system that includes the class; these objects are called the **instances** of the class

→ The precise nature of objects is explained in chapter [19](#).

An object may represent a real-world thing such as a radio signal in cell phone software, a document in text processing software or an electron in physics software. It may also represent an immaterial concept from that world, such as a fabrication process in factory control software. Or it may be a pure artefact of computer programming, such as an abstract syntax tree in compilation software.

Classes corresponding to these examples might be:

- *SIGNAL*, whose instances represent signals transmitted by some device.
- *DOCUMENT*, whose instances represent documents.
- *ELECTRON*, whose instances represent electrons.
- *NODE*, whose instances represent nodes of syntax trees.

Every object that may exist during the execution of a system is an instance of some class of that system. This is an important property, since it means that the type system is simple and uniform, being entirely based on the notion of class.

→ Chapter [11](#) covers types.

More precisely, every object is a **direct instance** of only one class, called its **generating class**. It may, however, be an *instance* (direct or not) of many classes: all the ancestors (in the sense of inheritance) of its generating class.

→ For exact definitions: “instance” and “direct instance”, page 330; “Generating class”, page 506; “ancestor”, page 177.

Some classes, said to be **deferred**, have no direct instances; they provide incomplete object descriptions. If *C* is deferred, an instance of *C* is a direct instance of some effective (that is to say non-deferred) descendant of *C*.

→ See chapters 6 and 10 about inheritance and deferred routines.

4.3 FEATURES

Viewed as a module, a class introduces, through its class text, a set of **features**. Some features, called **attributes**, represent fields of the class’s direct instances; others, called **routines**, represent computations applicable to those instances.

→ Features are studied in detail in chapter 5.



Since there is no other modular facility than the class, building a software system in Eiffel means identifying the types of objects the system will manipulate, and writing a class for each of these types.

A system that includes a certain class will usually contain operations to create instances of that class (creation instructions and expressions, for a non-deferred class) and to apply features to those instances (feature calls).

→ Creation: chapter 20; calls: chapter 23.

4.4 USE OF CLASSES

In some cases, one of the two roles of classes is more important than the other.

- At one extreme, a class may be interesting only as a module encapsulating a number of routines. (It then resembles the “packages” of older programming languages.) Often, it will not then have any variable attributes. A system that uses such a class will not create any direct instances of it; instead, other classes of the system will make use of its features by inheriting from it, or through “non-object calls”.
- At the other end, you may want to introduce a class simply because you need to describe a new type of object, without necessarily thinking of its role in the system architecture, at least at first. (It then resembles the “records” or “structures” of older programming languages, although it will usually include routines along with attributes.)

Even though it has no direct instances, such a class will have instances — the direct instances of proper descendants.

→ “NON-OBJECT CALLS”, 23.9, page 629.

Both of these uses of classes arise in practice and both are legitimate.

In most cases, however, classes live up to their reputation, making a name for themselves in both the module and type worlds.

4.5 THE CURRENT CLASS

Current class

The **current class** of a construct specimen is the class in which it appears.

Every Eiffel software element — feature, expression, instruction, ... — indeed appears in a class, justifying this definition. Most language properties refer directly or indirectly, through this notion, to the class in which an element belongs.

This will be complemented by the notion of “current type”, which includes the formal generic parameters. → “*Current type*”, page 365

4.6 CLASS TEXT STRUCTURE

A class text contains the class name and a number of parts, all optional except for **Class_header**, and all except **Formal_generics** introduced by a keyword:

- **Notes**, beginning with **note**.
- **Class_header**, beginning with one of: **class**; **deferred class**; **expanded class**; **separate class**.
- **Formal_generics**, beginning with a bracket [.
- **Obsolete**, beginning with **obsolete**.
- **Inheritance**, beginning with **inherit**.
- **Creators**, beginning with **create**.
- **Converters**, beginning with **converter**.
- **Features**, made of one or more **Feature_clause** each beginning with **feature**.
- **Invariant**, beginning with **invariant**.
- **Notes** again, for more specific index properties if desired.

Here is an extract from a class describing hash tables, which illustrates all clauses except **Obsolete**:

This class is a simplified form of one in the Eiffel-Base library. A “hash table” is a table used to record a number of elements, each identified by an individual key.



note

description: "Hash tables used to store items associated % with hashable keys."

names: *h_table*, *dictionary*

access: *key*, *direct*

representation: *array*

size: *resizable*

```

class HASH_TABLE [G, KEY → HASHABLE] inherit
  TABLE [G, KEY]
  redefine
    load
  end

create
  make, from_tree
convert
  from_tree ({BINARY_SEARCH_TREE})
feature -- Initialization
  make (n: INTEGER)
    -- Allocate space for n items.
    ... Procedure body omitted ...
  load ... Rest of procedure omitted
feature -- Access
  control: INTEGER
  Max_control: INTEGER is 5
feature -- Status report
  ok: BOOLEAN
    -- Was last operation successful?
  do
    Result := (control = 0)
  end
  ... Other features omitted...
feature -- Removal
  remove (k: KEY)
    -- Remove entry of key k.
  require
    valid_key: is_valid (k)
  do
    ... Procedure implementation omitted...
  ensure
    not has (k)
  end

invariant
  0 <= control; control <= Max_control
note
  date: "$Date: 1998/01/30 20:57:49 $"
  revision: "$Revision: 1.8 $"
  reviser: "Marcel Satchell, January 2000"
  changes: "Copy and equality semantics"
  original_author: "Eiffel Software, 1986"
end

```

This abbreviated example is a specimen of a `Class_declaration`, with the following general syntax:



Class declarations

```

Class_declaration ≙ [Notes]
                  Class_header
                  [Formal_generics]
                  [Obsolete]
                  [Inheritance]
                  [Creators]
                  [Converters]
                  [Features]
                  [Invariant]
                  [Notes]
                  end
  
```

The next section offers an informal overview of the various parts and their roles, using `HASH_TABLE` as illustration. Subsequent sections of this chapter will only cover in detail `Notes`, `Class_header`, `Formal_generics`, `Obsolete` and the closing `end`; describing the rest is the task of the following chapters.

→ *Inheritance is discussed in chapters 6, 10, and 16, Creators in chapter 20, Features in chapter 5, and Invariant in chapter 9.*

4.7 PARTS OF A CLASS TEXT

As noted, class `HASH_TABLE` includes all of the possible parts save for `Obsolete`. Let's examine them informally, in their order of appearance.



The first `Notes` part serves to associate note information with the class, to facilitate identification, archival and retrieval of the class based on properties not found elsewhere in its text. The `Notes` part is studied in detail in the next section. It is organized as a sequence of clauses, each containing an optional `Note` term, such as *description*, a colon, and one or more associated values. Examples include a short description of the scope of the class (*description* entry), or alternate names for the notion covered by the class. The `Note` terms and values are free, but this example uses some of the recommended ones, part of the style guidelines.

→ *Deferred classes: 10.11, page 272 and subsequent sections.*



The `Class_header` introduces the class name, here `HASH_TABLE`. Instead of just `class`, the class header could begin with `deferred class`, `expanded class` or `separate class`, making the class “deferred”, “expanded” or “separate”.

→ *Expanded: 11.9, page 335 and subsequent sections. Separate: chapter 33. Genericity: chapter 12.*

The **Formal_generics** part, if present, makes the class “**generic**”, which means it is parameterized by types. Here `HASH_TABLE` has two formal generic parameters: `G`, representing the type of the elements in a hash table; and `KEY`, representing the type of the keys which serve to retrieve these elements. To obtain a type from a generic class, you must provide types, called **actual generic parameters**. For example, you may declare an entity denoting a possible hash table as



```
ownership_record: HASH_TABLE [CAR, STRING]
```

using types `CAR` and `STRING` as actual generic parameters for `G` and `KEY`: the type `HASH_TABLE [CAR, STRING]` represents tables of cars retrievable through strings (perhaps the license plate numbers). A type obtained in this way is called a **generic derivation** of the base class, here `HASH_TABLE`. The entity `ownership_record` declared with this type may at run-time become attached to a table from which it is possible to retrieve cars from their associated strings.

The notation `KEY → HASHABLE` in class `HASH_TABLE` indicates that the second formal generic parameter, `KEY`, is “**constrained**” by the library class `HASHABLE`. This means that any corresponding actual generic parameter must be a descendant of `HASHABLE`; this is indeed the case with class `STRING`. The first formal generic parameter, `G`, is “**unconstrained**”, allowing any type to be used as the corresponding actual generic parameter.

→ On unconstrained and constrained generic derivations, see 12.3 and 12.6, starting on page 351.

The **Obsolete** part, if present, indicates that the class is an older version which should no longer be used except for compatibility with existing systems. For example, along with `HASH_TABLE`, a library may contain a class beginning with



```
class H_TABLE [G, KEY → HASHABLE] obsolete
    "Use HASH_TABLE, which relies on improved algorithms"
inherit
    ... Rest of class text omitted ...
```

The only effect of such a clause is that some language processing tools may produce a warning when they process such a class. The warning should reproduce the `String` listed after the **obsolete** keyword.

The **Inheritance** part, beginning with **inherit**, lists the parents of the class and any **feature adaptation** applied to the inherited features. `HASH_TABLE` has only one parent, `TABLE`; its **Feature_adaptation** part, beginning with **redefine**, simply indicates that the new class will provide a new version of the inherited procedure `load`. There is indeed a declaration of `load` in the class text.

→ Inheritance: chapter 6.

The **Creators** part, beginning with **create**, lists the procedures which clients may use to create direct instances of the class. Here there are two: *make* and *from_tree*. A client may create a direct instance of **HASH_TABLE** by executing a creation instruction (also using the keyword **create**) such as

→ *Creation: chapter 20.*



```
create ownership_record.make (80_000)
```

which will allocate a new table with room for eighty thousand items.

A **Converters** part lists some of the creation procedures as being also *conversion* procedures, allowing assignment from instances of other types. Here it specifies as creation procedure *from_tree*, taking a **BINARY_TREE** as argument; this permits, for *h* a hash table and *b* a binary tree, to abbreviate the creation instruction

→ *Conversion: chapter 15.*



```
create h.from_tree (b)
```

as just

```
h := b
```

The **Features** part introduces the features of the class. It is made of zero or more subparts, each called a **Feature_clause** and introduced by the keyword **feature**. There are two reasons for allowing more than one **Feature_clause**:

→ *Features: chapter 5.*

- It is part of the recommended style practice to group features into categories. This yields a good class structure, facilitating understanding and maintenance. The EiffelBase libraries define a number of feature clause headers, each with a standard header comment; they include the ones used in the example: **Initialization**, **Access**, **Status report**, **Removal**.
- Each may define an export status, making the corresponding features public, secret, or available to specific clients. In the absence of such a specification the default status is public availability.

→ *“GROUPING FEATURES”, 34.5, page 911.*

Here no **Feature_clause** departs from the default so that all the features shown — the procedures *make*, *remove* and *load*, the function *ok*, the variable attribute *control* and the constant attribute *Max_control* — are available to all clients. Calls from clients will use dot notation, as in



```
ownership_record.remove ("1745 BB 75")
--Assuming a Variable entity status of type INTEGER:
status := ownership_record.control
ownership_record.make (10_000)
```

→ *A feature is “exported” if it is available to all clients. See definition on page 211.*

The last of these calls applies to *make*, which is also a creation procedure but here is just used as a normal exported procedure. (Compare this call instruction with the **Creation** instruction above, using the keyword **create**.)

Of course, when deciding to export *make*, the designer of *HASH_TABLE* should make sure that calls occurring after the initial *Creation* instruction will have the proper effect; this probably means using a new size which is greater than or equal to the original one (in other words, keeping the original if the argument to the call is smaller), and writing the routine so that resizing does not lose any of the previously inserted elements.

To ensure that *make* is not available for outside calls, it would suffice to add a *Feature_clause* with an empty *Clients* list, beginning with *feature { }*, and move the declaration of *make* there. This is explained in detail in the chapters on features and exports. → “*Restricting exports*”, *page 201*. A full example appears in *5.5, page 13*.

The *Invariant part*, beginning with *invariant*, introduces consistency conditions on the features of the class; here the condition simply gives the bounds for attribute *control*. → “*CLASS INVARIANTS*”, *9.8, page 245*.

Finally you may have a new *Notes* clause, complementing the one at the beginning of the class, and introducing note information of a more specialized nature, such as copyright, revision history and author name.

After this general survey of the structure of a class text, the rest of this chapter examine five clauses which apply to the class as a whole: *Notes*, *Class_header*, *Formal_generics*, *Obsolete* and ending comment.

4.8 ANNOTATING A CLASS

Through a *Notes* entry you may include documentary information in the text of a class.

This is particularly important in the approach to software construction promoted by Eiffel, based on libraries of reusable classes: the designer of a class should help future users find out about the availability of classes fulfilling particular needs.

We may imagine the author of a class *DOCUMENT* writing the class text as follows:



note

description: "Documents of the most general form"
domains: text, text_processing, FrameMaker

class DOCUMENT inherit ... feature

...

note

author: "Tatiana Sergeevna Krasnojivotnaya"
approved_by: "Giovanni Giacomo della Gambagialla"
original: 21, March, 1999
last: 12, July, 2006

end

The general form is:



	Notes
Notes	\triangleq note Note_list
Note_list	\triangleq {Note_entry ";" ...}*
Note_entry	\triangleq Note_name Note_values
Note_name	\triangleq Identifier ":"
Note_values	\triangleq {Note_item ", ...}+
Note_item	\triangleq Identifier Manifest_constant



Notes parts (there may be up to two, one at the beginning and one at the end) have no effect on the execution semantics of the class. They serve to associate information with the class, for use in particular by tools for configuration management, documentation, cataloging, archival, and for retrieving classes based on their properties.

Each **Note_entry** starts with a **Note_name**, such as *author*., terminated by a colon. The rest of the **Note_entry** is a list of **Note_item** terms, each of which is an **Identifier** (such as *text_processing* or *July*) or a **Manifest_constant**, that is to say a value of a basic type, such as the integer *21*, or a string such as "*Tatiana Sergeevna Krasnojvotnaya*" etc.

→ **Manifest_constant** is introduced in [32.16](#), [page 899](#), and subsequent sections.

By the very nature of **Notes** parts, the choice of indices and values is free. Using consistent conventions will greatly facilitate the successful retrieval of reusable classes. Here you may wish to rely on the set of guidelines defined for the Eiffel Software Libraries.

→ "[GUIDELINES FOR ANNOTATING CLASSES](#)", [34.13](#), [page 921](#).

As illustrated by both the *HASH_TABLE* and *DOCUMENT* examples, a class may include up to two **Notes** clauses, one at the very beginning, before the keyword **class**, and one at the very end, before **end**. Their intended role is complementary:

- Use the initial **Notes** for critical information that you want every reader of the class to discover before reading anything else about the class, such as the *description* entry which succinctly explains the role of the class.
- Use the final **Notes** for archival and management information such as revision history, copyright and intellectual property notices, author and reviser names, and any supplementary information that will be useful to maintainers of the class.

The **Notes** parts of *HASH_TABLE*, shown earlier, illustrated these guidelines.

Notes semantics

A **Notes** part has no effect on system execution.

4.9 CLASS HEADER

The **Class_header** introduces the name of the class; it also serves to indicate whether the class is deferred or expanded. Here are two **Class_header** examples from EiffelBase and one from the Kernel Library, illustrating these possibilities:



```
class LINKED_LIST
deferred class SEQUENCE
expanded class INTEGER
```

The general form of the **Class_header** is simply:



Class headers

Class_header \triangleq [Header_mark] **class** Class_name
Header_mark \triangleq **deferred** | **expanded** | **frozen**

The **Class_name** part gives the name of the class. The recommended convention (here and in any context where a class text refers to a class name) is the upper name.

The upper name is the name written all in upper case..

The keyword **class** may optionally be preceded by one of the keywords **deferred**, **expanded** and **frozen**, corresponding to variants of the basic notion of class:

- A **deferred** class describes an incompletely implemented abstraction, which descendants will use as a basis for further refinement.
- Declaring a class as **expanded** indicates that entities declared of the corresponding type will denote objects rather than references to objects.
- Making a class **frozen** prohibits it from serving as “conforming parent” to other classes.

As the syntax specification indicates, these four options are exclusive. A class may not, for example, be both deferred and expanded; in fact, all non-expanded classes are considered to be reference classes.

This is part of a general characteristic of the syntax: unlike languages such as Ada, Java and C++, Eiffel does not use multiple successive keyword qualifiers. Where it allows you to write **property1** *x* or **property2** *x*, it does not permit **property1 property2** *x*. This keeps things simple and easy to remember.

The first two cases have an influence on the validity rule for **Class_header** and we now examine them in more detail.

Deferred classes



A class declared **deferred** describes an incompletely implemented abstraction, with the expectation that proper descendants of the class will provide or refine the implementation. This is useful to cover incompletely understood concepts or groups of related concepts. A typical example in EiffelBase is the deferred class **SEQUENCE**, which describes sequential data structures without prescribing any particular implementation. Proper descendants of this class, such as **LINKED_LIST**, describe concrete sequential structures. Such non-deferred classes are said to be **effective**.

→ For details see [“DEFERRED FEATURES”, 10.11, page 272.](#)

The deferred-effective distinction applies not just to classes but to their individual *features*: a feature is deferred if its class specifies it (often with a contract: precondition and postcondition) but does not provide an implementation. In general, a deferred class includes one or more deferred features. For example procedure *extend*, which adds an element at the end of a sequence, is deferred in **SEQUENCE** and **effected** (made effective) by **LINKED_LIST** and other effective descendants, each in its own way.

Deferred classes have no direct instances (you may not create an instance of the corresponding type, as in **create** *x* for *x* of type **SEQUENCE** [*T*]); only their effective descendants do, so that **create** *x* is valid for *x* of type **LINKED_LIST** [*T*].

A less drastic way of restricting clients' instantiation rights is through the **Creators** part.

→ [“RESTRICTING CREATION AVAILABILITY”, 20.7, page 539.](#)

The validity rule below requires that as soon as a class has at least one deferred feature you must declare it as class as **deferred class**. If not, the class would be considered effective; then clients could create instances, and call on them a feature that you haven't implemented.



There is no converse requirement: you may declare a class as **deferred** even if it has no deferred feature. This is a way of stating that you intend to use a class as an abstract concept even though you haven't included any deferred feature yet. In particular, you are prohibiting clients from creating direct instances through **create** *x* instructions.

Expanded classes

Declaring a class C as **expanded** changes the assignment and comparison semantics of the entities declared of the corresponding types. With $y: C$ (ignoring any generic parameters), and C not expanded, the assignment $x := y$ is a reference assignment, and the boolean expression $x = y$ compares references. But if C is expanded, the assignment copies the object denoted by y , and the test compares objects.

One application of this notion is to represent the notion of *sub-object*:



Sub-object vs. reference to another object

The figure shows an instance of a class with two attributes, one of a reference type and the other of an expanded type, representing a sub-object. The discussion of types will provide more details on the difference between expanded and reference semantics.

→ “[EXPANDED TYPES](#)”, 11.9, page 335.

To declare a class as expanded you must make sure that it retains *default_create* — the default initialization procedure coming, after possible renaming or redefinition, from the universal class *ANY* — as one of its creation procedures. The reason is that initializing an object with sub-objects, such as the one illustrated above, requires initializing all its sub-objects, for which all that’s available is the standard initialization.

→ “[OMITTING THE CREATION PROCEDURE](#)”, 20.4, page 527.

In the simplest case this requirement is automatically met: a class that doesn’t have a *Creators* part (that is to say, doesn’t explicitly list creation procedures) is considered to have *default_create* as its sole creation procedure. The details appear in the discussion of creation.

→ [20.4, page 527](#).

Validity of a class header

The validity rule on *Class_header* states the relationship between the actual class text and a declaration as **deferred**:



Class Header rule

VCCH

A *Class_header* appearing in the text of a class C is valid if and only if has either no deferred feature or a *Header_mark* of the **deferred** form.

If a class has at least one deferred feature, either introduced as deferred in the class itself, or inherited as deferred and not “effected” (redeclared in non-deferred form), then its declaration must start not just with **class** but with **deferred class**. *The definition of “deferred class” is on page 309.*

There is no particular rule on the other possible markers, **expanded** and **frozen**, for a **Class_header**. Expanded classes often make the procedure *default_create* available for creation, but this is not a requirement since the corresponding entities may be initialized in other ways; they follow the same rules as other “attached” entities.

The Class Header rule yields a simple definition:



Expanded, frozen, deferred, effective class

A class is:

- **Expanded** if its **Class_header** is of the **expanded** form.
- **Frozen** if its **Class_header** is of the **frozen** or **expanded** form.
- **Deferred** if its **Class_header** is of the **deferred** form.
- **Effective** if it is not deferred.

Making *C* **frozen** prohibits it from serving as “conforming parent” to other classes. The second case indicates the two ways of ensuring this:

- Inheritance from expanded classes, as explained in the discussion of inheritance, is non-conforming. As a consequence, any expanded class is also frozen.
- You can explicitly mark a non-expanded class as **frozen**.

The third case defines what makes a class *deferred*:

- If it has at least one deferred feature, the class itself is deferred. The Class Header rule below requires it to be marked **deferred** for clarity.
- If it only has effective features, the class is effective unless you decide otherwise: you can still explicitly mark it **deferred**. This ensures that it will have no direct instances, since one may not apply creation instructions or expressions to a variable whose type is based on a deferred class.

4.10 FORMAL GENERIC PARAMETERS

A class whose **Class_header** is followed by a **Formal_generics** part, as in

```
class HASH_TABLE [G, KEY -> HASHABLE]...
```

will be called a **generic class**. (If the **Formal_generics** part is absent, the class is, predictably, a **non-generic class**.) A generic class has one or more **formal generic parameters**, which are identifiers, here *G* and *KEY*, not conflicting with any name of a class in the surrounding universe. The mechanism that permits generic classes and the corresponding types is called **genericity**.

As noted, a generic class does not directly yield a type, although it is easy to derive a type from it: just provide a list of types, called **actual generic parameters**, one for each formal generic parameter. This was done above in the declaration of *ownership_record* to derive the type

```
HASH_TABLE [CAR, STRING]
```

from *HASH_TABLE*, with an **Actual_generics** list made of the types *CAR* and *STRING*. Such a type is said to be **generically derived**.

Genericity is the main reason classes and types are not identical notions: while any non-generic class is also a type, a generic class such as *HASH_TABLE* needs actual generic parameters to yield types such as the above. The notions of class and type are, of course, closely connected. More precisely, any type has a **base class** whose features provide the operations available on the type's instances; for a generically derived type such as the above, the base class is simply the type stripped of its actual generic parameters, here *HASH_TABLE*.

→ "**BASE CLASS, BASE TYPE AND TYPE SEMANTICS**", 11.7, page 332.

A whole chapter is devoted to genericity and will give the details. Here is a preview of the syntax of **Formal_generics** parts:

→ Chapter 12; see syntax and validity in "**GENERIC CLASSES**", 12.2, page 349.



```
Formal_generics Δ ["Formal_generic_list"]
Formal_generic_list Δ [Formal_generic ";" ...]
Formal_generic Δ [frozen] Formal_generic_name
                  [Constraint]
```



The **Constraint** construct, also detailed in the genericity chapter, governs *constrained genericity*, as in *C* [*G* → *CONSTRAINING_TYPE*], which specifies that *G* represents not arbitrary types, as in the basic (unconstrained) case, but types that conform to *CONSTRAINING_TYPE*.

→ "**CONSTRAINED GENERICITY**", 12.6, page 354.

4.11 OBSOLETE MARK

By specifying an **Obsolete** mark for a class, you indicate that the class does not meet your current standards, and you advise developers against continuing to use it as supplier or parent; but you avoid harming existing systems that may rely on this class.

The decision to make an entire class obsolete is not a frequent one in well-planned software development: through information hiding, uniform access, dynamic binding and genericity, the language often enables developers to change a class with little or no impact on clients and descendants. Even when some aspects of a class become obsolete, the class as a whole may remain appropriate; this is why you should usually prefer the related mechanism letting you make individual **features** obsolete. The next chapter explains how to do this, with further comments about software evolution and obsolescence.

→ “*OBSOLETE FEATURES*”, 5.21, page 165.

The decision to make a *class* obsolete is appropriate when you realize that even by obsoleting some of its features you won’t be able to bring it up to its ideal form without disturbing existing software, and decide to replace it by a new version. The civilized way to do this is to keep the old class, at least for a while, under its original name, but mark it obsolete; this signals to client and descendant developers that they will ultimately have to adapt their classes to the new version.

An **Obsolete** mark has no other effect; in particular it has no bearing on the software’s execution.

Here is the syntax of the mark, which comes after the **Class_header** and optional **Formal_generics**:



Obsolete marks

Obsolete \triangleq **obsolete** Message
 Message \triangleq Manifest_string



There is no validity constraint. The semantic specification covers both obsolete classes and obsolete features:

Obsolete semantics

Specifying an **Obsolete** mark for a class or feature has no run-time effect.

When encountering such a mark, language processing tools may issue a report, citing the obsolescence **Message** and advising software authors to replace the class or feature by a newer version.



Class obsolescence is not a way to cover up for bugs or flawed designs. If you realize that a class is incorrect or inadequate, you should face the consequences and repair the problem, even if this requires updating dependent classes. Any existing system using the flawed class cannot be functioning properly anyway. The **Obsolete** facility is meant for a different case: classes which were useful and sound, but cover needs for which you have now found improved solutions, based on a new design not backward-compatible with the original.

Features

5.1 OVERVIEW

A class is characterized by its features. Every feature describes an operation for accessing or modifying instances of the class.

A feature is either an *attribute*, describing information stored with each instance, or a *routine*, describing an algorithm. Clients of a class *C* may apply *C*'s features to instances of *C* through **call** instructions or expressions.

Every feature has an identifier, which identifies it uniquely in its class. In addition, a feature may have an *alias* to permit calls using operator or bracket syntax.

The following discussion introduces the various categories of feature, explains how to write feature declarations, and describes the form of feature names.

5.2 THE ROLE OF FEATURES

A feature of a class describes an operation which is applicable to the instances of the class. For example:



- A class *SIGNAL* might have such features as *amplitude* (amplitude of a signal) or *modulate* (modulate a signal with another).
- A class *DOCUMENT* might have such features as *character_count* or *print*.
- A class *ELECTRON* might have such features as *spin* or *valence*.
- A class *ABSTRACT_NODE* might have such features as *arity*, *is_leaf*, *is_root*, *add_child* or *remove_child*.

As these examples indicate, the operations represented by features may be of two kinds:

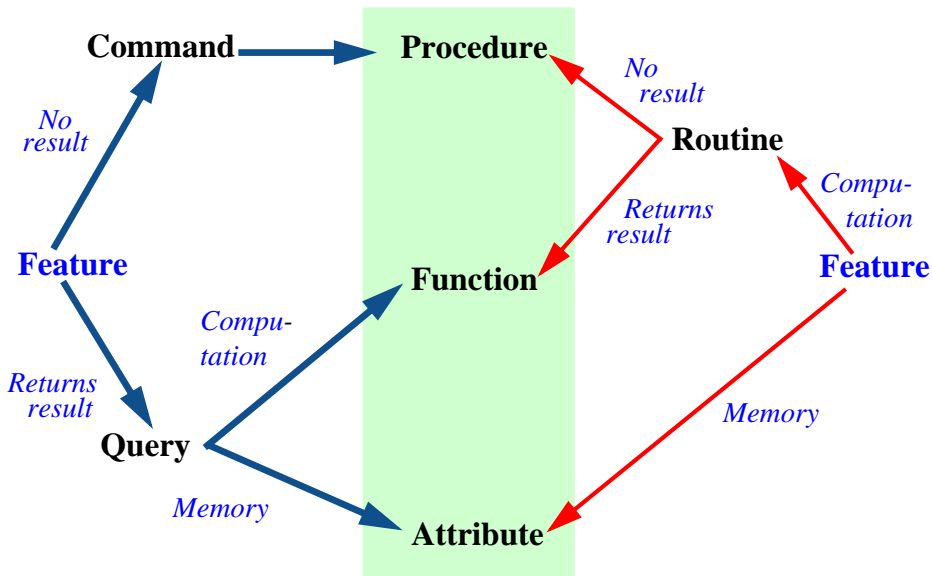
- Some are **query** operations, used to find out properties of objects (“What is the amplitude of this signal? How many characters does this document contain? Is this tree node a leaf?”).
- Others are **commands**, used to change objects or apply actions to them (“Print this document! Add a new child to this node!”).

A query will be implemented as an attribute or a routine. By nature, a command will always be a routine.

Queries are implemented as attributes or functions, commands as procedures.

5.3 FEATURE CATEGORIES

The following diagram shows the variants of the notion of feature and the associated terminology:



From the right, we have a classification based on the implementation of features:

- A feature implemented by storing information in every instance of the class (or, in the case of a constant, common to all instances) is an **attribute**.
- A feature implemented by an algorithm (a computation) applicable to all instances of the class is a **routine**. A routine that returns a result is a **function**; one that doesn't is a **procedure**.

From the left, we have a classification based initially on more abstract properties of features:

- A feature that does not return a result — but may modify its target object — is a **command**. Commands can only be implemented by *procedures* as just defined.
- A feature that provides a result — some information about its target object — is a **query**. A query may be implemented either by storing that information, giving an *attribute*, or by computing that information when requested, giving a *function*.

This book is precise and careful in its use of the terminology. Please make sure (possibly by reading this section once again) that you are familiar with the exact meanings of all the terms: *feature*, *command*, *query*, *routine*, *function* and *attribute*.



The word “method”, sometimes used in the object-oriented literature, may be viewed as a synonym for “routine”, i.e. a feature implemented by an algorithm rather than stored. Although this is a well-accepted term, it is redundant (there were already several words for this notion before O-O came about: routine, subroutine, subprogram...) and leads to confusion with the ordinary sense of the word “method”.

“Feature” is at a higher level, since it covers all categories. The closest word in the C++/UML/Java literature is “member”. Many presentations treat attributes as a data structure implementation mechanism, unrelated to routines; this loses the notion that at the highest level of abstraction we only have a notion of feature, with no commitment to any particular implementation choice. It’s OK to export an attribute (there is no need to encapsulate it in a function!) as long as, to the client, it appears only as a query, with an interface that doesn’t betray whether the query is implemented as an attribute or a function. This is Eiffel’s approach.

5.4 IMMEDIATE AND INHERITED FEATURES

The rest of this chapter will describe the properties of **Features** parts of a class, which introduces zero or more “features of the class”.



When thinking about features, we must be careful not to confuse two notions:

- The features **introduced in** a class.
- The features **of** that class.

The reason for this distinction is inheritance, which enables a class, in addition to the features declared in its own text, to obtain features declared in other classes — its parents.

The notion of parent is studied in chapters [6](#), [10](#) and [16](#).

Here is the precise terminology.



Inherited, immediate; origin; redeclaration; introduce

Any feature f of a class C is of one of the following two kinds:

- 1 • If C obtains f from one of its parents, f is an **inherited** feature of C . In this case any declaration of f in C (adapting the original properties of f for C) is a **redeclaration**.
- 2 • If a declaration appearing in C applies to a feature that is not inherited, the feature is said to be **immediate** in C . Then C is the **origin** (short for “class of origin”) of f , and is said to **introduce** f .

This defines the origin of immediate features only. The full definition, also covering inherited features, appears on page [311](#).

A feature redeclaration is a declaration that locally changes an inherited feature. The details of redeclaration appear in the study of inheritance; what is important here is that a declaration in the **Features** part only introduces a new feature (called “immediate” in **C**, or “introduced” by **C**) if it is not a redeclaration of some feature obtained from a parent.

→ *Redeclaration is studied in chapter 10, especially 10.28, starting on page 312.*

Every feature of a class is immediate either in the class or in one of its proper ancestors (parents, grandparents and so on).

The rest of this chapter only discusses immediate and redeclared features, by describing the **Features** part of a class declaration.

→ *Inherited features are studied in chapters 6, 10 and 16, with the full definition on page 470.*

5.5 FEATURES PART: EXAMPLE

A **Features** part is a sequence of one or more **Feature_clause**, as in the following sketch of a class from the EiffelBase Library:



```

note
    ... Notes clause omitted ...
class LINKED_LIST [T] inherit
    LIST [T]
        redefine
            first, start, return
        end
feature -- Access
    first: T
        -- Item at first position
    require
        not_empty: not empty
    do
        Result := first_element.item
    end
feature -- Measurement
    count: INTEGER
        -- Number of items in the list
    ... Other feature declarations and Feature_clause omitted ...
feature {LINKED_LIST} -- Implementation
    previous, next: like first_element
    merge_left (other: like Current)
        ... Rest of procedure omitted ...
    ...Other feature declarations omitted...

```

```

feature {NONE} -- Implementation
    first_element: LINKABLE [like first]
        -- First linkable element

    ...Other feature declarations omitted...

invariant
    empty = (first_element = Void)
    ...Other invariant clauses omitted...

end

```

A **Features** part contains one or more **Feature_clause**. Each **Feature_clause** is introduced by the keyword **feature**, which may be followed, as in the last two cases above, by a **Clients** subclause, which is a list of class names in braces, as in `{A, B, C, ...}`.

All the features of a **Feature_clause** have the same **export status**. If the beginning of the **Feature_clause** gives a list of clients in braces, the clause's features are available for calls to those clients and their proper descendants only; otherwise they are available to all clients. Here, for example:

→ Chapter 7 explains the details of the export policy and of Clients clauses.

- *first* and *count* are available for calls to all clients.
- *previous*, *next* and *merge_left* are available only to `LINKED_LIST` itself, when used as its own client.
- The remaining features are available only to `NONE`; this means that they are secret (accessible within class `LINKED_LIST` only, without use of dot notation).

For a class including many features, you may want to use more than one **Feature_clause** even for features which all have the same export status. This separates features into feature categories. In this case every **Feature_clause** should begin (after the keyword **feature** and the **Clients** list, if any) with a **Header_comment** indicating the feature category. Here, the comments indicating the various categories are



```

-- Access
-- Measurement
-- Cursor movement
-- Implementation

```

Because the inclusion of such a **Header_comment** is part of the recommended style, it appears as an optional component in the syntax for **Feature_clause** given below. Eiffel tools — such as documentation tools, or tools for archiving and retrieving classes — may treat it specially; for eon it for contents. In particular, it appears in the “contract view” serving as the basic documentation for a class.

→ “DOCUMENTING THE CLIENT INTER-FACE OF A CLASS”, 7.9, page 212 .

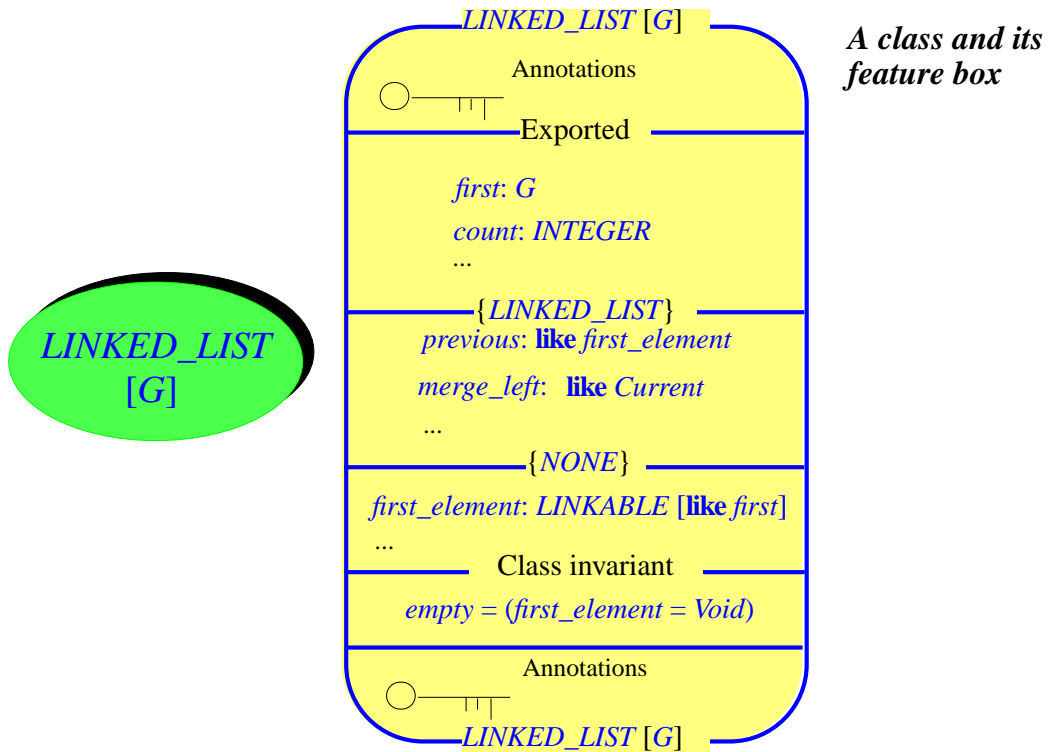


Although you may choose any text for header comments, the texts used here — [Access](#) and others — are among a dozen or so standard ones used, always in the same order, throughout classes of EiffelBase and other libraries. This yields a consistent style, greatly facilitating software understanding and maintenance. It's a good idea to use such a standard set of headers; start from the one in EiffelBase and extend it if necessary.

5.6 GRAPHICAL REPRESENTATION

In BON (Business Object Notation), the suggested graphical representation for classes and system structures, the features introduced in a class should appear next to the ellipse representing that class.

If enough display space is available and you want a full representation of the features, the format is that of a **feature box** appearing next to the class ellipse, and shown on the next figure for part of the class sketched in the previous section.



As in the textual form of the class, the features are grouped into successive divisions according to their export status.

Each feature includes type information as needed: argument types for routines; for a query (attribute or function), result type.

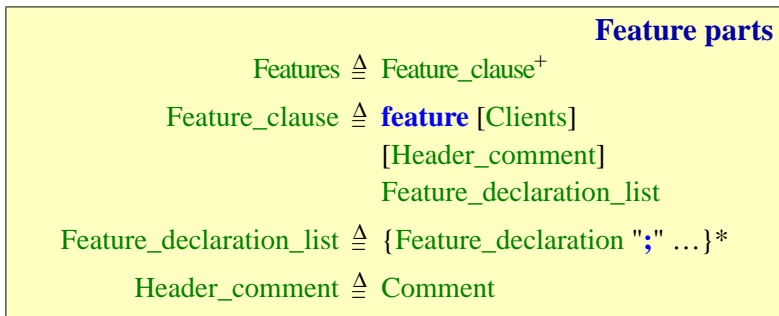
This graphical notation takes up a large amount of space and is mostly suitable for examining and designing classes in an interactive graphical environment, where you can see the various properties displayed on demand: the ellipses representing classes, the arrows representing the client and inheritance relations, the feature boxes. For printing on paper, or a whiteboard discussion, a more concise representation — frequently used in the present book — is appropriate:



An additional convention will be seen in the [discussion of attributes](#): if you know that a feature is an attribute, you may highlight this property by enclosing the feature's name in a rectangle. → [Page 499](#).

5.7 FEATURES PART: SYNTAX

Here is the precise format of the **Features** part of a class text, illustrated by the above example.



→ *The syntax for the Clients part appears on page 208.*

As part of a [general syntactical convention](#), semicolons are **optional** between a **Feature_declaration** and the next. The recommended [style rule](#) suggests omitting them except in the infrequent case of two successive declarations on a single line. ← *“Syntax (non-production): Semicolon Optionality rule”, page 103*

The rest of this chapter concentrates on the **Feature_declaration** construct, explaining what kinds of feature a class may declare.

5.8 FORMS OF FEATURE

Feature categories: overview

Every feature of a class is either an *attribute* or a *routine*.

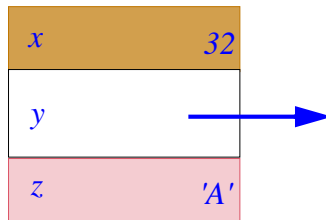
An attribute is either *constant* or *variable*.

A routine is either a *procedure* or a *function*.

A set of definitions in the discussion that follows introduces each of these notions precisely, making it possible to recognize, from a valid feature declaration, which kind of feature it introduces.

By introducing an *attribute* in a class, you specify that at run-time every instance of the class will possess a certain value, or field, corresponding to the attribute.

So you may picture any instance of the class as an object made of a number of fields, each giving the value defined by the object for one of the attributes of the class. The figure illustrates a direct instance of a class *C* with three attributes, *x*, *y* and *z*. (To picture a non-direct instance, we would also need to consider attributes introduced in proper descendants.)



A class instance with its fields

x is of type *INTEGER*, *z* of type *CHARACTER*, and *y* of some reference type. The field for *y* is attached to an object, which the figure does not show.

An attribute is either **variable** or **constant**:

- If an attribute is variable, the corresponding field may be different for various instances of the class and may change at run-time. As a consequence, the actual values must be stored for each instance.
- If an attribute is constant, the corresponding field is the same value for all instances, and may not change at run-time. This value appears in the class as part of the attribute declaration.

By introducing a *routine* in a class, you specify that a certain computation (an algorithm) must be applicable to every instance of the class. A routine, as we have seen, is either a **procedure** or a **function**:

- A procedure does not return a result; it may perform a number of operations, which may modify the instance.
- A function returns a result and may also perform operations.



A function should not change any object, except if the change only affects an object's representation, not its abstract properties. Because language processing tools cannot easily know which properties are abstract, the ban on object-modifying functions is a methodological guideline — the *Command-Query Separation principle* — and not a language rule.

See "*Object-Oriented Software Construction*".

5.9 FEATURE DECLARATIONS: EXAMPLES

To help you become familiar with the syntax of a **Feature_declaration**, here are a few artificial examples illustrating the various possibilities. The next sections give the precise syntax and detailed comments; for the most part, however, the examples should suffice as a guide for declaring features. The name of each example feature (such as *function_without_arguments*) suggests its nature.



```

variable_attribute: INTEGER
    -- Some field of integer type

other_variable_attribute: SOME_TYPE
    -- Some other field, of another type

Constant_attribute: REAL = 3.141592
    -- A constant real value used by the class

procedure (argument1: INTEGER; argument2: SOME_TYPE)
    -- (Here should come the description
    -- of the procedure's intended effect.)
do
    some_attribute.some_procedure
    other_attribute.other_procedure
end

deferred_procedure (argument1: SOME_TYPE )
    -- (Here should come the description
    -- of the procedure's intended effect.)
deferred
end

function_with_arguments (arg1, arg2: SOME_TYPE): OTHER_TYPE
    -- (Here should come the description
    -- of the result computed by the function.)
do
    create Result
    Result.some_procedure (arg2)
end

```

```

function_without_arguments: INTEGER
    -- (Here should come the description
    -- of the result computed by the function.)
do
    Result := some_value
end

plus alias "+" (some_matrix: like Current): like Current
    -- Matrix sum of Current and some_matrix
do
    ...(Computation of the sum into Result)...
end

attribute_with_contract: SOME_TYPE
    -- (Here should come the description of its role.)
require
    some_property
attribute
ensure
    other_property
end

self_initializing_attribute: SOME_TYPE
    -- (Here should come the description of its role.)
attribute
    initialization_instructions
end

```

5.10 FEATURE DECLARATIONS: SYNTAX

With the preceding examples in mind, we can now look at the exact ingredients that make up a `Feature_declaration_list`.

Such a list introduces immediate features of a class. It is a sequence of individual `Feature_declaration` clauses. In general each `Feature_declaration` introduces one feature, although it is possible to use a single declaration to introduce two or more "synonym" features. Each `Feature_declaration` includes the following pieces of information:

- The feature's original name (or names in the case of synonyms).
- The type of the feature, if it is an attribute or a function.
- The formal arguments, if the feature is a routine (procedure or function) with arguments.
- The actual value of the feature if it is a constant attribute.

- The contract and computation associated with the feature if applicable; a routine in particular must have an associated algorithm, but an attribute may also have a precondition and postcondition, as with *attribute_with_contract*, and a self-initialization algorithm, as with *self_initializing_attribute*.
- Possibly an Obsolete clause for a routine whose use is no longer recommended.
- Possibly the keyword **frozen**, appearing before the feature name to express that the declaration is final (not subject to redefinition in descendants).

→ “*OBSOLETE FEATURES*”, 5.21, page 165.

→ More on frozen declarations in discussion of the Feature Declaration rule page 162 below (condition 4), and of the Redefine Subclause rule, page 307 (condition 2).

The precise syntax is:



Feature declarations	
Feature_declaration	\triangleq New_feature_list Declaration_body
Declaration_body	\triangleq [Formal_arguments] [Query_mark] [Feature_value]
Query_mark	\triangleq Type_mark [Assigner_mark]
Type_mark	\triangleq ":" Type
Feature_value	\triangleq [Explicit_value] [Obsolete] [Header_comment] [Attribute_or_routine]
Explicit_value	\triangleq "=" Manifest_constant



Not all combinations of **Formal_arguments**, **Query_mark** and **Feature_value** are possible; the **Feature Body** rule and **Feature Declaration** rule will give the exact constraints. For example it appears from the above syntax that both a **Declaration_body** and a **Feature_value** can be empty, since their right-side components are all optional, but the validity constraints rule this out.

→ Pages 144 and 162.

The above examples illustrate some of the most important valid combinations.

What appears before the **Declaration_body** is not just a feature name but a **New_feature_list**, with the syntax

→ The form of an *Extended_feature_name* and the rules on multiple feature declarations appear later in this chapter (5.18).



New feature lists	
New_feature_list	\triangleq {New_feature ";" ... } ⁺
New_feature	\triangleq [frozen] Extended_feature_name

where an `Extended_feature_name` is a feature identifier possibly complemented by an `Alias` (for operator features).

Having a list of features, rather than just one, makes it possible for example to declare together several attributes of the same type or, in the case of routines, to introduce several “synonym” routines, with the same body.

A `Formal_arguments` part, possible only for a routine, describes the arguments to a routine and their types. An example is

→ *The syntax of Formal_arguments appears in 8.3, page 219.*



```
(arg1, arg2: TYPE1; arg3: TYPE2; arg4, arg5, arg6: TYPE3)
```

A `Query_mark` is present to mark that the feature is a query (attribute or function). It has a `Type_mark` specifying the type of the information returned by the query: for an attribute that’s the type of the field in instances of the class, for a function, it’s the type of the result computed by an execution of the function.. Examples of `Type_mark` are



```
: INTEGER
: SOME_TYPE
```

A `Query_mark` may also include an optional `Assigner_mark`. This lets you associate with the query a command of the class (a procedure), which can then be used to change the value of the query for the target object. A typical `Assigner_mark` is:



```
assign put
```

This may appear in a declaration of a function `item: T` for some type `T`, as in



```
item: T assign put ...
```

where `put` is a procedure of the same class, taking an argument of type `T`. This allows clients to use assignment syntax, `x.item := a` (for `a` of type `T`), as an abbreviation for the feature call `x.put(a)`. The mechanism also works for queries with arguments, as in `your_array.item(i) := 5` for a feature `item` taking an integer argument, as it does in class `ARRAY[G]`, where `item` has an integer argument; the associated assigner procedure correspondingly takes two two arguments, of types `G` and `INTEGER`. (Thanks to bracket syntax, you may also write this last example as `your_array[i] := 5`.)

→ *“BRACKET FEATURE”, 5.17, page 158.*

The procedure which an `Assigner_mark` associates with a query, such as `put` in these examples, is called an *assigner procedure*. The assignment-like instructions which this makes possible, such as `x.item := a` — with assignment-like syntax but the semantics of a call — is an *assigner call*.

→ *“ASSIGNER PROCEDURES”, 5.16, page 155.*

The `Feature_value` part, if present, gives the “value” of the feature, required in two cases:

- For a constant attribute, it introduces a literal value (integer, string etc.) with by an “equal: sign, as in *One*: *INTEGER = 1*.
- For a routine, it introduces the routine text.

For an attribute you can use a full form similar to that of routines, as in *x*: *A ... attribute ... end*, but for the most common case there’s an abbreviated form of the declaration: just *x*: *A*.

In the above example, the *Feature_value* for *constant_attribute* defines the constant’s value to be the real number 3.141592; the *Feature_value* for *procedure* is



```

-- (Here should come the description
-- of the procedure’s intended effect.)
do
  some_attribute.some_procedure
  other_attribute.other_procedure
end

```

A *Feature_value* may, according to the syntax, introduce some or all of the following components (the validity rules define which combinations are possible):

- An *Explicit_value* to specify the value of a *constant attribute*.
- An *Obsolete* mark to signal that the feature is *obsolete*.
- A *Header_comment* to explain the purpose of the feature.
- An *Attribute or routine* part to give the detailed specification of a routine or attribute, with clauses such as a precondition, a postcondition or, for a routine, the associated algorithm, as detailed next.

→ See [29.2, page 787](#) and subsequent sections about constants.

→ “[OBSOLETE FEATURES](#)”, 5.21, page 165.

5.11 FEATURE BODIES

Here indeed is the syntax of *Attribute_or_routine*:



Feature bodies

```

Attribute_or_routine ≙ [Precondition]
                    [Local_declarations]
                    Feature_body
                    [Postcondition]
                    [Rescue]
                    end
Feature_body ≙ Deferred | Effective_routine | Attribute

```

Subsequent chapters detail various elements of an [Attribute_or_routine](#):

- A [Precondition](#) and [Postcondition](#) express the contract of a feature. → [Chapter 9](#).
- [Local_declarations](#) introduce local variables needed by the feature's algorithm if any. → "[LOCAL VARIABLES AND RESULT](#)", [8.6, page 225](#).
- A [Feature_body](#) gives details of its implementation as an [Effective_routine](#) with an associated algorithm, or an attribute, or states that it is deferred routine, implemented only in proper descendants. → [Chapter 8](#).
- A [Rescue](#) clause takes over if a run-time exception arises during the execution of the feature. → [Chapter 26](#).

Only some combinations of these various clauses are meaningful. It is convenient to state the corresponding validity rule at the level of a [Feature_value](#) as a whole rather than just [Attribute_or_routine](#):



Feature Body rule

VFFB

A [Feature_value](#) is valid if and only if it satisfies one of the following conditions:

- 1 • It has an [Explicit_value](#) and no [Attribute_or_routine](#).
- 2 • It has an [Attribute_or_routine](#) with a [Feature_body](#) of the [Attribute](#) kind.
- 3 • It has no [Explicit_value](#) and has an [Attribute_or_routine](#) with a [Feature_body](#) of the [Effective_routine](#) kind, itself of the [Internal](#) kind (beginning with **do** or **once**).
- 4 • It has no [Explicit_value](#) and has an [Attribute_or_routine](#) with neither a [Local_declarations](#) nor a [Rescue](#) part, and a [Feature_body](#) that is either [Deferred](#) or an [Effective_routine](#) of the [External](#) kind.

The variants of [Feature_body](#) appear on page [222](#) as part of the discussion of routines.

The [Explicit_value](#) only makes sense for an attribute — either declared explicitly with [Attribute](#) or simply given a type and a value — so cases [3](#) and [4](#) exclude this possibility.

The [Local_declarations](#) and [Rescue](#) parts only make sense (case [4](#)) for a feature with an associated algorithm present in the class text itself; this means a routine that is neither deferred nor external, or an attribute with explicit initialization.

In both cases 1 and 2 the feature will be an attribute. Case 1 is a constant attribute declaration such as *n: INTEGER = 100*, with no further details. Case 2 is the long form, explicitly using the keyword **attribute** and making it possible, as with routines, to have a **Precondition**, a **Postcondition**, and even an implementation (including a **Rescue** clause if desired) which will be used, for “self-initializing” types, on first use of an uninitialized field.

The Feature Body rule is the basic validity condition on feature declarations. But for a full view of the constraints we must take into account a set of definitions appearing next, which say what it takes for a feature declaration — already satisfying the Feature Body rule — to belong to one of the relevant categories: *variable attribute*, *constant attribute*, *function*, *procedure*. Another fundamental constraint, the Feature Declaration rule (**VFFD**), will then require that the feature described by any declaration match one of these categories. So the definitions below are a little more than definitions: they collectively yield a validity requirement complementing the Feature Body rule.

→ Page 162.

5.12 HOW TO RECOGNIZE FEATURES

The precise form and properties of attributes and routines, as described by the syntax given above for **Feature_declaration**, are studied in later chapters. You should, however, learn right away how to recognize attributes (constant or variable) and routines (procedures or functions). This is not difficult and the above examples illustrate the most common cases. First, variable attributes:



Variable attribute

A **Feature_declaration** is a **variable attribute** declaration if and only if it satisfies the following conditions:

- 1 • There is no **Formal_arguments** part.
- 2 • There is a **Query_mark** part.
- 3 • There is no **Explicit_value** part.
- 4 • If there is a **Feature_value** part, it has an **Attribute_or_routine** with a **Feature_body** of the **Attribute** kind.

The first two features in the earlier **example**, *variable_attribute* and *other_variable_attribute*, were in this category. Here is an extract from a **Feature_clause** with two declarations introducing three variable attributes:

← Page 139.



```
count, capacity: INTEGER
backup: LINKED_LIST [INVESTMENT]
```

These examples use the abbreviated form without a keyword. You would obtain the same semantics by specifying **attribute** explicitly:



```
count, capacity: INTEGER
```

```
attribute  
end
```

And similarly for *backup*.

The keyword is required if you need to include a precondition, a postcondition, or a for self-initialization algorithm, as in *attribute_with_contracts* and *self_initializing_attribute* above, or



```
bounding_rectangle: RECTANGLE
```

```
-- Smallest rectangle including this figure
```

```
require
```

```
non_empty: not empty
```

```
attribute
```

```
create Result.make (lower, leftmost, height, width)
```

```
ensure
```

```
Result.lower = lower ; Result.higher = higher
```

```
Result.leftmost = leftmost ; Result.rightmost = rightmost
```

```
Result.height = height ; Result.width = width
```

```
end
```

The next kind of feature is the constant attribute:



Constant attribute

A *Feature_declaration* is a **constant attribute** declaration if and only if it satisfies the following conditions:

- 1 • It has no *Formal_arguments* part.
- 2 • It has a *Query_mark* part.
- 3 • There is a *Feature_value* part including an *Explicit_value*.

Two examples, introducing an *Integer_constant* and a *Manifest_string* are:



```
maximum_discount: INTEGER = 25
```

```
message: STRING = "No such site"
```

Even though the value is known from the declaration, it may still be useful to associate a contract (precondition, postcondition or both) to emphasize its more fundamental properties, which presumably would survive a change of the value in a revision of the software:



```
message: STRING = "No such site"
```

```
ensure
```

```
Result /= Void
```

```
Result.count <= Max_message_length
```

Finally, the case of a routine, with two variants:



Routine, function, procedure

A `Feature_declaration` is a **routine** declaration if and only if it satisfies the following condition:

- There is a `Feature_value` including an `Attribute_or_routine`, whose `Feature_body` is of the `Deferred` or `Effective_routine` kind.

If a `Query_mark` is present, the routine is a **function**; otherwise it is a **procedure**.

For a routine the `Formal_arguments` (like the `Query_mark`) may or may not be present.

By convention this definition treats a deferred feature as a routine, even though its effectings in proper descendants may be, in the case of a query, attributes as well as functions.

In the example illustrating the various categories of feature, the features with the following names are routines: ← Page 139 on.



```
procedure
```

```
deferred_procedure
```

```
function_with_arguments
```

```
function_without_arguments
```

```
plus alias "+"
```

`plus`, with its infix alias "+", is a function; the others are procedures or functions as indicated by their names.



Why do we need such rules for recognizing various kinds of feature? To put it more critically, why doesn't the language distinguish more clearly between them — for example by requiring specific keywords such as **procedure** at the beginning of each declaration?

The reason is methodological. As seen by clients, a feature is an abstract property of the instances of the class. Its particular choice of implementation within the class is a subordinate concern. As a consequence, the syntax downplays the differences between these forms of features instead of emphasizing them.

More precisely, what matters for clients is whether a feature returns a result, or just affects the state of objects without directly producing a result. This distinction is reflected in the following notions:



Command, query

A **command** is a procedure. A **query** is an attribute or function.

These notions underlie two important principles of the Eiffel method:

- The Command-Query separation principle, which suggests that queries should not change objects.
- The Uniform Access principle, which enjoins, whenever possible, to make no distinction between attributes and argumentless functions.

Of course, it is sometimes necessary to check what category a feature really belongs to. As the above examples indicate, you should quickly become familiar with the various forms.

See the discussion of these principles in "Object-Oriented Software Construction" and "UNIFORM ACCESS", 23.4, page 624 in the present book. These ideas also lead to the notion of contract view, studied in 7.9, page 212.

5.13 THE SIGNATURE OF A FEATURE



As defined above, a query feature — attribute or function — has a result type. But for any feature, query or command, we often need a more complete characterization of the feature's type properties, involving both the type of its result (in the query case) and the number and type of its arguments if any.

The notion of **signature** provides this. The signature of a feature is made of two lists of types:

- The list of argument types (empty for an attribute, or a routine without arguments).
- The list of result types (empty for a procedure).

To represent lists of types, we can borrow Eiffel's tuple notation, as in [\[TYPE1, TYPE2, TYPE3\]](#). (Such a list is not an Eiffel component, simply a notation to talk about properties of Eiffel features.) So a function whose declaration begins with

Chapter 13 discusses tuples.

```
f(a: INTEGER; b: X): LINKED_LIST [STOCK] is...
```

has the signature

```
[INTEGER, X], [LINKED_LIST [STOCK]]
```

We do not really need a sequence for the second component of a signature, since it will have zero or one element (zero if for a procedure, one for a query). Using a list on both sides provides more symmetry and generality. (The discussion of tuples will illustrate that symmetry.)

→ "Emulating multiple results", page 379

Here are the signatures of the features in the example at the beginning of this chapter. An empty sequence is shown as [].



<i>variable_attribute</i> :	[], [INTEGER]
<i>other_variable_attribute</i> :	[], [SOME_TYPE]
<i>constant_attribute</i> :	[], [REAL]
<i>procedure</i> :	[INTEGER, SOME_TYPE], []
<i>deferred_procedure</i> :	[SOME_TYPE], []
<i>function_with_arguments</i> :	[SOME_TYPE, SOME_TYPE], [OTHER_TYPE]
<i>function_without_arguments</i> :	[], [INTEGER]
plus alias "+":	[like Current], [like Current]
<i>attribute_with_contract</i> :	[], [SOME_TYPE]
<i>self_initializing_attribute</i> :	[], [SOME_TYPE]



The signatures of *variable_attribute* and *function_without_arguments* are identical, even though one is an attribute and the other a function. For clients, as noted, the difference won't be visible.

See above, page 147, and
"UNIFORMACCESS",
23.4, page 624

The notion of signature deserves a precise definition:



Signature, argument signature of a feature

The **signature** of a feature *f* is a pair *argument_types*, *result_type* where *argument_types* and *result_type* are the following sequences of types:

- For *argument_types*: if *f* is a routine, the possibly empty sequence of its formal argument types, in the order of the arguments; if *f* is an attribute, an empty sequence.
- For *result_type*: if *f* is a query, a one-element sequence, whose element is the type of *f*; if *f* is a procedure, an empty sequence.

The *argument_types* part is the feature's **argument signature**.

The argument signature is an empty sequence for attributes and for routines without arguments.

In the above examples, the argument signature of *variable_attribute* is [] (empty sequence); the argument signature of *procedure* is [INTEGER, SOME_TYPE].

5.14 FEATURE NAME

Feature names serve to identify features.

A feature name always involves an identifier; this means that it is always possible to write a valid call in ordinary object-oriented dot notation, as in

→ As detailed in the chapter on calls: chapter [23](#).

$x.f(a)$	-- Qualified: from a client, applied to x
$f(a)$	-- Unqualified: from within the class, -- applied to the current object



For some features syntactic variants are available, but the availability of these basic forms is guaranteed:

Feature principle

Every feature has an associated identifier.

Any valid call (qualified or unqualified) to the feature can be expressed through this identifier.

The syntactic variants, available through **alias** clauses, offer other ways to express calls, reconciling object-oriented structure with earlier notations:

- You may qualify the name with **alias** " \S " where \S is some operator. For example if a feature is named *plus*, clients must call it as *a.plus(b)*; by naming it *plus alias "+"* you still allow this form of calls — per the Feature principle — but you also permit *a + b* in accordance with traditional syntax for arithmetic expressions. The details of alias operators, as well as the associated conversion mechanism, appear next.
- You may also use a “bracket alias”, written simply **alias** "[" (with an opening bracket immediately followed by a closing bracket). This allows access through bracket syntax *x[index]*. For example if a class describing some table structure has a feature *item alias "[" (index: H): G* where *H* is some index type, items can be accessed through *your_table.item(i)* but also through the more concise *your_table[i]*. Again this is just a syntactic facility: the second form is a synonym for the first, which remains available.

Such bracket aliases appear for example in the *ARRAY*, *LIST* and *HASH_TABLE* classes of EiffelBase, so that you can access an element of one of these structures as *t[i]*, where *i* is an integer in the first two cases and a key in the last one.

You may combine such **alias** clauses in the extended feature name with assigner procedures. For example the EiffelBase class *HASH_TABLE* Introduced in defines the access feature

```
item alias "[" (key: H): G assign put ...
```

where *put* is a procedure to insert an element with a given key. This means that you can not only use bracket syntax for accessing items, as in

```
your_element := your_hash_table [your_key]
```

(an abbreviation for *your_element := your_hash_table.item (your_key)*), but also in an assigner call

```
your_hash_table [your_key] := your_element
```

an abbreviation for *your_hash_table.put (your_element, your_key)*.

We have now seen all kinds of feature name. Here is the syntax:



Feature names	
Extended_feature_name	△ Feature_name [Alias]
Feature_name	△ Identifier
Alias	△ alias "" Alias_name "" [convert]
Alias_name	△ Operator Bracket
Bracket	△ "["

The optional **convert** mark, for an operator feature, supports mixed-type expressions causing a conversion of the target, as in the expression *your_integer + your_real*, which should use the “+” operation from *REAL*, not *INTEGER*, for compatibility with ordinary arithmetic practice. See the presentation of conversions.

→ “*MIXED-TYPE EXPRESSIONS: TARGET CONVERSION*”, 15.12, page 428.

For the record we must clarify the use of quotes and spaces in an **Alias**:

→ *Similar rules apply to signed constants, in “Syntax (non-production): Sign Syntax rule”, page 788, and, further in that chapter, to character and string constants.*

Syntax (non-production): Alias Syntax rule

The **Alias_name** of an **Alias** must immediately follow and precede the enclosing double quote symbols, with no intervening characters (in particular no breaks).

When appearing in such an **Alias_name**, the two-word operators **and then** and **or else** must be written with exactly one space (but no other characters) between the two words.

In general, breaks or comment lines may appear between components prescribed by a BNF-E production, making this rule necessary to complement the grammar: you must write **alias** "+", not **alias** " + ".

It is useful to give official names to the aliased forms:



Operator feature, bracket feature, identifier-only

A feature is an **operator feature** if its `Extended_feature_name` *fn* includes an **Operator** alias, a **bracket feature** if *fn* includes a **Bracket** alias. It is **identifier-only** if neither of these cases applies.

The most common case is identifier-only. The other two kinds provide convenient modes of expression (“*syntactic sugar*”) for some cases where a shorter form, compatible with traditional mathematical conventions, is desirable for **calling the feature**.

→ Calls are studied in chapter [23](#). See also chapter [28](#) about expressions.

When referring to feature names, some syntax rules use the `Extended_feature_name`, and some use the `Feature_name`, which is just the identifier, dropping the **Alias** if any. The criterion is simple: when a class text needs to refer to one of its own features, the `Feature_name` is sufficient since (from the Feature Identifier principle below) it uniquely identifies the feature. So the `Extended_feature_name` is used in only two cases: when you first introduce a feature, in a `Feature_declaration` as discussed above, and when you change its name for a descendant, in a `Rename` clause (for both inheritance and constrained genericity).

→ “**RENAMING**”, [6.9, page 183](#).

This also means that in descendants of its original class a feature will retain its **Alias**, if any, unless a descendant explicitly renames it to a name that may drop the **Alias**, or provide a new one. In particular, redeclaring a feature does not affect its **Alias**.

There are indeed three forms of call:

- For the standard case, identifier features, calls use **dot notation**, as in



```
a.variable_attribute
b.procedure (b, c)
a.plus (b)
your_array.item (some_index)
```

- Giving a feature an operator alias, such as **plus alias** "+", allows calls to take the form of ordinary arithmetic expressions, such as $a + b$, rather than the more “obviously O-O” but heavier **a.plus (b)**.
- A class may have one (and only one) feature with a bracket alias, such as **item alias** "[]". The purpose, for a class representing container data structures such as arrays or tables, is to let clients access the structures using the traditional syntax of array and function access, for example **your_array [x]** as a synonym for **your_array.item (some_index)**.

→ Studied in more detail in the next section.

→ Studied in more detail below.



Remember that the operator and bracket aliases are only there to allow a form of feature call with a syntax other than dot notation, conforming to widely accepted notations (operator expressions, bracket access for arrays). Per the Feature principle, every feature has a **Feature_name** (which you can use to call the feature, although most people find the operator or bracket form clearer when available). If we need to clarify, we talk of “the identifier” of the feature: ← Page 150.



Identifier of a feature name

The **Identifier** that starts a **Extended_feature_name** is called the **identifier of** that **Extended_feature_name** and, by extension, of the associated feature.

This notion is closely related to one of the language’s design principles:



Feature Identifier principle

Given a class *C* and an identifier *f*, *C* contains at most one feature of identifier *f*.

This principle reflects a critical property of object-oriented programming in general and Eiffel in particular: no “*overloading*” of feature names within a class. It is marked as “validity” but has no code of its own since it is just a consequence of other validity rules. ← Discussed at the end of this chapter: “NO IN-CLASS OVERLOADING”, 5.22, page 167.

Another general notion that we need to define for feature name is when two feature names or operators are “the same”. The definition ensures that we ignore letter case:

Same feature name, same operator, same alias

Two feature names are considered to be “**the same feature name**” if and only if their identifiers have identical lower names.

Two operators are “**the same operator**” if they have identical lower names.

An **Alias** in an **Extended_feature_name** is “**the same alias**” as another if and only if they satisfy the following conditions:

- They are either the same **Operator** or both **Bracket**.
- If either has a **convert** mark, so does the other.

← The lower name is (page 102) the name all in lower case.

So *my_name*, *MY_NAME* and *mY_nAMe* are considered to be the same feature name. The recommended style uses a name with an initial capital and the rest in lower case (as in *My_name*) for constant attributes, and the lower name, all in lower case (as in *my_name*) for all other features. If letters appear in operator feature names, letter case is also irrelevant when it comes to deciding which feature names are the same and which different.

This notion is useful in particular to enforce the rule that, in any class, there can be only one feature of a given name (no “overloading”), and to determine what constitutes a “name clash” under multiple inheritance. In such cases the language rules simply ignore letter case.

→ “*NO IN-CLASS OVERLOADING*”, 5.22, page 167.

5.15 OPERATOR FEATURES

Operator features — those declared with an *Alias* listing a binary or unary operator — allow class authors, as previewed above, to provide their clients with a form of call based on the time-honored conventions of arithmetic expressions, using infix and prefix operators.

A matrix class can use *plus alias* “+” as the name of an addition function, enabling users of this feature to write additions in the usual mathematical form



```
matrix1 + matrix2
```

rather than in dot notation, which in this case might come out as *matrix1.plus(matrix2)*.

Similarly, naming a negation function *negated alias* “-” allows calls written in the form *-matrix1* as well as *matrix1.negated*.

The following syntax shows that an operator is either a free operator (*Free_unary* or *Free_binary*) or a **standard operator**. The standard operators, listed explicitly below, use special symbols, except for boolean operators which, following tradition, use keywords, simple (as **and**) or double (as **and then**).



Operators

```
Operator ≙ Unary | Binary
  Unary ≙ not | "+" | "-" | Free_unary
  Binary ≙ "+" | "-" | "*" | "/" | "//" | "\" | "^" | ".." |
           "<" | ">" | "<=" | ">=" |
           and | or | xor | and then | or else | implies |
           Free_binary
```


Free operators enable developers to define their own operators with considerable latitude. This is particularly useful in scientific applications where it is common to define special notations, which Eiffel will render as prefix or infix operators. You may for example define operators such as `**`, `|-` (maybe as an infix alias for a *distance* function), or various forms of arrow such as `<->`, `-|>`, `=>`.

The rule, given formally in the lexical analysis chapter, lets you use the symbols appearing in standard operators and any others non-alphabetic symbols as long as the result does not create any ambiguity with standard operators, special symbols, and predefined operators used for equality and inequality (`=`, `/=`, `~`, `/~`).

→ “*OPERATORS*”,
32.13, page 892

To avoid spurious parentheses in the writing of expressions, each of the standard operators, **Unary** and **Binary**, has a precedence level, according to a table appearing in the discussion of expressions. All free operators have the same precedence, higher than for standard operators.

→ “*Operator precedence levels*”, page 768.



Remember that an operator is not by itself a feature name but only appears in the **alias** of an **Extended_feature_name**, which also lists an identifier. This means that you never *have* to use operator notation to call a feature, as in `matrix1 + matrix2`: dot notation, using the feature’s identifier as in `matrix1.plus (matrix2)`, is always available, with the same semantics.

5.16 ASSIGNER PROCEDURES

Bla Bla Bla



Assigner marks

Assigner_mark \triangleq assign Feature_name

In an assignment $x := v$ the target x must be a variable. If $item$ is an attribute of the type T of a , programmers used to other languages may be tempted to write an assignment such as $a.item := v$ to assign directly to the corresponding object field, but this is not permitted as it goes against all the rules of data abstraction and object technology. The normal mechanism is for the author of the base class of T to provide a “setter” command (procedure), say put , enabling the clients to use $a.put(v)$.

The class author may, for convenience, permit $a.item := v$ as a shorthand for this call $a.put(v)$, by specifying put as an **assigner command** associated with $item$. An instruction such as $a.item := v$ is not an assignment, but simply a different notation for a procedure call; it is known as an **assigner call**. This scheme, a notational simplification only, is also convenient for features that have a **Bracket** alias, allowing for example, with a an array, an assigner call $a[i] := v$ as shorthand for a call $a.put(v, i)$.

The mechanism is applicable not just to attributes but (in line with the Uniform Access principle) to all queries, including functions with arguments.

The following rule defines under what conditions you may, as author of a class, permit such assigner calls from your clients by associating an assigner command with a query.



Assigner Command rule

VFAC

An **Assigner_mark** appearing in the declaration of a **query** q with n arguments ($n \geq 0$) and listing a **Feature_name** fn , called the **assigner command** for q , is valid if and only if it satisfies the following conditions:

- 1 • fn is the identifier of a command c of the class.
- 2 • c has $n + 1$ arguments.
- 3 • The type of c 's first argument and the result type of q have the same deanchored form.
- 4 • For every i in $1..n$, the type of the $i+1$ -st argument of c and the type of the i -th argument of q have the same deanchored form.



The feature q can only be a query since, from the syntax of **Declaration_body**, an **Assigner_mark** can only appear as part of a **Query_mark**, whose presence makes the feature a query.

In cases [3](#) and [4](#), we require the types (more precisely their deanchored forms, obtained by replacing any anchored type such as **like** x by the type of the anchor x) to be identical, not just compatible (converting or conforming). To understand why, recall that the assignment $a.item := y$ is only a shorthand for a call $a.put(x)$ with, as a typical implementation:

```

item: T assign put do ... end
put (b: U) do ... item := b ... end

```

WARNING: not valid for different T and U ; see text.

Now assume that U is not identical to T but only compatible with it, and consider the procedure call

```
a.put (a.item)
```

or the equivalent assignment form

```
a.item := a.item
```

which are in principle useless — they reassign to a field its own value — but should certainly be permitted. They become invalid, however, because the source $a.item$ (actual argument of the call or right side of the assignment) is of type T , the target (the formal argument) of type U , and it's generally impossible for two different types to be each compatible with the other.

In the conformance case, two-way compatibility would mean that the base classes are proper descendants of each other, causing an inheritance cycle.

→ Prohibited by clause 1 of the Parent rule, page 178.

In the convertibility case, it would violate the Conversion Asymmetry principle.

→ Page 408.

It is in fact possible to have conformance one way and convertibility the other, but this case is not useful enough to justify a special rule.

This explains clause 3: the first argument of the assigner procedure must have *exactly* the same type as the result of the query (once both have been deanchored). Similar reasoning applied to other arguments, if any, leads to clause 4.

5.17 BRACKET FEATURE

Besides operator aliases, the syntax of [Alias](#) offers the [Bracket](#) variant, ← [Bracket](#), page [151](#). allowing you for example to declare, in a class `HASH_TABLE [G, H]` describing tables of elements of type `G` with keys of type `H`, a feature



```

item alias "[]" (key: K): G
    -- Item having the given key
require
    present: has (key)
do
    ... "Appropriate implementation" ...
ensure
    ...
end

```

This is a normal feature, here a function, distinguished only by a new form of call. Although you may still use the standard dot-notation form

```
your_table.item ("ABC")
```

the bracket alias allows another phrasing for exactly the same semantics:

```
your_table ["ABC"]
```

To avoid ambiguity, at most one feature of a class may have a bracket alias; it must be a function with at least one argument. These requirements appear in the general constraint on aliases: the [Alias validity rule](#).

→ Page [163](#).

If the function has more than one argument, the bracket notation will use commas, as in `matrix3 [i, j, k]`.

It is often convenient, as already noted, to use the assigner procedure mechanism in connection with a bracket alias. If the declaration of `item` reads



```

item alias "[]" (key: K): G assign put
    ... The rest as above ...

```

referring to a procedure `put` of the same class, with compatible signature:

```
put (value: G; key: K) ... end
```

then instead of

```
your_table.put (v, "ABC")
```

you may write, with identical semantics:

```
your_table ["ABC"] := v
```

5.18 SYNONYMS AND MULTIPLE DECLARATION

Because the first part of a `Feature_declaration` is a `New_feature_list`, not just one `Extended_feature_name`, each feature declaration may introduce more than one feature, as in ← Syntax on pages 141 and 141.

```

a, b, c : INTEGER           -- Attributes
f, g require ... do ... ensure ... end  -- Routines
```

Such features introduced together are known as synonyms:

DEFINITION

Synonym

A **synonym** of a feature of a class C is a feature with a different `Extended_feature_name` such that both names appear in the same `New_feature_list` of a `Feature_declaration` of C .

Synonym declarations should be viewed as an abbreviation, according to the following rule:

DEFINITION

Unfolded form of a possibly multiple declaration

The **unfolded form** of a `Feature_declaration` listing one or more feature names, as in:

$$f_1, f_2, \dots, f_n \text{ declaration_body} \quad (n \geq 1)$$

where each f_i is a `New_feature`, is the corresponding sequence of declarations naming only one feature each, and with identical declaration bodies, as in:

```

f1 declaration_body
f2 declaration_body
...
fn declaration_body
```

Thanks to the unfolded form, we may always assume, when studying the validity and semantics of feature declarations, that each declaration applies to only one feature name. This convention is used throughout the language description; to define both the validity and the semantics, it simply refers to the unfolded form, which may give several declarations even if they are all grouped in the class text.

→ “*Feature Declaration rule*”, page 162.



A multiple declaration introduces the feature names as synonyms. But the synonymy only applies to the enclosing class; there is no permanent binding between the corresponding features. Their only relationship is to have the same **Declaration_body** at the point of introduction.

This means in particular that a proper descendant of the class may **rename** or **redeclare** one without affecting the other.

→ *Renaming: chapter 6*
Redeclaration: chapter 10.

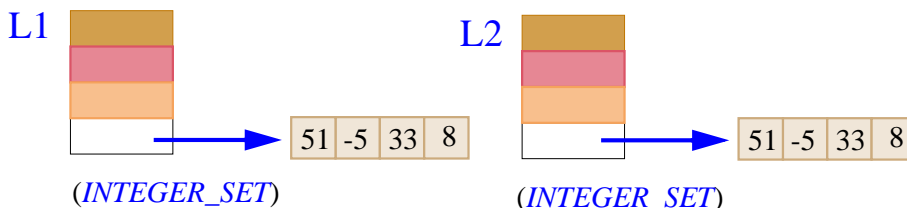
Each f_i , being a **New_feature**, may include a **frozen** mark. In the unfolded form this mark only applies to the i -th declaration.



When should we use multiple declarations? The last observations provide a clue. If you anticipate that a feature may have different variants in descendant classes, it may be better to introduce it as two features, initially identical, in its class of origin. This is in particular the case when you expect descendants to redefine the feature, but want to guarantee them access to the original — for themselves and, if appropriate, their clients. Then you should declare one of the two features as frozen.

ANY, the Kernel Library class serving as ancestor of all developer-defined classes, provides several examples of this technique. **ANY** offers a general comparison function, *is_equal*, originally comparing two objects for field-by-field equality. The semantics of the object comparison operator \sim is defined in terms of *is_equal*. Any class may redefine *is_equal* (and hence the meaning of \sim for operands of the corresponding types) to account for the specific semantics of equality desired for the class. For example, if objects **L1** and **L2** below are instances of a class **INTEGER_SET**, they are not field-by-field equal (since they contain references to different objects), but the author of **INTEGER_SET** may decide that *is_equal* and \sim must return true on these objects as they represent the same set. The class will redefine *is_equal* to test for the desired notion of equality.

See [21.6, page 580](#) about the respective roles of functions *is_equal* and *equal*.



Equivalent objects not field-by-field equal

Along with such redefinitions of *is_equal*, it is useful to keep the default version (performing field-by-field comparison) for all classes. This is why *ANY* introduces two equality functions, originally as synonyms:

is_equal, **frozen** *default_is_equal*
(*x*: like *Current*): **BOOLEAN** *is...*

with the consequence (enforced through the **frozen** mark) that the second function may not be redefined, so that developers can trust it to retain a fixed, universal semantics — indeed, a fixed implementation.



It is also important to understand when multiple declarations are **not** appropriate. This includes the following two situations:

- If you devise a better name for an existing feature, but wish to provide upward compatibility for existing clients and descendants, then a better mechanism, described below, is available: “obsoleting” the feature. This has the advantage of facilitating the eventual phasing out of the obsolete version, whereas there is no incentive to remove a synonym.
- The availability of a synonym mechanism is usually not a good excuse for refusing to choose between possible names. Class designers, especially designers of reusable library classes, should not be fickle; even if two sets of names appear equally good, it is generally better to choose just one than to provide both. By passing on the choice to client developers, the latter solution would only confuse them, and make the class appear more complex than it is.

"Reusable Software" discusses how to choose the names of library features; see also Appendix 34 of the present book.

These observations suggest that multiple declarations, although an important facility for cases such as the one mentioned above, should remain a relatively infrequent occurrence in normal Eiffel development.



The example also suggests what kinds of use are proper for frozen features. The very idea of “freezing” a feature is, in general, contrary to the fundamental Eiffel concepts of software extendibility and adaptability, which the feature adaptation mechanisms (in particular redeclaration) support directly. When you inherit from a class, you should be able to adapt its features to the new context; you may use the assertion mechanism to guarantee that the specification remains compatible with the framework defined in the original, although the implementation may be different.

This mechanism is so central in the Eiffel method that it leaves only a limited role for frozen features: taking care of predefined, system-level operations such as *is_identical*, for which we require not only the specification but the implementation to be determined once and for all.

5.19 VALIDITY OF FEATURE DECLARATIONS

To be valid, a [Feature declaration](#) must satisfy a constraint, known as the Feature Declaration rule. Here is the rule in full, followed by a detailed explanation of its clauses.

← The syntax of [Feature_declaration](#) was given on page [141](#)



Feature Declaration rule

VFFD

A [Feature_declaration](#) appearing in a class C is valid if and only if it satisfies all of the following conditions for every declaration of a feature f in its [unfolded form](#):

- 1 • The [Declaration_body](#) describes a feature which, according to the rules given [earlier](#), is one of: [variable attribute](#), [constant attribute](#), [procedure](#), [function](#).
- 2 • f does not have the [same feature name](#) as any other feature [introduced in \$C\$](#) (in particular, any other feature of the [unfolded form](#)).
- 3 • If f has the same feature name as the [final name](#) of any inherited feature, the [Declaration_body](#) satisfies the [Redeclaration rule](#).
- 4 • If the [Declaration_body](#) describes a [deferred feature](#), then the [Extended_feature_name](#) of f is not preceded by [frozen](#).
- 5 • If the [Declaration_body](#) describes a [once function](#), the result type is [stand-alone](#).
- 6 • Any [anchored type](#) for an argument is [detachable](#).
- 7 • The [Alias](#) clause, if present, is [alias-valid](#) for f .

← The rules for determining the kind of feature are those of [5.12](#).

→ The full Redeclaration rule is on page [313](#).

→ As defined on page [163](#) below.

Additional obligations apply if there is an [Assigner_mark](#); they are covered by the Assigner Procedure rule (automatically included thanks to the [General Validity rule](#)).

← Assigner Procedure rule: page [156](#); General Validity rule: page [98](#).

As stated at the beginning of the rule, the conditions apply to the [unfolded form](#) of the declaration; this [means](#) that the rule treats a multiple declaration f_1, f_2, \dots, f_n [declaration_body](#) as a succession of n separate declarations with different feature names but the same [declaration_body](#).

← “[Unfolded form of a possibly multiple declaration](#)”, page [159](#).

Conditions [1](#) and [2](#) are straightforward: the [Declaration_body](#) must make sense, and the name or names of the feature being introduced must not conflict with those of any other feature introduced in the class.

A redeclaration is either a **redefinition** of an inherited feature (changing its specification, signature or implementation) or an **effecting** (an effective implementation of a feature inherited in deferred form). The exact requirements in this case are captured by the Redeclaration rule, which will be given when we complete the study of inheritance, redefinition and deferred features.



In applying conditions [2](#) and [3](#), remember that two feature names are “the same” not just if they are written identically, but also if they only differ by letter case. Only the identifiers (**Feature_name**) of the features play a role in this notion, not any **Alias** they may have. ← Definition of “same feature name”, page [153](#).

The Feature Name **rule** will state a consequence of conditions [2](#) and [3](#) that may be more appropriate for error messages in some cases of violation. → Page [474](#).

Condition [4](#) prohibits a **frozen feature** from being declared as deferred. The two properties are conceptually incompatible since frozen features, by definition, may not be redeclared, whereas the purpose of deferred features is precisely to force redeclaration in proper descendants. ← Frozen features were introduced on page [141](#).

A companion constraint, seen as part of the Redefine Subclause rule in a **later chapter**, will prohibit the *redefinition* of a frozen feature. → Page [307](#).

Condition [5](#) applies to once functions. A **once routine** only executes its body on its first call. Further calls have no effect; for a function, they yield the result computed by the first call. This puts a special requirement on the result type *T* of such a function: if the class is **generic**, *T* should not depend on any formal generic parameter, since successive calls could then apply to instances obtained from different generic derivations; and *T* must not be **anchored**, as in the context of dynamic binding it could yield incompatible types depending on the type of the target of each particular call. The notion of *stand-alone type* captures these constraints on *T*. → “ONCE ROUTINES”, [23.14, page 641](#).

Condition [6](#) addresses delicate cases of polymorphism and dynamic binding, where anchored arguments and their implicit form of “covariance” may cause run-time errors known as “catcalls”. It follows from the general rule for signature conformance and is **discussed** with it. → See, in the chapter on types: “STAND-ALONE TYPES”, [11.12, page 347](#).

The last condition, [7](#), is the consistency requirement on features with an operator or bracket alias. It relies on the following definition (which has a validity code enabling compilers to give more precise error messages). → “Signature conformance”, page [386](#).



Alias Validity rule

VFAV

An **Alias** clause is **alias-valid** for a feature *f* of a class *C* if and only if it satisfies the following conditions:

- 1 • If it lists an **Operator** *op*: *f* is a **query**; no other query of *C* has an **Operator** alias using the **same operator** and the same number of arguments; and either: *op* is a **Unary** and *f* has no argument, or *op* is a **Binary** and *f* has one argument.
- 2 • If it lists a **Bracket** alias: *f* is a query with at least one argument, and no other feature of *C* has a **Bracket** alias.
- 3 • If it includes a **convert** mark: it lists an **Operator** and *f* has one argument.



The first two conditions express the uniqueness and signature requirements on operator and bracket aliases:

- An operator feature *plus alias* " \S " can be either unary (called as $\S a$) or binary (called as $a \S b$), and so must take either zero or one argument. Two features may indeed share the same alias— like *identity alias* "+" and *plus alias* "+", respectively unary and binary addition in class *INTEGER* and others from the Kernel Library — as long as they have different identifiers (here *identity* and *plus*) and different signatures, one unary and the other binary.

Such a feature must return a result and hence be a query — attribute or function. An attribute is only possible in the unary case, and is indeed permitted in line with the Uniform Access principle, although in most practical cases you'll need a function.

→ "UNIFORM ACCESS", 23.4, page 624.

- A bracket feature, of which there may be at most one in a class, will be called under the form $x [a_1, \dots, a_n]$ with $n \geq 1$, and so must be a query with at least one argument (and hence a function). Condition 2 tells us that there may be at most one bracket feature per class.



Condition 3 indicates that a **convert** mark, specifying "target conversion" as in *your_integer + your_real*, makes sense only for features with one argument, with an **Operator** which (from condition 1) must be a **Binary**.

5.20 SCOPE OF NAMES

Any feature of a class is accessible for use in any **Feature_declaration** of the class, and in its **Invariant** clause. Examples of the first use include unqualified (direct) calls in the **Feature body**, **Precondition**, **Postcondition** and **Rescue** clauses of a routine, and use as **Anchor** for an **Anchored type** in a declaration of a feature or of a routine argument.

→ See chapters 8 about **Feature_body**, 9 about **Precondition and Postcondition**, 26 about **Rescue**.

To avoid any ambiguity, **constraints** will prohibit reusing the name of a feature of the class for any other entity appearing in the class: formal argument or local variable of a routine, Object-Test Local of an **Object_test**.

→ Formal argument rule, page 220; Local variable rule, page 226; Object Test rule, page 659.

The Feature Declaration rule does **not**, however, prohibit conflicts between feature names and names of **classes**. It is possible for a feature to bear the same lower name as a class of the universe. You may sometimes find it convenient to write a feature declaration such as

```
error_window: ERROR_WINDOW
```



in a class text which only needs one feature of a certain type (here given by class *ERROR_WINDOW*) if you consider that the type name provides enough information to describe the role of the feature.

5.21 OBSOLETE FEATURES

As classes evolve in the constantly changing world of software development, you may find that a feature is no longer satisfactory.

If all you need is to change its implementation, then you should be able to update the feature without affecting its dependent classes (clients and proper descendants). For example, you may change a **Feature_body**, even if this causes replacing an attribute by a function or conversely, with minimum impact on dependents.

Unfortunately, this is not always the case. You may become unhappy with a feature's name, its signature or its specification — all of which are part of the interface and known to the clients.

The specification defines the routine's semantics. It is normally expressed by assertions; see chapter 9.

In such a situation, if you are certain that you have found a better replacement for the feature, you should perform the change without delay, for fear of prolonging the life of inferior software versions. But you must also take into consideration any existing dependent classes that relied on the feature. Clearly, you should avoid any change that would suddenly prevent such classes from functioning; but you may want to encourage their authors to adapt them to the new version within a reasonable time.

The preceding chapter showed how to declare an entire **class** as obsolete. This is a rather drastic decision; more often, the class as a whole remains adequate, but you want to update a few features.

← See [4.11, page 128](#), about obsoleting an entire class.

The feature obsolescence mechanism supports this need. By declaring a feature as **obsolete**, you keep it usable exactly as it was, while alerting its users to the existence of a better version. This provides a graceful way to phase out a feature while remaining friends with the developers of its clients.

Both routines and attributes may become obsolete. To mark a routine obsolete, give it an **Obsolete** clause, of the form

```
obsolete Message
```

where *Message* is a **Manifest_string**. This serves to warn authors of dependent classes that the routine should no longer be used. The *Message* should direct readers to alternate features.

Here is an obsolete routine which once figured in class **ARRAY** of the Kernel Library:



```

enter (i: INTEGER; new_value: T)
    obsolete "Use 'put (new_value, i)'"
    -- Replace by new_value the element at index i
require
    i >= lower; i <= upper
do
    ...(Appropriate algorithm)...
ensure
    set: item (i) = new_value
end

```

In older versions of the library, *enter* was the routine used to replace by *new_value* the value of the element at index *i* in an array. An examination of the consistency of names and conventions in the library resulted in a decision to update the routine; both the name (*put* rather than *enter*) and the order of arguments were changed. The [Message](#) explains this change.



To avoid cluttering library classes with features that are no longer relevant, library maintainers should not allow obsolete routines to loiter forever. After a suitable grace period — time for one or two new releases of the software to displace the older generations — they will have fulfilled their duties as Client-Friendly Transition Facilitators and should be retired with honors. This indeed happened to the above version of *enter*, which (although fondly remembered) disappeared long ago from class *ARRAY*, allowing — by an unforeseen twist of fate — *enter* to reappear as a synonym of *put*.

The syntax of class-level **Obsolete** clauses also applies — through the [production](#) for **Feature_value** — to routine-level clauses; here it is again: ← [Page 141](#).



```

Obsolete  $\triangleq$  obsolete Message
Message  $\triangleq$  Manifest_string

```

← This syntax appeared originally on page [129](#), followed by the semantics, applicable to obsolete features as well as obsolete classes.

As we saw there, marking a feature as **Obsolete** does not affect its semantics. But language processing tools may produce a warning when they process a client or descendant class that uses the feature. The warning should include the [Message](#).

The [contract view](#) of a class does not retain any feature marked obsolete. → “[-DOCUMENTING THE CLIENT INTERFACE OF A CLASS](#)”, [7.9, page 212](#).

A compiler or other language processing tool may also go further and provide an option that, under some conditions, will automatically update the text of client classes, replacing all calls to an obsolete routine by the body of the routine with appropriate argument substitution.



As noted in the discussion of obsolete classes, the availability of a feature obsolescence mechanism is not an excuse to grant a reprieve to software components that are buggy or otherwise deficient. If you discover a specification, design or implementation flaw, only one reaction is reasonable: correcting the mistake. A routine is a candidate for obsolescence only when, as originally written, it adequately covered a certain need, but is not in a manner satisfying your current standards. You prefer the new version, but the obsolete version is not *wrong*; it's just not what you wish to keep for the future.

5.22 NO IN-CLASS OVERLOADING

A consequence of the various validity rules on features and their names — to be expressed fully by the Feature Name rule — is that Eiffel never permits the same name to denote different features within the scope of a given class. This is expressed by the Feature Name rule. → Page 474.

When you see a feature name f in a class you immediately know what feature it refers to; and when you see a feature call $a.f(\dots)$ with a of type T you can immediately find the feature f to which it refers in T . (The actual feature to be called may of course, as a result of dynamic binding, be a redeclared version from a descendant.)

The full Feature Name rule appears only in a later chapter because it must take into account, along with the features introduced in a class, those inherited from its parents, and possibly renamed in the process: no name conflicts must arise between any of these. The rule must handle the effect of all inheritance mechanisms, including renaming, redefinition, and sharing under repeated inheritance.

The result, however, is simple: no name conflicts, period. The “overloading” mechanisms permitted by some languages is a confusing facility with no known benefit. It contradicts the principles of object technology and creates difficulties for both language users and compiler writers. The idea of using the same name to denote *different things* within a given scope can at best be described as rather puzzling.

Eiffel enforces instead the clear rule that in one class one feature name means one thing. (Nothing prevents you, as noted, from using the same name for feature names and *class* names, which can cause no ambiguity.) ← “SCOPE OF NAMES”, 5.20, page 164.

The only form of overloading permitted by Eiffel is the reuse of a feature name across *different* classes. The systematic naming conventions of the recommended style actually encourage you in this direction; the idea is to use a single name for features that correspond to the *same* basic semantics adapted to different contexts — the reverse of in-class overloading. Inter-class overloading takes its full power through dynamic binding, which allows dynamic (run-time) selection of the proper semantic variant, where intra-class overloading is static (compile-time). The dynamic form of inter-class overloading can also be called *semantic* overloading, in contrast with the *syntactic* nature of in-class overloading.

What is commonly known as “operator overloading”, the possibility of using the same operators, arithmetic in particular, for operations on different data types, is provided in a more general and flexible way by the combination of [Alias](#) clauses, permitting operator syntax for calls, and the conversion mechanism.

The inheritance relation

6.1 OVERVIEW

Inheritance is one of the most powerful facilities available to software developers. It addresses two key issues of software development, corresponding to the two roles of classes:

- As a **module extension** mechanism, inheritance makes it possible to define new classes from existing ones by adding or adapting features.
- As a **type refinement** mechanism, inheritance supports the definition of new types as specializations of existing ones, and plays a key role in defining the type system.

This chapter introduces the fundamental properties of inheritance, concentrating on the first view — the module aspect. It describes in particular the *renaming* mechanism, which brings considerable flexibility by letting you decide anew in each class on the names of the features it inherits. Later chapters discuss the type view of inheritance, which leads to Eiffel’s type system, and explore the feature adaptation mechanisms that go with it: redefinition, effecting, undefinition, and the sharing and replication mechanisms of repeated inheritance.

→ Chapters [11](#) to [13](#) on typing and [14](#) on conformance.

6.2 AN INHERITANCE PART

To define a class as inheriting from one or more others, include one or more **Inheritance** parts, each introduced by the keyword **inherit**.

Below is a slightly simplified form (omitting in particular the **Notes** clause) of the beginning of class `FIXED_TREE` from the EiffelBase Library. It shows a typical **Inheritance** part, indicating that `FIXED_TREE` obtains some of its features from three other classes:

- `TREE`, describing the general notion of tree, regardless of representation.
- `CELL`, describing elements used to store an individual piece of information (such as a tree node).
- `FIXED_LIST`, providing some of the implementation.



```

class
    FIXED_TREE [T]
inherit
    TREE [T]
        redefine
            attach_to_higher
        end
    CELL [T]
inherit {NONE}
    FIXED_LIST [T]
        rename
            off as child_off,
            after as child_after,
            before as child_before
        redefine
            duplicate, first_child
        end
feature
    ... (Rest of class omitted) ...

```

The classes listed in the two **Inheritance** parts, *TREE*, *CELL* and *FIXED_LIST*, are said to be the “parent classes”, or just “**parents**”, of *FIXED_TREE*. This is defined as a case of *multiple* inheritance. As the fixed-tree example shows, there is often a need to adapt the features of parents to a new class. This is achieved through the **Feature_adaptation** part of a **Parent** part, highlighted above: a **Redefine** clause for the *TREE* parent and a **Rename** clause for *FIXED_LIST*.

→ The notion of parent is defined precisely in the next section.

The first inheritance clause, introduced by just **inherit**, guarantees conformance of the class to the two parents listed. The other one, introduced by **inherit {NONE}**, provides non-conforming inheritance, giving the new class access to the features of the parent — *FIXED_LIST* — without introducing a “subtyping” (conformance) relation.

A **Feature_adaptation** part may contain **Redefine** and **Rename** subclasses, as here, as well as others — **Undefine**, **New_exports**, **Select** — listed in the syntax below.

6.3 FORM OF THE INHERITANCE PART

Here is the relevant syntax:



	Inheritance parts
Inheritance	\triangleq <code>Inherit_clause</code> ⁺
Inherit_clause	\triangleq inherit [<code>Non_conformance</code>] <code>Parent_list</code>
Non_conformance	\triangleq <code>"{ NONE }"</code>
Parent_list	\triangleq { <code>Parent</code> <code>;"</code> ... } ⁺
Parent	\triangleq <code>Class_type</code> [<code>Feature_adaptation</code>]
Feature_adaptation	\triangleq [<code>Undefine</code>] [<code>Redefine</code>] [<code>Rename</code>] [<code>New_exports</code>] [<code>Select</code>] end



As with all other uses of semicolons, the semicolon separating successive **Parent** parts is optional. The [style guidelines](#) suggest omitting it between clauses that appear (as they should) on successive lines.

→ “[OPTIONAL SEMI-COLONS](#)”, [34.10](#), [page 919](#).

Bla bla bla =====



Syntax (non-production): Feature adaptation

A `Feature_adaptation` part must include at least one of the optional components.

This rule removes a potential syntax ambiguity by implying that the **end** in **class *B* inherit *A* end** closes the class; otherwise it could be understood as closing just the `Parent` part.

A `Parent_list` names one or more `Parent` parts. Each is relative to a `Class_type`, that is to say a class name *B* possibly followed by actual generic parameters (as in *B* [*T*, *U*]). *B* must be the name of a class in the universe to which the current class belongs. This property yields a definition:

→ *Class types are studied in chapter 11. The requirement that *B* be a class of the universe follows from the Class Type rule, page 333.*



Parent part for a type, for a class

If a `Parent` part *p* of an `Inheritance` part lists a `Class_type` *T*, *p* is said to be a **Parent part for *T***, and also for the base class of *T*.



So in **inherit** *TREE [T]* there is a **Parent** part for the type *TREE [T]* and for its base class *TREE*. For convenience this definition, like those for “parent” and “heir” below, applies to both types and classes.

The earlier declaration of *FIXED_TREE* contains **Parent** parts for classes *TREE*, *CELL* and *FIXED_LIST*.

Specifying **{NONE}** (a **Non_conformance** marker) in an **Inherit_clause** yields a restricted form of inheritance, where the new class has access to the features and invariant of each parent listed, but the corresponding types do not conform to the parent types. This is known as *non-conforming inheritance* and detailed later in this chapter.

→ “**NON-CONFORMING INHERITANCE**”, [6.8, page 180](#).

After the **Class_type** in a **Parent** part you may also specify an optional **Feature_adaptation** clause listing the modifications that the new class wants to perform on the features it inherits from that parent. These modifications may affect various properties of the features, each handled by a subclause of **Feature_adaptation**:

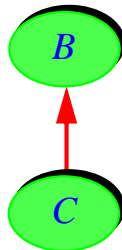
- Their effectiveness status, deferred or effective (**Undefine**).
- Their signature and implementation (**Redefine**).
- Their names (**Rename**).
- Their export status (**New_exports**).
- Their resolution of dynamic binding conflicts under repeated inheritance (**Select**).

Rename is studied later in this chapter, the others in subsequent chapters, in particular one devoted entirely to feature adaptation.

→ See [6.9, page 183](#) on **Rename**; [chapter 10](#) on **Feature_adaptation**, especially **Redefine** and **Undefine** (the latter in [10.19, page 290](#)); [page 204](#) on **New_exports**; [16.12, page 463](#) on **Select**.

6.4 GRAPHICAL CONVENTION

In pictorial representations of system structures, where classes appear as labeled ellipses, the inheritance relation is represented by single arrows (red if color is available) pointing from heirs’ ellipses to parents’ ellipses.



Parent and heir

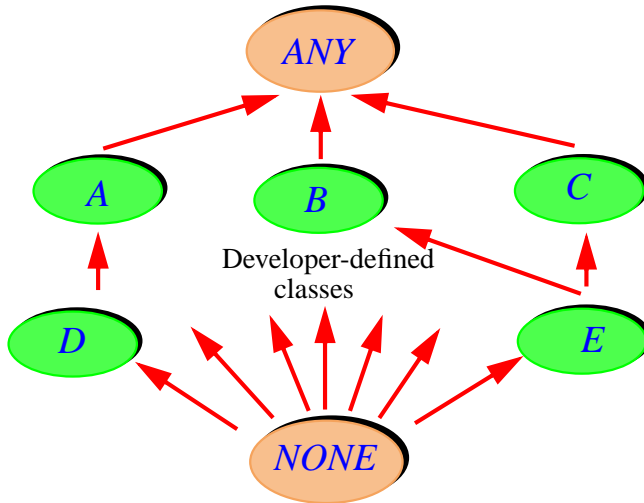
6.5 ANY

No class that you write is an orphan.

An important property of the inheritance structure is that every class inherits, directly or indirectly, from a class called *ANY*, of which a version is provided in the Kernel Library as required by the next rule. The semantics of the language depends on the presence of such a class, whether the library version or one that a programmer has provided as a replacement.

The convention ensuring this property — illustrated by the following figure — is that any class that doesn't have an explicit *Inheritance* part is considered to have *ANY* as its parent.

The inheritance structure



The figure also shows, at the bottom, a fictitious class *NONE*, studied next. But there's nothing fictitious about *ANY*:

→ "*NONE*", 6.6, page 175.



Class *ANY* rule *VHCA*
 Every system must include a non-generic class called *ANY*.

The key property of *ANY* is that it is not only an ancestor of all classes and hence types, but that all types **conform** to it, according to the following principle, which is not a separate validity rule (although for reference it has a code of its own) but a consequence of the definitions and rules below.



Universal Conformance principle *VHUC*
 Every type conforms to *ANY*.

To achieve the Universal Conformance principle, the semantics of the language guarantees that a class that doesn't list any explicit *Parent* is considered to have *ANY* as its parent. This is captured by the following notion: Unfolded Inheritance Part. The above definition of "parent", and through it the definition of "ancestor", refer to the Unfolded Inheritance Part of a class rather than its actual *Inheritance* part.

Unfolded Inheritance Part of a class

Any class *C* has an **Unfolded Inheritance Part** defined as follows:

- 1 • If *C* has an **Inheritance** part: that part.
- 2 • Otherwise: an **Inheritance** part of the form **inherit ANY**.

The fictitious clause **inherit ANY** is conforming.

If a class had one or more **Parent** clauses, but all were non-conforming, it would violate the Universal Conformance principle; we won't allow this. The solution is simple: in this (rare) case, just add **inherit ANY**, explicitly.

→ “[Parent rule](#)”, [page 178](#), [condition 4](#).



The special status of **ANY** implies two key properties, corresponding to the type and module views of inheritance:

- 1 • **ANY** is the most general of directly useful types: any type that you may define will conform to **ANY**.
- 2 • The features of **ANY**, describing general-purpose operations, are universal: any class that you may define will have access to them.

As a consequence of property 1, if you want a routine to be applicable to objects of arbitrary developer-defined types, you may give it an argument of type **ANY**. An example is the function, declared in **ANY** itself, that produce a duplicate of an object:

```
cloned (other: ANY): like Current
  -- Void if other is void; otherwise, new object
  -- field-by-field identical to object attached to other
  ... Rest of routine omitted ...
```

→ “[CLONING AN OBJECT](#)”, [21.4](#), [page 575](#).

Property 2 provides every developer-defined class with a set of important universal features coming from **ANY**. Some examples are the function *cloned* as sketched above, the procedures *default_rescue* and *default_create* giving default exception and creation behavior and the function *out* producing a string representation of any object.

→ See [26.5](#), [page 694](#), about *default_rescue*.



If you write a class that has no explicit **Parent**, and hence automatically inherits **ANY**, you can't do anything — renaming, redefinition, ... — to the features from **ANY**. If you do want to adapt them, the solution is simply to make the inheritance explicit:

```
class C inherit
  ANY

  redefine copy, default_rescue, ... end

feature
  ...
end
```



The special role of *ANY* turns any case of *multiple* inheritance into a case of *repeated* inheritance: on the earlier figure, *E* is an heir to both *B* and *C*, and hence an indirect descendant of *ANY* in two ways. Such situations are addressed through the normal rules of repeated inheritance (discussed below and detailed in a [later chapter](#)). Unless you specify otherwise, repeated inheritance from *ANY* will produce the expected effect for a class such as *E*: the class will have just one version of every feature from *ANY*, as if there were just one inheritance path.

← Page [173](#).

→ Chapter [16](#); see especially “[SHARING AND REPLICATION](#)”, [16.4, page 436](#).

6.6 NONE

The [overall inheritance figure](#) shows, along with *ANY* at the top, another special class at the bottom: *NONE*. This class is considered to inherit from all classes that have no other heirs — assuming appropriate renaming to remove any resulting name clashes.

← Page [173](#).

Unlike *ANY*, *NONE* does not actually exist as a class text (if only because that text would need to be updated every time anyone, anywhere, writes a new class!), but serves as a convenient fiction to make the inheritance structure and the type system complete.

NONE has no useful instance. It serves as the type of *Void*, which denotes a void reference. Since *NONE* is assumed to be a descendant of every class, the Parent rule [below](#) implies that no class may be an heir of *NONE*. The class does not export any feature, to help ensure that no feature call has a void target.

→ Page [178](#).

6.7 RELATIONS INDUCED BY INHERITANCE

The syntax tells us exactly when inheritance is “multiple”:

DEFINITION

Multiple, single inheritance

A class has **multiple inheritance** if it has an Unfolded Inheritance Part with two or more **Parent** parts. It has **single inheritance** otherwise.



What counts for this definition is the number not of parent classes but of **Parent** parts. If two clauses refer to the same parent class, this is still a case of multiple inheritance, known as **repeated inheritance** and studied later on its own. If there is no **Parent** part, the class (as will be seen below) has a de facto parent anyway, the Kernel Library class **ANY**.

→ See chapter [16](#)

The definition refers to the “Unfolded” inheritance part which is usually just the **Inheritance** part but may take into account implicit inheritance from **ANY**, as detailed in the corresponding definition below.

→ Page [174](#).



Multiple inheritance is a frequent occurrence in Eiffel development; most of the effective classes in the widely used EiffelBase library of data structures and algorithms, for example, have two or more parents. The widespread view that multiple inheritance is “bad” or “dangerous” is not justified; most of the time, it results from experience with imperfect multiple inheritance mechanisms, or improper uses of inheritance. Well-applied multiple and repeated inheritance is a powerful way to combine abstractions, and a key technique of object-oriented software development.

Inheritance introduces the “parent” and “heir” relations between classes:



Inherit, heir, parent

A class **C** **inherits** from a type or class **B** if and only if **C**’s Unfolded Inheritance Part contains a Parent part for **B**.

B is then a **parent** of **C** (“parent type” or “parent class” if there is any ambiguity), and **C** an **heir** (or “heir class”) of **B**. Any type of base class **C** is also an heir of **B** (“heir type” in case of ambiguity).

Listing **{NONE}** indicates that the relation does not imply conformance of the associated types:



Conforming, non-conforming parent

A parent **B** in an **Inheritance** part is **non-conforming** if and only if every Parent part for B in the clause appears in an Inherit_clause with a Non_conformance marker. It is **conforming** otherwise.

The reflexive transitive closures of the basic relations are also of interest:

“*Reflexive transitive closure*” means the relation iterated any number of times (zero or more).

Ancestor types of a type, of a class

The **ancestor types** of a type CT of base class C include:

- 1 • CT itself.
- 2 • (Recursively) The result of applying CT 's generic substitution to the ancestor types of every parent type for C .

The ancestor types of a *class* are the ancestor types of its current type.

The basic definition covers ancestor types of a *type*; the second part of the definition extends this notion to classes.

Case 1 indicates that a type is its own ancestor.

Case 2, the recursive case, applies the notion of *generic substitution* introduced in the discussion of genericity. The idea that if we consider the type C [$INTEGER$], with the class declaration **class** C [G] **inherit** D [G] ..., the type to include in the ancestors of C [$INTEGER$] as a result of this **Inheritance** part is not D [G], which makes no sense outside of the text of C , but D [$INTEGER$], the result of applying to D [G] the substitution $G \rightarrow INTEGER$; this is the substitution that yields the type C [$INTEGER$] from the class C [G] and is known as the generic substitution of that type.

From ancestor types we obtain ancestor classes, called just ancestors:



Ancestor, descendant

Class A is an **ancestor** of class B if and only if A is the base class of an ancestor type of B .

Class B is a **descendant** of class A if and only if A is an ancestor of B .

Any class, then, is both one of its own descendants and one of its own ancestors. *Proper* descendants and ancestors exclude these cases.

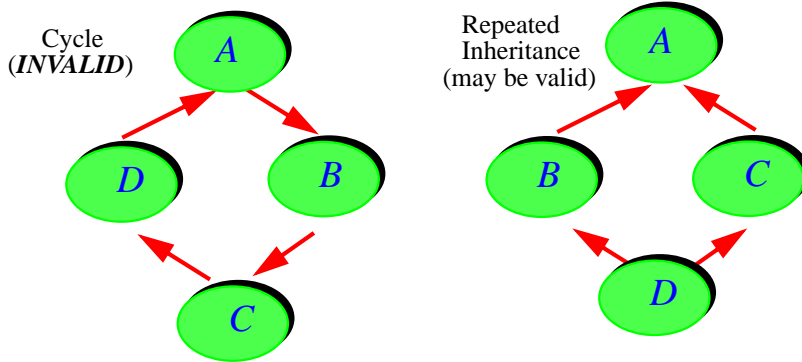


Proper ancestor, proper descendant

The **proper ancestors** of a class C are its ancestors other than C itself. The **proper descendants** of a class B are its descendants other than B itself.

6.6 PROHIBITING CYCLES

An important constraint governs the inheritance relation: there must be no inheritance cycles.



*Invalid cycle
vs. valid
repeated
inheritance*

In other words, you may not build a class structure as in the left part of the figure, where D inherits from B , B from A , A from C and C from D . More generally, it is invalid to have a set of classes C_0, C_1, \dots, C_n ($n \geq 1$), where C_0 and C_n are the same class and every C_i is an heir of C_{i+1} .

The reason for this restriction is easy to understand: making C an heir to B means defining the set of features of C as an extension of B 's feature set; the relationship cannot be mutual.



Prohibiting cycles does not mean prohibiting a class D from being a descendant of another class A in more than one way, as illustrated by the structure appearing in the right part of the above figure. This is a case of **repeated inheritance**, valid if it meets the relevant validity constraints. → Chapter 16.

These observations lead to the validity constraint on **Inheritance** parts:



Parent rule

VHPR

The Unfolded Inheritance Part of a class D is valid if and only if it satisfies the following conditions:

- 1 • In every Parent part for a class B , B is not a descendant of D .
- 2 • No conforming parent is a frozen class.
- 3 • If two or more Parent parts are for classes which have a common ancestor A , D meets the conditions of the Repeated Inheritance Consistency constraint for A .
- 4 • At least one of the Parent parts is conforming.
- 5 • No two ancestor types of D are different generic derivations of the same class.
- 6 • Every Parent is generic-creation-ready.

Condition [1](#) ensures that there are no cycles in the inheritance relation. → Page [466](#).

The purpose of declaring a class as **frozen** (case [2](#)) is to prohibit subtyping. We still permit the *non-conforming* form of inheritance, which permits reuse but not subtyping.

Condition [3](#) corresponds to the case of repeated inheritance; the [Repeated Inheritance Consistency constraint](#) will guarantee that there is no ambiguity on features that *D* inherits repeatedly from *A*. → Page [466](#).

Condition [4](#) ensures a central property of the type system: the Universal Conformance principle, stating that all types conform to *ANY*. Without this condition, it would be possible for all **Parent** parts of a class to be *non-conforming* and hence to cause violation of the principle. Note that in the Unfolded Inheritance Part there is always at least one **Parent** part, since the absence of an **Inheritance** part is a shorthand for **inherit ANY**, ensuring that condition [4](#) holds. → Studied below: [“NON-CONFORMING INHERITANCE”](#), [6.8, page 180](#).

Condition [5](#) avoids various cases of ambiguity which could arise if we allowed a class *C* to inherit from both *A [T]* and *A [U]* for different *T* and *U*. For example, if *C* redefines a feature *f* from *A*, the notation *Precursor {A}* in the redefinition could refer to either of the parents’ generic derivations. → Studied below: [“NON-CONFORMING INHERITANCE”](#), [6.8, page 180](#).

Condition [6](#) also concerns the case of a generically derived **Parent A [T]**; requiring it to be “[generic-creation-ready](#)” guarantees that creation operations on *D* or its descendants will function properly if they need to create objects of type *T*. → [“Generic-creation-ready type”](#), [page 360](#).



When applying the Parent rule, do not be misled by the “if” part of the “if and only if”: to guarantee that an **Inheritance** part is valid, you will also have to check conditions which do not appear explicitly in the rule. In particular:

- Every parent *P* must be a valid type; this means among other requirements that if *P* is generically derived, appearing as *B [X, ...]*, then *B* must be the name of a generic class in the surrounding universe and the actual parameters *X, ...* must be valid types matching the formal parameters of *B*. → The Class Type rule, [“VTCT”](#), [page 333](#), requires *P* to be the name of a class in the universe. On generic parameters, see the rule [“VTGD”](#), [page 359](#).
- Every **Feature_adaptation** clause (with its **Rename**, **Redefine** and other subclauses) must be valid.

The Parent rule does not need, however, to express such requirements explicitly: The General Validity rule implicitly adds to the constraint that all the sub-components are valid too. Be sure to remember this convention — without which the validity rules would become hopelessly complicated — whenever you see an “if and only if” validity constraint throughout this book. If you have the impression that the constraint does not cover every necessary condition, this is probably just because it omits the validity requirements on sub-components, as permitted by the [General Validity rule](#).

← General Validity rule: [page 98](#).

6.7 ADAPTING INHERITED FEATURES

The major purpose of inheriting from one or more classes is to obtain their features (together with the associated assertions, and the classes' invariants) as an addition to one's own. The features obtained by a class from its parents are called its *inherited* features. As already noted, this yields one of the two categories of features of a class; the others are *immediate* features, introduced in a class itself.

← "Features of a class" and "inherited features" were first discussed in [5.4, page 133](#)



The very notion of inherited feature indicates how inheritance provides an accumulation process enabling classes to use features defined in one or more previously existing classes – its proper ancestors.

Although a class inherits all its proper ancestors' features, it retains the flexibility to adapt them to its own context in various ways:

- A feature introduced in a certain class under a certain name may be known under different names in descendant classes. This is achieved through **renaming**.
- A feature defined with a certain signature, specification and implementation may get a new declaration changing any of these properties. This is achieved through **redefinition**.
- A feature introduced with a certain signature may get a new one. This is also achieved through redefinition, and through the associated mechanism of **anchored declaration**.
- A feature introduced in a proper ancestor with a specification but no implementation, known as a *deferred* feature, may get an implementation. This is the process of **effecting**.
- If a class *C* inherits two or more deferred features with compatible signatures and specifications, it may merge them into a single feature. This is a **join**.
- When a class *C* inherits the same feature from two or more of its parents, which themselves inherit it from a common ancestor, simple techniques are available to ensure that the result in *C* is only one feature (sharing) or several (duplication). The applicable rules are those of **repeated inheritance**.
- Under repeated inheritance, polymorphism and dynamic binding could cause conflicts, which you must remove through the **Select** mechanism.

→ "[THE JOIN MECHANISM](#)", [10.21, page 292](#).

The first of these techniques, renaming, is purely syntactical, affecting feature names rather than the features themselves. It is studied later in this chapter. The others determine the semantic adaptation of features to the context of new descendants; [later chapters](#) explore them in detail.

→ Chapter [10](#) on feature adaptation and [16](#) on repeated inheritance

N-CONFORMING INHERITANCE

(The mechanism described here is for advanced users. On first reading you may [skip](#) the present section.)

→ Skip to "[RENAMING](#)", [6.9, page 183](#)



One of the principal applications of inheritance — in its “type” rather than “module” persona — is to govern conformance. The basic idea is simple: in the most common cases, an assignment of the form $al := bl$ with al of type A and bl of type B is valid if B is a descendant of A . You can similarly call $f(bl)$ if f has a formal argument of type A . The details appear in the [conformance](#) chapter.

→ Chapter 14.

Sometimes, you may want inheritance *without* conformance: the module-only side of inheritance, disallowing such assignments and arguments passing. To force this it suffices to mention of A in the corresponding [Parent](#) part by keyword `{NONE}`, as in



```
class B inherit
  {NONE} A
  ... Feature_adaptation clause if needed ...
  ... Rest of class omitted ...
```

Adding `{NONE}` in this fashion does not affect the basic properties of the inheritance relation; it simply means that type B will not conform to A through this inheritance link.

In a case of repeated inheritance, B might still conform to A through another inheritance link.



The syntax is reminiscent of the possibility of declaring features in a clause `feature {NONE}`, rather than just `feature`, to restrict its export status.

This facility is useful only in specific cases of restricting an inheritance link to “implementation inheritance” or “facility inheritance”: you want the reusability benefits of inheritance, but not the subtyping part.



Some simple-minded presentations of object technology will tell you that this is “wrong” and that inheritance should always involve subtyping. Although they can legitimately point to incorrect uses of inheritance, it is improper to disallow implementation inheritance altogether, as it has many perfectly valid uses. The chapter on the methodology of inheritance in *Object-Oriented Software Construction* discusses these issues in detail and presents a taxonomy of the uses of inheritance.

In this book we will see two major applications of non-conforming inheritance, both of which use it to remove potential ambiguities: repeated inheritance and convertibility.

- The repeated inheritance chapter will show that it is sometimes possible for a class to obtain two different versions of a feature inherited from a common ancestor through more than one path. This creates a potential ambiguity because of polymorphism and dynamic binding, since a call of the form $a.f$, where a is of the repeated ancestor type, could in principle trigger either of the two variants if a is attached at run time to an instance of the common descendant type. When such a conflict arises, you will resolve it through a **Select** clause. The problem only arises, however, if both paths are conforming; by using non-conforming inheritance whenever you don't need subtyping you reduce the need for **Select** and simplify your class texts.
- The study of convertibility will show how to make a type convertible to another by including conversion procedures, as in → *Chapter 15.*



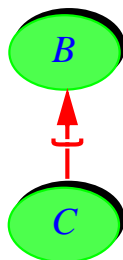
```
class A create
  from_B convert {B}
  ... Rest of class omitted ...
```

which makes assignments such as $a1 := b1$ (and corresponding argument passing) valid; they will cause a conversion using the listed creation procedure $from_B$. To avoid any ambiguity, the Conversion Procedure rule prohibits such a scheme when B conforms to A , as this would also make the assignment valid but with a different semantics (reference reattachment with no conversion). The general principle is that a type may conform or convert to another, but not both. In some cases you might still like B to inherit from A for its features only. It suffices in this case to make B list $\{NONE\} A$, rather than just A , as its **Parent**.

→ “*Conversion Procedure rule*”, page 411; “*Conversion principle*”, page 408.

This discussion also explains why we needed condition 4 of the Inheritance rule, requiring that if there are **Parent** parts they can't all be non-conforming: we need at least one conforming branch to ensure that all types conform to **ANY** — the Universal Conformance rule. ← *Page 178 (both rules).*

The graphical representation of inheritance links has a slightly different form (similar to the convention for the “expanded client” relation) to signal non-conforming inheritance: → *Page 198, in the next chapter.*



Parent and non-conforming heir

6.9 RENAMING

As part of its **Feature_adaptation**, any **Parent** part may include a **Rename** subclause, which serves to adapt names of inherited features to the local context of the new **class**.

Here is a **Rename** subclause from the previous example:



```
rename
  off as child_off,
  after as child_after,
  before as child_before
```



Renaming is especially useful in two cases:

- With renaming, you may correct any **name clash** occurring because of multiple inheritance. A name clash occurs when two or more parents of a class have a feature of the same name, and would usually make the class invalid if not removed by renaming.
- Renaming also enables a class to offer its inherited features to its clients and descendants under a terminology appropriate to its own context, rather than to the context of the parents from which it inherited them. In other words, it helps make sure that, beyond offering the right *features*, you also offer them under the right *feature names*.

→ *“NAME CLASHES”*, 10.23, page 297, discusses the exact cases in which name clashes are prohibited.

The general syntax of a **Rename** clause is:



Rename clauses

```
Rename ≙ rename Rename_list
Rename_list ≙ {Rename_pair ", " ...}+
Rename_pair ≙ Feature_name as Extended_feature_name
```

The first component of a **Rename_pair** is just a **Feature_name**, the identifier for the feature; the second part is a full **Extended_feature_name**, which may include an **alias** clause. Indeed:

- To identify the feature you are renaming, its **Feature_name** suffices.
- At the same time you are renaming the feature, you may give it a new operator or bracket alias, or remove the alias if it had one.

Forms of feature adaptation other than renaming, in particular effecting and redefinition, do not affect the **Alias**, if any, associated with a **Feature_name**.

So if *B* has the features

```
plus alias "+"
multiplied alias "*"
divided alias "/"
item alias "["
f
g
```

you may define a new class

```
class C inherit
  B
  rename
    plus as sum alias "+",
    multiplied as times,
    divided as divided alias "//",
    item as item,
    f as f alias "[",
    g as h alias "||"
  end
... Rest of class omitted ...
```

Warning: this is an extreme case, illustrating the possibilities but not intended as a model of style!

Then for the features offered by *C* to its direct clients:

- *plus* changes its identifier to *sum* and keeps its alias. Without the **alias** part it would no longer have an operator alias in *C*.
- *multiplied* is renamed to *times* and loses its alias.
- *divided* keeps its identifier but changes its alias; you can't change just the alias without giving a full new **Extended_feature_name**, which in this case reuses the previous **Feature_name** (the identifier *divided*).
- *item* keeps its identifier and loses its bracket alias; again you have to repeat the identifier.
- *f* takes over the bracket alias vacated by *item*. Since every class may have at most one feature with the bracket alias, this would not be possible without the change to *item*.
- *g* gets a new identifier and a new alias, the **free operator** `||`.

→ "[Free operator](#)",
page 893

The aliases all assume that the corresponding features have the right signatures; for example "+" as a **Binary requires** a one-argument query.

← "[Alias Validity rule](#)", page 163

The **Rename** clause is subject to a constraint.:



Rename Clause rule

VHRC

A **Rename_pair** of the form *old_name as new_name*, appearing in the **Rename** subclause of the **Parent** part for *B* in a class *C*, is valid if and only if it satisfies the following conditions:

- 1 • *old_name* is the final name of a feature *f* of *B*.
- 2 • *old_name* does not appear as the first element of any other **Rename_pair** in the same **Rename** subclause.
- 3 • *new_name* satisfies the Feature Name rule for *C*.
- 4 • The **Alias** of *new_name*, if present, is alias-valid for the version of *f* in *C*.

→ The *Feature Name rule*, page 474, expresses that no other feature of *C* has *new_name* as its final name.

→ “*Feature Name rule*”, page 474.

In condition 4, the “alias-valid” condition captures the signature properties allowing a query to have an operator or bracket aliases. It was enforced when we wanted to give a feature an alias in the first place and, naturally, we encounter it again when we give it an alias through renaming.

← “*Alias Validity rule*”, page 163.

← Clauses 5 and 7 of “*Feature Declaration rule*”, page 162.

Renaming is a purely syntactical mechanism:



Renaming principle

Renaming does not affect the semantics of an inherited feature.

The “positive” semantics of renaming (as opposed to the negative observation captured by this principle) follows from the definition of *final name* and *extended final name* of a feature below.

→ Page 186.

This principle indeed adds nothing by itself to the semantics of the language; it is there to remove any uncertainty. Experience has shown that renaming sometimes confuses newcomers to object technology — surprisingly, since the idea is particularly simple: to distinguish between a feature and its name.

See “*Repentant Java programmer can’t understand the difference between a feature and a feature name*”, in *Proc. BEIROOT ‘05 (Bizarre Experiences In Remedial Object-Oriented Training)*, Beirut, Aug. 2005, pages 22345-27226.

6.10 FEATURES AND THEIR NAMES



A class defines a set of features, each with a certain feature names. The two concepts are clearly distinct.

A feature is a certain component (attribute or routine), characterized by a signature, an associated algorithm (for a routine), a value (for a constant attribute), and possibly other properties. Such a feature is “*a feature of*” one or more classes: the class which introduces it, and (subject to feature adaptation mechanisms) all the descendants of that class.

Every feature of a class has a name in that class. This association between a feature and a feature name only exists relative to the class. The same *feature* may have different *feature names* in different classes.

This is precisely what renaming achieves. The presence, in a **Parent** clause for B in C , of a **Rename** subclass of the form

```
rename ..., f as g, ...
```

implies that the inherited feature known as f in B is known as g in C .

The precise definitions are the following:

Final name, extended final name, final name set

Every feature f of a class C has an **extended final name** in C , an **Extended_feature_name**, and a **final name**, a **Feature_name**, defined as follows:

- 1 • The final name is the identifier of the extended final name.
- 2 • If f is immediate in C , its extended final name is the **Extended_feature_name** under which C declares it.
- 3 • If f is inherited, f is obtained from a feature of a parent B of C . Let *extended_parent_name* be (recursively) the extended final name of that feature in B , and *parent_name* its final name of f in B . Then the extended final name of f in C is:
 - If the **Parent** part for B in C contains a **Rename_pair** of the form **rename parent_name as new_name: new_name**.
 - Otherwise: *extended_parent_name*.

The final names of all the features of a class constitute the **final name set** of a class.

Since an inherited feature may be obtained from two or more parent features, case 3 only makes sense if they are all inherited under the same name. This will follow from the final definition of “inherited feature” in the discussion of repeated inheritance.

The extended final name is an **Extended_feature_name**, possibly including an **Alias** part; the final name is its identifier only, a **Feature_name**, without the alias. The recursive definition defines the two together.

Also convenient is the notion of “inherited name” of an inherited feature:

Inherited name

The **inherited name** of a feature obtained from a feature f of a parent B is the final name of f in B .

The notion of “class of origin” was first introduced on page 133. The full definition appears on page 311.

→ How the final name set is actually determined depends on renaming, redefinition and joining, as discussed in chapters 10 and 16. See further comments about the final name set on page 473.

→ “Inherited features”, page 470.

In the rest of the language description, references to the “name” of a feature, if not further qualified, always denote the final name.



Renaming — to press the point! — does not change any of the inherited features, but simply changes the names under which those features will be known by clients and descendants. Consider a feature f , which has the final name old_name in a class B . By writing an heir C as



```
class C inherit
  ...,
  B
  rename ..., old_name as new_name, ... end
```

you decide to make the inherited feature available to C , C 's descendants and (if it is exported) C 's clients under the name new_name .

As a consequence, you have also freed the inherited name of f , here old_name , so that another feature of C may now use this name. That other feature could come from various places:

- 1 • It could be a new feature introduced by C itself, for which you wish to use the name old_name .
- 2 • It could be a feature inherited from a parent of C other than B , and having the name old_name in that parent. Here, without renaming, you would have introduced a — usually invalid — name clash in C .
- 3 • It could even be a feature inherited from B or another parent under some other name, and renamed old_name in C . This case is somewhat contorted, but it does occasionally arise.

Whatever the case, remember that if you do decide to reuse old_name for another feature of C , you do not introduce any connection between that feature and the original feature f , obtained from B under the inherited name old_name . The two are unrelated; for example one could be a procedure and the other an attribute.

The following example illustrates these properties. Assume a class $COLORS$ with features of names red , $orange$, $black$, $white$, and $FRUITS$ with a feature of name $orange_fruit$. You can write a class of the form



```

class FRUITS_AND_COLORS inherit
  COLORS
    rename
      orange as orange_color, red as red_color,
      black as white, white as black
    end
  FRUITS
    rename
      orange_fruit as orange
    end
  feature
    red: INTEGER
  end

```

There is no assumption that these classes and features have any use as abstractions reflecting their names; they just illustrate some language properties.

The feature *orange* of class *COLORS* is known in *FRUITS_AND_COLORS* as *orange_color*; this makes the name *orange* available for the feature inherited from *FRUITS* under the name *orange_fruit*. The feature *red* of *COLORS* is known in *FRUITS_AND_COLORS* as *red_color*, making the name *red* free for a new attribute introduced in *FRUITS_AND_COLORS* with no connection to the original *red*. Finally each of *COLORS*'s features *black* and *white* is known in *FRUITS_AND_COLORS* under the other's name.



As this example illustrates, you should understand the renamings induced by a *Rename* subclass as all simultaneous; this allows such constructions as *rename black as white, white as black* to make sense. In other words, even if the *Rename* subclass includes a *Rename_pair* *old_name as new_name*, other occurrences of *old_name* or *new_name* as the first element of a *Rename_pair* in the same subclass must still be interpreted as in the parent.



This last case, which swaps the names of two inherited features, is rather extreme. It illustrates, however, the importance of renaming to the building of professional-quality reusable software components. Writing a class as heir to another means endowing the new class with a certain *functionality*, as provided by the parent's features. But this does not by itself make these features available under a *terminology* consistent with the heir's specific context. Renaming is there to guarantee that, for the heir, its clients and its descendants, the terminology is just as right as the functionality is.

An auxiliary notion resulting from this discussion proves convenient:

Declaration for a feature

A *Feature_declaration* in a class *C*, listing a *Feature_name* *fn*, is a **declaration for** a feature *f* if and only if *fn* is the final name of *f* in *C*.

Although it may seem almost tautological, we need this definition so that we can talk about a declaration “for” a feature *f* whether *f* is immediate — in which case *fn* is just the name given in its declaration — or inherited, with possible renaming. This will be useful in particular when we look at a *redeclaration*, which overrides a version inherited from a parent.

6.11 INDEPENDENCE OF INHERITANCE AND EXPANSION

The “expanded” or “reference” status of a class is not inherited.

As you may remember, a **Class_header** may begin with

```
expanded class C...
```

→ See “*CLASS TEXT STRUCTURE*”, 4.6, page 117..

as opposed to the more common **class C** or **deferred class C**. If the **expanded** mark is present, the class and types based on it are said to be expanded. Creation of an instance, as in

```
x: C
...
create x...
```

will yield an objects with *copy semantics* rather than reference semantics. What effect does this have on heirs of *C*?



The answer is straightforward: no effect. The only consequence of the expansion status of a class is the semantics of objects of the corresponding types, such as the object attached to *x* above. An expanded class may inherit from a non-expanded one, and conversely. The expansion status is not transmitted, but entirely determined by the class’s own **Class_header**.

This convention makes it easy to provide both a reference and expanded versions of the same class, as in

```
class RC feature
    ... Full class declaration: feature declarations, invariant etc. ...
end

expanded class EC inherit
    RC
    -- No need to write anything else, except possibly
    -- Notes and Creation clauses
end
```

The two classes have the same features; one is expanded, the other is not. Because of the rules on creation, each will have to list the procedures, if any, that it plans to use as creation procedures.

Clients and exports

7.1 OVERVIEW

Along with inheritance, the client relation is one of the basic mechanisms for structuring software.

In broad terms, a class C is a client of a type S — which is then a *supplier* of C — when it can manipulate objects of type S and apply S 's features to them.

The simplest and most common way is for C to contain the declaration of an entity of type S .

This occurs for example when C includes a declaration of the form

```
x: S
```

To an entity such as x , C may apply the features that the designer of S has explicitly made **available** (has **exported**) to the clients of S . In other words, the client relation allows a class to rely on the facilities provided by another as part of its official interface.

Variants of the relation introduce similar dependencies through other mechanisms, in particular generic parameters.

Although the original definitions introduce “client” in its various forms as a relation between a class and a type, we’ll immediately extend it, by considering S 's base class, to a relation between classes.

This chapter defines the client relation in its diverse forms; it studies how a class can export its features to its clients, and how these clients can use the exported features. The discussion ends with a solution, resulting from the export mechanism, to an important practical question: how to document a class.

7.2 ENTITIES



Classes become clients of one another by using typed components, **entities** and **expressions**, both denoting run-time values (references or objects). An *entity* of a class C is one of the following:

→ Chapter 19 covers entities, with a full definition on page 512. Chapter 28 covers expressions.

- An attribute of C .
- A formal argument to a routine of C .
- A local variable of a routine of C , including (for a function) the predefined entity **Result** denoting the result.
- An Object-Test Local (in an `Object_test`).
- **Current**.

Any such entity has a type, defined in its declaration.

Expressions are obtained by combining entities and function calls through operators (which themselves denote calls). Any expression has a type, deduced from the type of its components.

It's those entities and expression types that generates the client relation, by making C a "simple client" of T , as defined below, as soon as it has an entity or expression of type T .

7.3 CONVENTIONS

We need a few conventions to simplify the discussion of the client and supplier relations.

It is useful to distinguish between several variants of the client relation: simple client, expanded client and generic client relations. Each is studied below. The more general notion of client is the union of these cases, according to the following definition.



Client relation between classes and types

A class C is a **client** of a type S if some ancestor of C is a simple client, an expanded client or a generic client of S .

Recall that the ancestors of C include C itself. The definition involves all of C 's ancestors to include dependencies caused by inherited features along with those due to the immediate features of C . Assume that an inherited routine r of C uses a local variable x of type S ; this means that C may depend on S even if the text of C does not mention S . (If C redefines r , the definition may then needlessly make C a client of S , but this has no harmful consequences.)

← "RELATIONS INDUCED BY INHERITANCE", 6.7, page 175.

Next, we need to clarify a technical point: when does the discussion of clients and suppliers involve classes, and when is it about types? If, as above, you declare in class C the entity x as being of type S , S is a type. That type may be a class, but it may also be a less trivial type; for example, S may be the *generically derived* type

← A similar problem arose for inheritance: syntactically, a **Parent** is a type, not a class, but the definitions in 6.3, page 170 and 6.7, page 175, made it possible to talk about parent classes.



$D[U]$

where D is a generic class and U , itself a type, is the actual generic parameter for this particular generic derivation of D .

As this example indicates, the client relation in its most basic form holds between a class and a type, not necessarily between a class and another class. It generalizes immediately, however, to a relation between classes, since every type is derived from some class called its *base class*. In most cases, the base class of a type is obvious: for example, in a generic derivation such as $D [U]$, the base class is D ; and if a non-generic class is used as a type, it is its own base class. Hence a simple convention:

→ The complete definition of "base class", for every possible category of type, appears in chapters [11](#) to [13](#).



Client relation between classes

A class C is a **client of a class B** if and only if C is a client of a type whose base class is B .

The same convention applies to the simple client, expanded client and generic client relations.

As a result of these conventions, it suffices for the following sections to define what it means for a class C to be a client (in one of the three variants) of a type S .

If C is a client of S and S is a client of B , we will say that C is an *indirect* client of B :

Finally, we sometimes need to refer to the inverse relations:



Supplier

A type or class S is a **supplier** of a class C if C is a client of S , with corresponding variants: simple, expanded, generic, indirect.

7.4 SIMPLE CLIENTS

The most immediate case of the client relation is for a class C to be a **simple client** of a type S , which is then said to be a **simple supplier** of C . This happens in particular whenever C contains a declaration of the form

```
x: S
```

Assuming the class skeletons:



```
class A feature
```

```
  x: B
```

```
  y: C [D]
```

```
  ...
```

```
end
```

```

class B feature
  z: E
  ...
end

```

Then A is a simple client of B and C , and B a simple client of E . B and C are, conversely, simple suppliers of A , and E of B .

In this example a class becomes a simple client of certain types through the declarations of its entities. C will also be a simple client of S whenever it contains an expression of type S .

Here is the precise definition:

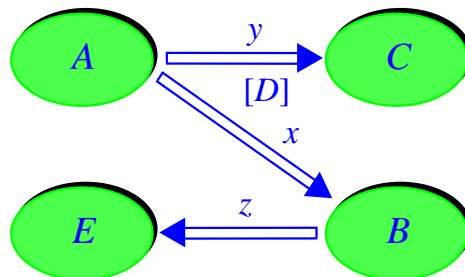


Simple client

A class C is a **simple client** of a type S if, in C , S is the type of some entity or expression or the `Explicit_creation_type` of a `Creation_instruction`, or is one of the `Constraining_types` of a formal generic parameter of C , or is involved in the `Type` of a `Non_object_call` or of a `Manifest_type`.

The constructs listed reflect the various ways in which a class may, by listing a type S in its text, enable itself to use features of S on targets of type S .

The suggested graphical representation, illustrated below, shows the simple client relation with a double arrow. The arrow may be labeled above by the name of the corresponding entity, and below by the names of the actual generic parameters in brackets, as with $[D]$ for the relation between A and C .



*Simple Clients
and suppliers*

No constraint restricts how the classes of a system may be simple clients of one another. In particular, cycles are permitted: a class may be its own simple client, both directly according to this definition and indirectly.

For example you might need a class *PERSON* introducing attributes

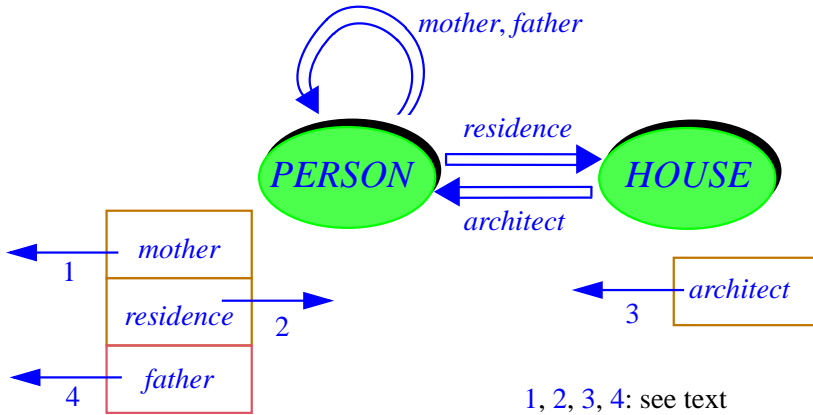


mother, father: PERSON

This is an example of a direct cycle of the simple client relation. Cycles may also be indirect; for example, a class *HOUSE* might introduce an attribute

architect: PERSON

with class *PERSON* having an attribute *residence* of type *HOUSE*.



Cycles in the simple client relation

As usual, the ellipses represent classes. The rectangles show typical instances of these classes, with their fields

This means is that every person has a mother, father and residence, and every house has an architect. There is nothing contradictory (no vicious circle) in these declarations; at the implementation level they create no difficulty either since it is possible to implement the corresponding attributes as references, as the lower half of the figure suggests by showing typical instances of the classes: references 1, 3 and 4 are to instances of *PERSON*, reference 2 to an instance of *HOUSE*.

Some of these references could also be void, but only if the attribute types are declared as detachable: *? PERSON*, *? HOUSE*.

→ Chapter 24



To avoid any confusion we must distinguish the client relation between *classes* (and types), which is the topic of the chapter, from any specific link that it induces between individual *objects* that are instances of these classes. In particular, a cycle between two classes does not imply a cycle between specific objects; in the situation of the above figure, links 2 and 3 will only connect the objects shown in a “Frank Lloyd Wright setup” (the case of an architect that lives in a house he has designed). Links 1 and 4 cannot be cyclic since no person is his own father or mother. This should in fact be an invariant of the class: *mother* != *Current*.

7.5 EXPANDED CLIENTS

Expanded types introduce a special variant of the client relation, called “expanded client”.



Expanded types describe objects that behave with **copy semantics** rather than reference semantics: an assignment or argument passing will copy the object, not just attach a reference to it. Non-expanded types, which use reference semantics, are called **reference types**. → See chapter 11 for the details of expanded types, starting with 11.9, page 335.

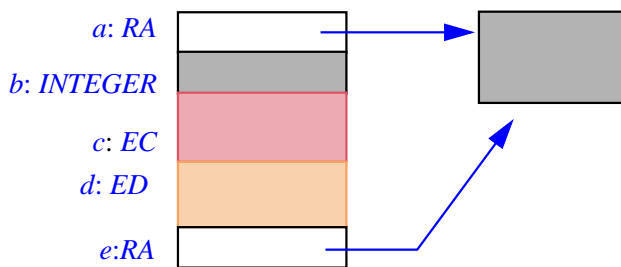
A type is expanded if and only if its base class is itself expanded; it must be declared as **expanded class** rather than just **class**.

An application of expanded classes and types is to describe **composite objects**, the name given to objects that contain **subobjects**. Consider a class declaration with the following attributes (routines omitted)



```
class B feature
  a: RA
  b: INTEGER
  c: EC
  d: ED
  e: RA
end
```

where *RA* is a reference type, *EB* and *ED* are expanded types; *INTEGER*, a basic type, is also expanded. Then instances of *C* can be viewed as composite objects. The figure below shows a typical one



Composite object

The figure shows a conceptual view of the objects and subobjects; it does not necessarily describe the actual representation, since it is always possible to represent expanded fields by references rather than subobjects. See below.

This example illustrates the expanded variant of the client relation:



Expanded client

A class *C* is an **expanded client** of a type *S* if *S* is an expanded type and some attribute of *C* is of type *S*.

Only attributes matter for this definition, since other expressions and queries do not cause subobjects.



The last example and its illustration appear to suggest that we should prohibit cycles in the expanded client relation, as in

```
expanded class EA feature
  c: EC
  ...
end
```

```
expandedclass EC feature
  a: EA
  ...
end
```

or the even more absurd-looking case of a direct cycle:

```
expandedclass EB feature
  b: EB
  ...
end
```

It's indeed not possible physically for every instance of *EC* to contain an instance of *EA* if every instance of *EA* contains an instance of *EC*, or for every instance of *EB* to contain another of the same type.

But in fact such examples — useful or not — create no particular problem and we don't need to prohibit them. Remember that the figure showing expanded fields as subobjects is just an illustration; the only semantic property that matters is that instances of expanded types have *copy semantics*, meaning that:

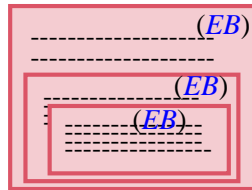
- An assignment or argument passing will copy the object, not just assign a reference.
- An equality operation will compare objects contents, not references.

To support these rules, expanded types have **lazy** initialization semantics: expanded objects need only be created when first accessed.

Any implementation of expanded attributes that supports these properties is acceptable. In particular, while the *subobject* representation is generally preferable when possible (that is to say, in the absence of cycles), it is always possible to use *references* instead, and create the associated objects on demand, as part of lazy initialization. Cycles are then not a problem.

This solution is available for attributes *a*, *c* and *b* in the last example.:

Conceptually, you may consider that if an object **OB** of the expanded type **EB** has a field of that same type (the same would apply to the indirect case involving instances of **EA** and **EC**), the field still represents a that subobject, just “**written smaller**” inside the first:



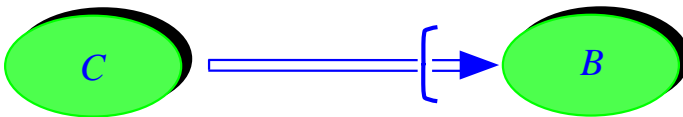
Embedded objects

Lazy semantics implies that all subobjects are evaluated only when needed, and an execution can only perform a finite number of such evaluations; so the process will stop and the level of object embedding remains finite.

Of course we can't really “write smaller” in the memory of a computer, so the most obvious implementation will use embedded sub-objects for expanded attributes at the first level only, and then references for the (rare) case of cycles in the expanded relation. But the subobject embedding picture remains applicable conceptually.

Earlier versions of Eiffel had an “Expanded Client rule” prohibiting cycles in the expanded client relation. The lazy semantics of expanded types now makes it unnecessary.

The graphical representation of the expanded client relation uses a double line, as with the simple client relation, but with a brace near the arrowtip:



Expanded client

See corresponding convention for expanded inheritance, page 182.

7.6 GENERIC CLIENTS



Assume that **B** is a generic class, and that class **C** contains a declaration of the form

$x: B [S]$

using **S** as actual generic parameter for the generic derivation of **B**.

As seen above, this declaration makes **C** a simple client of **B**. But it also introduces a dependency between **C** and **S**. This dependency is in fact similar to what happens if **C** has an entity or expression of type **S**; this variant of the client relation is called **generic client**.



Generic client, generic supplier

A class C is a **generic client** of a type S if for some generically derived type T of the form $B [\dots, S, \dots]$ one of the following holds:

- 1 • C is a client of T .
- 2 • T is a parent type of an ancestor of C .

Case 1 captures for example the use in C of an entity of type $B [S]$ (with B having just one generic parameter). Case 2 covers C inheriting directly or indirectly (remember that C is one of its own ancestors) from $B [S]$.

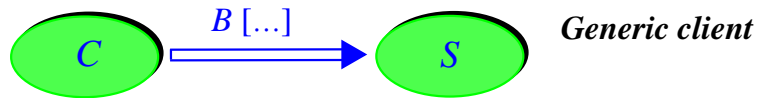
As case 2 of the definition indicates, C may become a generic client of S by using S as actual generic parameter not just in the type of an entity or expression (as with x above), but also in a **Parent** part, as in



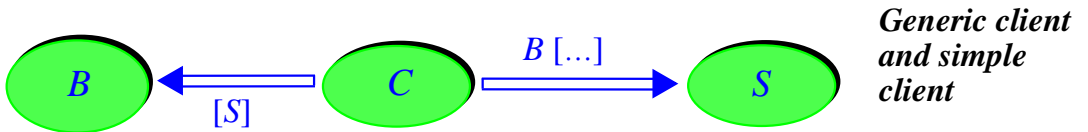
```
class C inherit
    B [..., S, ...]
feature
    ...
```

The **Parent** part may appear not just in C itself as here, but in any one of its ancestors: generic client status is passed on through inheritance.

The graphical convention for the generic variant of the client relation uses a double arrow from the generic client to its generic supplier, listing the base class with three dots in brackets [...]. For a declaration $x: B [S]$:



The full picture is in this case:



Do not confuse the two forms of client relation arising here: C is a simple client of B , with S as a generic parameter, through the declaration of x (left part of the figure); but that declaration also makes C a generic client of S , assumed here for simplicity to be a non-generic class..

There is no restriction on how the classes of a system may become generic clients of each other.

7.7 INDIRECT CLIENTS

=====
Text was moved after all the other definitions per ISO recommendation



Indirect client

A class A is an **indirect client** of a type S of base class B if there is a sequence of classes $C_1 = A, C_2, \dots, C_n = B$ such that $n > 2$ and every C_i is a client of C_{i+1} for $1 \leq i < n$.

The indirect forms of the simple client, expanded client and generic client relations are defined similarly.

7.8 EXPORT CONTROLS AND INFORMATION HIDING

The client relation determines how a class may call features of a certain type, on entities of that type. Such calls are subject to *export* controls, implementing a policy of “information hiding”.

Assume C is a simple or expanded client of S . C declares one or more entities or expressions of type S ; let $x: S$ be one of them. The benefit, for C , is to be able to call S 's features on entities and expressions such as x . The simplest form of call, occurring in C , is

$$x.f(\dots)$$

where r is a feature of S . This form uses dot notation; forms using operators and assignment procedures are also possible.

Not all such calls, however, are permitted; in particular, not all the features of a class need be callable by all clients. The designer of a supplier class may want to keep some features private, or available to some clients only, because they are only of internal use and subject to change; letting any client access them directly would jeopardize further evolution, by requiring a change of the client classes every time these features change.

This is especially true of features that reflect not the services directly offered by a class to its clients, but internal support for the implementation of these services, resulting from specific choices of representation and algorithms. By keeping such features private, the designer of the supplier class protects clients against the effects of later reversals of these choices. This policy is part of **information hiding**, a central principle of software development, which holds that the developer of a module must make a clear distinction between two categories of properties: those which are local to the module itself (its “secrets”, or “private properties”); and those that are available to clients (“public properties”).

→ See chapter 23 about the various forms, uses and properties of calls.

→ Chapter 25 covers the conditions on call validity.

Eiffel supports information hiding in a number of ways, including Design by Contract, the notion of contract view, and the principle of Uniform Access. One of the principal tools for information hiding is the ability for a class to define a specific export status for every one of its features. You can achieve this through two related mechanisms:

- For immediate features (those introduced in the class itself), you may specify export restrictions by listing clients in a **Feature_clause**. In the absence of such a restriction, features are available to all possible clients.
- For inherited features (those obtained from parents), you may change the export policy specified by each parent through the **New_exports** subclasses of the corresponding **Parent** parts. Inherited features not listed there retain the export status they had in the parent.

The following discussion explain these two mechanisms in detail.

Restricting exports

You define the export status of an immediate feature by specifying authorized clients in the **Feature_clause** where it is declared. A **Feature_clause** begins with the keyword **feature** followed by an optional **Clients** part; if present, this **Clients** part lists the classes to which the feature is available.

← **Feature_clause** was introduced in [5.7, page 137](#).

If there is no **Clients** part, then every feature introduced in the **Feature_clause** is available to any client that cares to use it. So if class **PARAGRAPH** includes a **Feature_clause** of the form

The **feature** keyword is not technically part of the **Feature_clause**, but introduces it.



```
feature
  indent (n: INTEGER)
    -- Indent paragraph by n positions.
    ... Procedure body omitted ...
```

then any other class may declare an entity *p* of type **PARAGRAPH** and include a call such as

```
p.indent (5)
```

If, however, the **Clients** part of a **Feature_clause** is present, it consists of a list of classes in braces, and makes the features introduced by the clause available only to those classes and their descendants.

Here is such a **Feature_clause**, appearing in a class **LINKABLE** and listing three clients:



```
feature {LINKABLE, LINKED_LIST, TWO_WAY_TREE}
  right: like Current
  put_right (other: like Current)
    --Make other right neighbor of this object.
    ... Procedure body omitted...
```

This `Feature_clause` introduces two features, `right` and `put_right`, and makes them available to clients `LINKABLE` (the class itself, viewed as its own client), `LINKED_LIST` and `TWO_WAY_TREE`. This means that, for `l` of type `LINKABLE`, calls of the form

```
l.right
l.put_right (...)
```

are permitted if they appear in descendants of any of the classes `LINKABLE`, `LINKED_LIST` and `TWO_WAY_TREE`.

The next subsections explore two properties visible on this example:

- A class may need to make features available to itself.
- Making a feature available to a class also makes it available to all of its proper descendants.

Exporting to oneself

The above `Clients` part, appearing in class `LINKABLE`, listed `LINKABLE` itself among the classes to which `right` and `put_right` are available. This is required if the class contains a **qualified call** such as

```
l.put_right (...)
```

with `l` of a type based on `LINKABLE`. If the `Feature_clause` started with just **feature** `{LINKED_LIST, TWO_WAY_TREE}`, such a qualified call would be invalid outside of the two classes listed and their descendants; in particular, it would be invalid in `LINKABLE` itself.

The reason is clear: a qualified call `x.f(...)` always makes the enclosing class a client of `x`'s type; so the above call makes `LINKABLE` a client of itself, and if `LINKABLE` has made `put_right` selectively available to some clients only this will only be permitted if it has listed itself among them.



Although perhaps strange at first sight, this convention is consistent with the general rules on export. (Making exceptions for clients that happen to be the class itself, or one of its descendants, would lead to complicated rules.) Be sure to note, however, that all this only applies to qualified calls. There is no restriction, in the text of a class, on **unqualified calls** to features of the class itself, as with → *Chapter 23.*

```
put_right (...)
```


appearing in a routine of *LINKABLE*, with the semantics of calling *put_right* → “*Target of a call*”, [page 628](#). on the **current object**. This is always permitted regardless of the export status of *put_right* — that is, even if *put_right* appears in a *Feature_clause* whose *Clients* part does not include *LINKABLE*. Clearly, a secret or selectively available feature such as *put_right* would be useless if it couldn’t be called in this way from within the class. Unlike qualified calls such as *l.put_right(...)*, such an unqualified call is not considered to make the class a client of itself.



A general semantic property is that, except for invariant monitoring, an unqualified call *f(...)* will always have the same effect as the qualified call *Current.f(...)*. But as a result of this discussion the validity constraints are slightly different: if *f* is not exported to the class itself, the first form may be valid and the second one not.

Exporting to descendants

Making a feature available to a class also makes it available to the proper descendants of that class. This is because a class needs the same privileges that its parents had; for example, it could redefine an inherited routine, changing the original algorithm into a slightly different one, which still needs access to the same information from suppliers.

As a consequence, declaring features in a *Feature_clause* of the form

```
feature {ANY}
    ... Feature declarations ...
```

makes them available to all classes, since every developer-defined class is a descendant of *ANY*. Such a clause has the same effect as no *Clients* part at all, as in

```
feature
    ... Feature declarations ...
```

← “*ANY*”, [6.5, page 172](#); see also [chapter 35](#) for more details.

Making a feature secret

The export control mechanism as just described gives us, as a special case, the ability to make a feature *f* completely **secret** — available for call to no client. It suffices to declare the feature in a *Feature_clause* that starts\

```
feature {NONE}
    ... Declaration for f and other secret features ...
```

where *NONE* is the fictitious class at the bottom of the inheritance hierarchy. Because *NONE* has no usable instance, and no developer-written class can be a descendant of *NONE*, this makes it impossible for any class to use *f* as feature of a qualified call $x.f(\dots)$. ← “*NONE*”, 6.6, page 175.

The treatment of *ANY* and *NONE* for export controls is pleasantly symmetric: **feature** {*ANY*} introduces public features, available to all classes; **feature** {*NONE*} introduces private features, available to no class.

The conventions for shorthands are, however, different:

- **feature** with no *Clients* part is an abbreviation for **feature** {*ANY*}.
- **feature** { } with an empty *Clients* part is not permitted by the syntax: the production for *Clients* requires a *Class_list*, which cannot be empty.

→ Page 208 below.

feature { } could be accepted as a synonym for **feature** {*NONE*} (and actually was in earlier versions of Eiffel, although seldom used), but the language design has settled on a single convention, and chosen the more explicit one for clarity. It is not in the usual Eiffel style to use empty brace or parenthesis enclosures.



Adapting the export status of inherited features

The preceding discussion has explained the export status of features introduced in a class (although the formal definitions have not yet been given). We also need to know what happens to *inherited* features.

If a feature is redeclared, its new declaration will appear in a **Feature_** *clause*, whose *Clients* part, or absence thereof, will determine the export status as we have just seen. But what if the feature is not redeclared?

The rule is simple. By default, the feature will keep its export status. An heir can change that status, however, through a **New_exports** part, appearing as part of the **Feature_adaptation** subclause of a **Parent** part.

→ A class “redeclares” a feature if it provides a new declaration for it. This may be either a redefinition or an effecting. See chapter 10 for details.

As an example, here is the beginning (Notes clause excluded) of a class of EiffelBase:

```
class FIXED_STACK [T] inherit
  STACK [T]
inherit {NONE}
  ARRAY [T]
  rename
    put as array_put,
    ...Other renaming pairs omitted ...
  export
    {NONE} -- Implementation
    all
  end
feature
  ...
```

The `New_exports` part appears with the other possible subclasses of a `Feature_adaptation`: after `Undefine`, `Redefine` and `Rename` (only the last one present here) and before `Select`.

← All these subclasses, and the `Feature_adaptation` as a whole, are optional. The syntax appeared on page 171.

The `New_exports` subclass has the general form (shown here with the `export` keyword that introduces the subclass)

```
export
  {A, B, C} -- Feature category 1
  f1, f2, ...
  {X, Y} -- Feature category 2
  g1, g2, ...
  ...
```

meaning: unless a redeclaration specifies a different status, make *f1, f2, ...* available to clients *A, B, C* and any of their descendants; make *g1, g2, ...* available to clients *X, Y* and any of their descendants; and so on.

It is good, as illustrated, to include after each `Clients` list a header comment, such as `-- Implementation`, indicating the new feature category. The notion of feature category, and the recommendation to list it through a `Header_comment`, are derived from the practice of labeling feature clauses in a similar way.

← “FEATURES PART: EXAMPLE”, 5.5, page 134; syntax, page 137.

If, instead of a feature list such as *f1, f2, ...*, a `Clients` list is followed by the keyword `all`, then all non-redeclared inherited features are available to the given clients and their descendants, except for any features for which other parts of the subclass specify a different policy. For example, `FIXED_STACK` above hides all features inherited from `ARRAYED_LIST` from all clients, by exporting them to `NONE` only. This is a typical example of a class which inherits its interface from one parent (here `STACK`) and uses another parent (here `ARRAY`) for implementation purposes only.



If no part of the subclass mentions **all** in lieu of a feature list, any non-redeclared inherited feature that is not explicitly given a new export status keeps the exact export status that it had in the parent. Assuming class declarations of the form:



```

class B feature
  x: INTEGER
  feature {A}
    f, g, h: INTEGER
  feature {NONE}
    i, j, k: INTEGER
end

class C inherit
  B
  export
    {D} -- Implementation
        i, j
    {ANY} -- Access
        f
  end
end

```

the features of *C* have the following status:

- *x*, available to all clients, *h*, available to *A* and its descendants, and *k*, secret, do not appear in any of the **New_exports** subclauses: they both keep the status they had in *B*.
- *i* and *j*, regardless of their original status in *B*, are now available to *D* and its descendants.
- *f* is now available to all clients. (Re-exporting to *ANY* is how you make generally available a feature that was selectively available, or secret, in a parent.)

Expanding or restricting the export status



Elaborate changes of export status in inheritance, as in the last example, are uncommon. But two simpler cases causing the use of a **New_exports** clause do occur fairly often:

- *Extending*: you may want to re-export a feature which was used in the parent for implementation purposes only, but turns out to be of direct value for the clients of the new class, as with *f* in the last example.
- *Restricting*: in designing a new class, you may want to hide features that were exported by a parent.

The second case does not arise in the last example; it does appear in the previous one, for the inheritance of `FIXED_STACK` from `ARRAY`, which hides **all** inherited features. It is not by accident that the `Inherit_clause` in that case started with

```
inherit {NONE}
```

meaning, as we have seen, non-conforming inheritance. Restricting the export availability of a class is, indeed, applicable only to non-conforming inheritance, as it could cause type problems in the conformance case.

Extending the export status of an inherited feature is always possible, whether in conforming or non-conforming inheritance.

← “*NON-CONFORMING INHERITANCE*”, 6.8, page 180.

→ “*NOTES ON THE TYPE POLICY*”, 25.7, page 672.



The rule that defines this policy is not a validity constraint but instead a part of the semantics. The “client set” of a feature f of a class C — the set of classes that have access to f for qualified calls — is defined below as the union of all applicable `Clients` lists: the list governing its declaration or redeclaration in C , the `New_exports` if applicable, and the applicable lists from *conforming* parents. So with non-conforming inheritance you can override the original status as you please; but with a conforming parent, even though it is not invalid to write `export {NONE}`, this will have no effect since the `Clients` list `{NONE}` will be combined with the feature’s status in the parent, which will then remain applicable.

→ “*Client set of a Clients part*”, page 207.

The export status of features

The previous discussion allows us to give a precise definition of the **export status** of any feature, which will determine to what classes the feature is **available** for qualified calls. This notion determines the validity of

```
x.f(...)
```

or the equivalent using operator expressions or assignment procedure calls, appearing in a class C which declares x of type S : the feature of final name f in S must be available to B .

→ See chapter 25 on call validity. The precise requirement is condition 2 of export validity, page 632.

We first need a notion of “client set”, applying to `Clients` parts such as `{A, B, C}` which, as we have seen, may appear both at the beginning of a `Feature_clause` and in a `New_exports` subclass:



Client set of a Clients part

The **client set** of a `Clients` part is the set of descendants of every class of the universe whose name it lists.

By convention, the client set of an absent `Clients` part includes all classes of the system.

The descendants of a class include the class itself. The “convention” of this definition simplifies the following definitions in the case of no **Clients** part, which should be treated as if there were a **Clients** part listing just *ANY*, ancestor of all classes.



No validity rule prevents listing in a **Clients** part a name *n* that does not denote a class of the universe. In this case — explicitly permitted by the phrasing of the definition — *n* does not denote any class and hence has no descendants; it does not contribute to the client set.

This important convention is in line with the reuse focus of Eiffel and its application to component-based development. You may develop a class *C* in a certain system, where it lists some class *S* in a **Clients** part, to give *S* access to some of its features; then you reuse *C* in another system that does not include *S*. You should not have to change *C* since no bad consequence can result from listing a class not present in the system, as long as *C* does not itself use *S* as its supplier or ancestor.

Even in a single system, this policy means that you can remove *S* — if you find it is no longer needed — without causing compilation errors in the classes that list it in their **Clients** parts. With a stricter rule, you would have to remove *S* from every such **Clients** part. But then if you later change your mind — as part of the normal hesitations of an incremental design process — you would have to put it back in each of these places. This process is tedious, and it wouldn't take many iterations until programmers start making many features public just in case — hardly an improvement for information hiding, the purpose of all this.

Rules on setting the export status



(This section introduces no new concepts but gives a more formal presentation of ideas introduced above. You may skip it on first reading.)

→ Next section: “[DOCUMENTING THE CLIENT INTERFACE OF A CLASS](#)”, 7.9, [page 212](#)

The two constructs that determine the export status of a feature are **Clients** and **New_exports**. To conclude this discussion on export controls and information hiding, we need to express their precise syntax, constraints and semantics.

Here is the syntax of the **Clients** part:



Clients	$\text{Clients} \triangleq \{ \text{" Class_list " } \}$ $\text{Class_list} \triangleq \{ \text{Class_name " , " ... } \}^+$
----------------	---

This construct may appear in two positions. One is in a **New_exports**, as seen next; the other is as an optional component of a **Feature_clause**, as in

← *Feature_clause* was specified on page 137.



<pre>feature {A,B,C} ... Feature declarations ...</pre>

There is **no validity constraint** on **Clients** part. In particular, it is valid for a **Clients** part both:

- To list a class that does not belong to the universe.
- To list a class twice.



These properties may at first seem at odds with the language's emphasis on including validity constraints that permit detection of errors and inconsistencies at compile time. But in fact there is no adverse effect:

- As noted, permitting a **Clients** part in a class *C* to listing a non-exist class *S* gives us useful flexibility. Of course you may misspell a class name in a **Clients** part and, in the absence of any constraint, not get a validity error. But this is not really cause for concern: if you mean to export *f* to *A* in *C* and mistakenly start the **Feature_clause** with **feature** *{B}* instead of **feature** *{A}*, then for any call *cl.f(...)* with *cl* of type *C* in *A* you will get a validity error. So the absence of a constraint on the class names listed in a **Clients** part introduces no risk of accidentally violating information hiding requirements.

This policy contrasts with the **Class Type rule**, which addresses the only other possible use of a **Class_name** in the language: as part of a **Class_type**. There we will need, of course, to require that any class used as part of a type be part of the surrounding universe.

→ “*Class Type rule*”,
page 333.

- Similar reasoning explains why it is not invalid for a class to appear twice in a **Clients** part, as in *{A, A}*. Export privileges extend to descendants; so if we disallowed **feature** *{A, A}* we should also prohibit **feature** *{A, B}* if *B* is a proper descendant of *A*, since exporting to *A* also exports to *B*. Such a rule is too complicated for the benefits it brings.

Since there is no restriction on the classes listed in the **Class_list**, one of them may be the enclosing class or one of its ancestors, allowing the class, as noted earlier, to make a feature selectively available to the current class.

Now for **New_exports**. It is an optional element of **Feature_adaptation** in a **Parent** part, as illustrated by **FIXED_STACK** above, and has the following form:

← Syntax on page 171.



Export adaptation	
New_exports	\triangleq export New_export_list
New_export_list	\triangleq { New_export_item ";" ... } ⁺
New_export_item	\triangleq Clients [Header_comment] Feature_set
Feature_set	\triangleq Feature_list all
Feature_list	\triangleq { Feature_name ";" ... } ⁺

← The optional **Header_comment** indicates a feature category: see ,page 204.

A constraint applies to any **New_exports** clause:



Export List rule

VLEL

A **New_exports** clause appearing in class C in a Parent part for a parent B , of the form

export

$\{class_list_1\} feature_set_1$

...

$\{class_list_n\} feature_set_n$

is valid if and only if for every $feature_set_i$ (for i in the interval $1..n$) that is a **Feature_list** (rather than **all**):

- 1 • Every element of the list is the final name of a feature of C inherited from B .
- 2 • No feature name appears more than once in any such list.

SEMANTICS

To obtain the export status of a feature, we need to look at the **Feature_clause** that introduces it if it is immediate, at the applicable **New_exports** clause, if any, if it is inherited, and at the **Feature_clause** containing its redeclaration if it is inherited and redeclared. In a **New_exports**, the keyword **all** means that the chosen status will apply to all the features inherited from the given parent.

The following definitions and rules express these properties. They start by extending the notion of “client set” from entire **Clients** parts to individual features.

DEFINITION

Client set of a feature

The **client set** of a feature f of a class C , of final name $fname$, includes the following classes (for all cases that match):

- 1 • If f is introduced or redeclared in C : the client set of the **Feature_clause** of the declaration for f in C .
- 2 • If f is inherited: the union of the client sets (recursively) of all its precursors from conforming parents.
- 3 • If the **Feature_set** of one or more **New_exports** clauses of C includes $fname$ or **all**, the union of the client sets of their **Clients** parts.

This definition is the principal rule for determining the export status of a feature. It has two important properties:

- The different cases are cumulative rather than exclusive. For example a “redeclared” feature (case [1](#)) is also “inherited” (case [2](#)) and the applicable `Parent` part may have a `New_exports` (case [3](#)).
- As a result of case [2](#), **the client set can never diminish under conforming inheritance**: features can win new clients, but never lose one. This is necessary under polymorphism and dynamic binding to avoid certain type of “catcalls” leading to run-time crashes.

This is what “available”, used informally up to now, exactly means:



Available for call, available

A feature f is **available for call**, or just **available** for short, to a class C or to a type based on C , if and only if C belongs to the client set of f .

In line with others in the present discussion, the definition of “available for call” introduces a notion about *classes* and immediately generalizes it to *types* based on those classes.

The key validity constraint on calls, export validity, will express that a call $a.f(\dots)$ can only be valid if f is available to the type of a .

There is also a notion of “available for creation”, governing whether a `Creation_call` `create a.f (...)` is valid. “Available” without further qualification means “available for call”.

→ “*RESTRICTING CREATION AVAILABILITY*”, 20.7, page 539.

There are three degrees of availability, as given by the following definition.



Exported, selectively available, secret

The export status of a feature of a class is one of the following:

- 1 • The feature may be available to all classes. It is said to be **exported**, or **generally available**.
- 2 • The feature may be available to specific classes (other than *NONE* and *ANY*) only. In that case it is also available to the descendants of all these classes. Such a feature is said to be **selectively available** to the given classes and their descendants.
- 3 • Otherwise the feature is available only to *NONE*. It is then said to be **secret**.

This is the fundamental terminology for information hiding, which determines when it is possible to call a feature through a *qualified call* $x.f$. As special cases:

- A feature introduced by **feature** $\{NONE\}$ (case 3) is available to no useful classes.
- A feature introduced by **feature** $\{ANY\}$, or just **feature**, is available to all classes and so will be considered to fall under case 1.
- A feature introduced by **feature** $\{A, B, C\}$, where none of $\{A, B, C\}$ is *ANY*, falls under case 2.

*Or even **feature** $\{ANY, A, B, \dots\}$; adding classes after *ANY* brings nothing.*

A feature available to a class is also available to all the proper descendants of that class. As a consequence, selective export does not restrict reuse as much as it may seem at first: while the features will only be available to certain classes, these may be classes written much later, as long as they are descendants of one of the listed **Clients**.

7.9 -DOCUMENTING THE CLIENT INTERFACE OF A CLASS

Now that we have seen the details of the client and export mechanisms, we can obtain an answer to a central issue of software development, especially relevant to the component-based, reuse-oriented software culture promoted by Eiffel: how can the author of a class provide authors of client and descendant classes, and maintainers of the class itself, with a clear description of the facilities offered?

Selecting features

A class will be documented through its features (as well as other properties such as the class invariant and the list of its parents). The first question is: which features do we show? Not necessarily all of them: for example, a client class needs only the features available to it, while a descendant has access to all features. Also, we might consider inherited features, or not. These observations suggest two orthogonal distinctions:

- Between views relevant to authors of: the class itself; all clients; a specific client (taking into account selective exports as discussed earlier in this chapter); proper descendants.
- Between views that take into account: *immediate* features and invariant clauses (those from the class itself) only; *inherited* ones as well.

Retaining the useful combinations gives the following ways of selecting features to document:

Available to all clients	Available to client X	Available to descendants (all features)
--------------------------	-------------------------	---

Immediate + redeclared	Incremental view	<i>X</i> -client incremental view	
All including inherited features (“flat views”)	Client view	<i>X</i> -client view	Descendant view

In the first row, we select not only immediate features but also those inherited from a parent and *redeclared*. This is for two reasons:

- The combination of immediate and redeclared features gives us a good idea of the “added value” of the class: what it adds to its parents’ features. The term *incremental view* expresses this notion.
- More prosaically, such a view is easy to produce: a simple parsing tool, working on the basis of just one class without having to access its proper ancestors, can process all feature declarations — not having to differentiate between new declarations and redeclarations — and, on the basis of **Clients** parts, retain the public ones (incremental view) or those available to a specific client (*X*-client view).

It would be possible to spot the redefinitions (by analyzing the Redefine clauses) but not the effectings.

In the second row, we include all inherited features.

Contract views

Once we have selected the features to document, what information do we retain for them? The most obvious answer would be to give the source code. But this is usually not appropriate: for a “client programmer” (author of a client class), the class text usually includes implementation details along with interface properties. The principle of information hiding requires that we include only the latter. For the view to be offered to a client programmer, the interface properties include:

- The name of a feature.
- Its signature: types of arguments and result if any.
- The **contracts**: precondition, postcondition.
- Some properties of the class other than its features, in particular the class invariant.

Introducing this new dimension into our classification gives the following variant of the previous table, again retaining useful combinations only:

	Available to all clients, contracts only	Available to client <i>X</i> , contracts only	Available to descendants (all features), source text
Immediate + redeclared	Incremental contract view	<i>X</i> -client incremental contract view	

All including inherited features (“flat views”)	Contract view	X-client contract view	Descendant view
---	---------------	------------------------	-----------------

The leftmost column yields the most interesting form of documentation for Eiffel classes: the **contract view**, incremental or not.

Here are the precise definitions leading to this notion. The basic ideas are in the preceding discussion, but the definitions need to take all cases and details into account.

Secret, public

A property of a class text is **secret** if and only if it involves any of the following, describing information on which client classes cannot rely to establish their correctness:

- 1 • Any feature that is not available to the given client, unless this is overridden by the next case.
- 2 • Any feature that is not available for creation to the given client, unless this is overridden by the previous case.
- 3 • The body and rescue clause of any feature, except for the information that the feature is external or **Once** and, in the last case, its once keys if any.
- 4 • For a query without formal arguments, whether it is implemented as an attribute or a function, except for the information that it is a constant attribute.
- 5 • Any **Assertion_clause** that (recursively) includes secret information.
- 6 • Any parent part for a non-conforming parent (and as a consequence the very presence of that parent).
- 7 • The information that a feature is frozen.

Any property of a class text that is not secret is **public**.

Software developers must be able to use a class as supplier on the basis of public information only.

A feature may be available for call, or for creation, or both (cases **1** and **2**). If either of these properties applies, the affected clients must know about the feature, even if they can use it in only one of these two ways.

Whether a feature is external (case **3**) or constant (case **4**) determines whether it is possible to use it in a **Non_object_call** and hence is public information.

These notions yield the definition of the incremental contract view:

Incremental contract view, short form

The **incremental contract view** of a class, also called its **short form**, is a text with the same structure as the class but retaining only public properties.

Eiffel environments usually provide tools that automatically produce the incremental contract view of a class from the class text. This provides the principal form of software documentation: abstract yet precise, and extracted from the program text rather than written and maintained separately.

The definition specifies the information that the incremental contract view must retain, but not its exact display format, which typically will be close to Eiffel syntax.

Below is an extract — beginning, middle and end — from the incremental contract view of the *HASH_TABLE* class of EiffelBase, displayed at the click of a button by Eiffel Software's EiffelStudio (5.6) running on Windows.

*An
incremental
contract view
(extracts)*

As you will note from this example, the view relies on syntactic conventions slightly different from those of Eiffel; for example, it uses **class interface** instead of the Eiffel keyword **class**. This avoids any confusion with actual Eiffel, since a short form is not a class *text* but class *documentation*. The above definition indeed leaves environments such freedom as to the exact appearance of such views; it only specifies which information to retain and which to discard.

The next definition introduces the non-incremental variant:

DEFINITION

Contract view, flat-short form

The **contract view** of a class, also called its **flat-short form**, is a text following the same conventions as the incremental contract view form but extended to include information about inherited as well as immediate features, the resulting combined preconditions and postconditions and the unfolded form of the class invariant including inherited clauses.

The contract view is the full interface information about a class, including everything that clients need to know (but no more) to use it properly. The “combined forms” of preconditions and postconditions take into account parents’ versions as possibly modified by **require else** and **ensure then** clauses, and hence describing features’ contracts as they must appear to the clients. The “unfolded form” of the class invariant includes clauses from parents. In all these, of course, we still eliminate any clause that includes secret information, as with the incremental contract view.

The contract view is the principal means of documenting Eiffel software, in particular libraries of reusable components. It provides the right mix of abstraction, clarity and precision, and excludes implementation-dependent properties. Being produced automatically by software tools from the actual text, it does not require extra effort on the part of software developers, and stands a much better chance to remain accurate when the software changes.

The contract views, incremental and full, are a fundamental tool for many aspects of software construction. By providing the right level of abstraction to talk about classes, they constitute the Eiffel method’s technique of choice for discussing class designs and documenting reusable components. The resulting documentation is free — no need to hire a technical writer, since environment tools take care of producing the document — and, being extracted from the class text, is not subject to the major risk of software documentation, the *reverse Dorian Gray phenomenon*: ceasing to be truthful as the software evolves.

Routines

8.1 OVERVIEW

Routines describe computations.

Syntactically, routines are one of the two kinds of feature of a class; the other kind is attributes, which describe data fields associated with instances of the class. Since every Eiffel operation applies to a specific object, a routine of a class describes a computation applicable to instances of that class. When applied to an instance, a routine may query or update some or all fields of the instance, corresponding to attributes of the class. → Chapter 18 explores attributes.

A routine is either a procedure, which does not return a result, or a function, which does. A routine may further be declared as **deferred**, meaning that the class introducing it only gives its specification, leaving it for descendants to provide implementations. A routine that is not deferred is said to be **effective**.

An effective routine has a **body**, which describes the computation to be performed by the routine. A body is a **Compound**, or sequence of instructions; each instruction is a step of the computation.

The present discussion explores the structure of routine declarations, ending with the list of possible various forms of instructions.

8.2 ROUTINE DECLARATION

A routine declaration describes the interface of a routine and, unless the routine is deferred, its implementation.

Here are two routine declarations; *total* is a function, *move* a procedure.



```
total: INTEGER
    -- Sum of attributes a, b and c
deferred
ensure
    summed: Result = a + b + c
end
```

```
move (mice: MOUSE; men: MENU)
    -- Move mouse cursor to first item in menu.
require
    men_exist: men /= Void
do
    mice.move (men)
end
```



It is not necessary to repeat the name of the routine as an ending comment, writing for example **end** -- *move*, as an earlier convention suggested. Most routine texts in well-written Eiffel texts are short, so the ending comment tends to obscure, not help. You may still use an ending comment for the occasional long routine.



A **Feature_declaration**, as you will remember, declares a routine if and only if it satisfies the following condition:

- There is a **Feature_value** including an **Attribute_or_routine**, whose **Feature_body** is of the **Deferred** or **Effective_routine** kind.

The **Formal_arguments** and **Type_mark** parts may or may not be present. If the **Query_mark** is present, the declaration describes a function; otherwise it describes a procedure.

As with any other feature, a routine declaration may include more than one routine name, as in the following declaration of three procedures:



```
proc2, proc3, proc4 (x, y: REAL)
require
    x > y
do
    print (sqrt (x - y))
end
```

The **meaning**, as in the general case of “synonym” features, is the same as that of three separate declarations with identical **Declaration_body**.

The routines remain otherwise independent; in particular, **redefining** or **renaming** one in a descendant does not affect the others.

← “[HOW TO RECOGNIZE FEATURES](#)”, 5.12, page 145.

← “[SYNONYMS AND MULTIPLE DECLARATION](#)”, 5.18, page 159.

→ Renaming: chapter 6
:redefinition: chapter 10.

8.3 FORMAL ARGUMENTS



A routine may have arguments, corresponding to information that callers will pass to every execution of the routine.



Formal argument, actual argument

Entities declared in a routine to represent information passed by callers are the routine's **formal arguments**.

The corresponding expressions in a particular call to the routine are the call's **actual arguments**.

Rules on Call require the number of actual arguments to be the same as the number of formal arguments, and the type of each actual argument to be compatible with (conform or convert to) the type of the formal argument at the same position in the list.

→ Chapter 25. See 8.4, page 221 below about achieving the effect of a variable number of arguments.

A note on terminology: Eiffel always uses the term **argument** to refer to the arguments of a routine. The word “parameter” is never used in this context, because it could create confusion with the types that can parameterize *classes*, called **generic parameters**.

← About genericity see chapter 12.

Function *total* seen earlier has no arguments. Procedure *move* has two formal arguments called *mice* and *men*. Assuming both *move* and *total* appear in a class *C*, instructions using typical calls to these routines, appearing in some routine of a class *B*, might be



```
c1.move (mo, me)
n := c1.total
```

with *c1* of type *C*, *mo* of type *MOUSE*, *me* of type *MENU*, *n* of type *INTEGER*. Expressions *mo* and *me* are the actual arguments of the first call.

The formal arguments of *move* were all of different types. As with feature names in a Feature_declaration, you may group two or more formal arguments of the same type into an Entity_declaration_group. The comma serves as separator, as in this routine from class *TWO_WAY_LIST* in EiffelBase:



```
update_after_deletion
    (one, other: like first_element; index: INTEGER)
    ... Rest of routine omitted ...
```

This declares both *one* and *other* as being of type **like** *first_element*. The effect would have been identical with a routine header of the form

```
update_after_deletion
(one: like first_element;
 other: like first_element;
 index: INTEGER)
```

The preceding examples illustrate the general form of the **Formal_arguments** part of a routine declaration.



Formal argument and entity declarations

```
Formal_arguments  $\triangleq$  "(" Entity_declaration_list ")"
Entity_declaration_list  $\triangleq$  {Entity_declaration_group ";" ... }+
Entity_declaration_group  $\triangleq$  Identifier_list Type_mark
Identifier_list  $\triangleq$  {Identifier "," ... }+
```

As with other semicolons, those separating an **Entity_declaration_group** from the next are optional. The [style guidelines](#) suggest including them for successive declarations on a line, as with short formal argument lists, but omitting them between successive lines, as with local variable declarations (also covered by **Entity_declaration_group**).

A validity constraint mandates a choice of name avoiding any ambiguity:



Formal Argument rule VRFA

Let *fa* be the **Formal_arguments** part of a routine *r* in a class *C*. Let *formals* be the concatenation of every **Identifier_list** of every **Entity_declaration_group** in *fa*. Then *fa* is valid if and only if no **Identifier** *e* appearing in *formals* is the final name of a feature of *C*.

Another rule, given later, applies the same conditions to names of **local variables**. Permitting a formal argument or local variable to bear the same name as a feature could only cause confusion (even if we had a scoping rule removing any ambiguity by specifying that the local name overrides the feature name) and serves no useful purpose.

→ [“LOCAL VARIABLES AND RESULT”, page 225.](#)



The standard Eiffel style suggests different conventions anyway for features and formal arguments. Features, which have a wide scope (meaning that they can be used throughout a class and all its descendants), must have clear, meaningful names, typically made of one or more full words, separated, if more than one, by underscores, as in *spouse_name* (but not overqualified by the class name: if this feature appears in a class **EMPLOYEE**, do not call it *employee_spouse_name* as this would be redundant). For a formal argument, which has a small scope — most routines in Eiffel are short — the declaration of the argument and its uses will seldom be more than a few lines apart; you should choose short, simple names. Abbreviations are perfectly all right, as in *update_price* (*r*: **RATE**; *pc*: **PROMOTION_CODE**).

Being too pompous about names of formal arguments may *decrease* readability by giving arguments more attention that they deserve. Features are the aristocracy of a class and deserve full glory; formal arguments (and local variables) are their servants and should not try to shine above their rank.

Beginners sometimes use names of the form `a_TYPE_NAME`, as in `raise_salary(a_rate: RATE; a_promotion_code: PROMOTION_CODE)`. Seasoned Eiffel developers consider this revolting kitsch.

See “Further reports of abominable taste in the provinces”, in Proc. BISTOORI (45th Intl. Conf. on Biedermeier Influences on the Style of Typical Object-Oriented Retrograde Implementations), Sochi, 2004, pp. 2045-3497.

Complementing the Formal Argument rule is a general rule — also applicable to local variables, studied later in this chapter — that precludes using the same identifier twice in an `Entity_declaration_list`. Clearly, in



```
x: T1
x, y, y: T2
```

WARNING: not valid!

the type of `x` would be ambiguous. The type of `y` would not be ambiguous since the two occurrences are part of the same `Entity_declaration_group`, but the duplicate listing of `y` is invalid all the same; it can serve no useful purpose. This is the only condition on an `Entity_declaration_list`:



Entity Declaration rule VRED

Let `el` be an `Entity_declaration_list`. Let `identifiers` be the concatenation of every `Identifier_list` of every `Entity_declaration_group` in `el`. Then `el` is valid if and only if no `Identifier` appears more than once in the list `identifiers`.

8.4 USING A VARIABLE NUMBER OF ARGUMENTS



From the above syntax, and the previewed constraint on valid calls, it follows that every routine has a fixed number of arguments, which is the number of entities appearing in the `Entity_declaration_list` of its `Formal_arguments` part.



These rules do not prevent you from obtaining the effect of routines with variable numbers of arguments if you so desire. If the arguments are of arbitrary types, you may replace by a single argument of type `TUPLE`, corresponding to a sequence of arbitrary values, as in

→ See chapter 13 about tuples.

```
write_formatted (values: TUPLE; format: STRING)
    -- Print all elements of values, under given format.
    ...See below about the procedure body ...
```

which you can then call with a “Manifest tuple” consisting of a sequence of values in brackets:

```
write_formatted ([your_integer, your_string, your_real],
                your_output_format)
```

The procedure body will analyze the successive tuple elements and their types. The discussion of tuples shows how to write it.

→ “*Emulating a variable number of arguments*”, page 380.

If all the items are of types conforming to a known *T* you can, as an alternative to tuples, use `ARRAY [T]` as the argument type. A routine to print numeric values could read

```
write_numerics ( values: ARRAY [ NUMERIC ] )
    -- Print all elements of values.
    ...Procedure body omitted ...
```

where a typical call appears as:

```
write_numerics ( { ARRAY [ NUMERIC ] } [ your_integer, your_real ] )
```

Here we are passing an *INTEGER* and a *REAL*; both of these types conform to *NUMERIC*. The manifest array passed as argument is a tuple converted into an array.

8.5 ROUTINE BODY

A `Feature_body` had three possible forms:

← This syntax appeared first on page 143.

```
Feature_body  $\triangleq$  Deferred | Effective_routine | Attribute
```

The last case will be studied in the discussion of attributes. Routines correspond to the first two cases:



Routine bodies

```
Deferred  $\triangleq$  deferred
Effective_routine  $\triangleq$  Internal | External
Internal  $\triangleq$  Routine_mark Compound
Routine_mark  $\triangleq$  do | Once
Once  $\triangleq$  once [ "(Key_list)" ]
Key_list  $\triangleq$  { Manifest_string ", " ... }+
```

A `Feature_body` of the first possible form, `Deferred`, consists of the sole keyword `deferred`; this indicates that the routine, and as a consequence the enclosing class, are deferred.

→ See “*DEFERRED FEATURES*”, 10.11, page 272 and subsequent sections.

A routine of the other form, *Effective_routine*, may be *External*, indicating that it is implemented in another language. In the remaining and by far the most common case, *Internal*, the routine body is a *Compound*: a sequence of instructions describing the algorithm to be executed (after initialization of any local variables including *Result* for a function) on a call to the routine.

→ *Compound and other control structures are the topic of chapter 17.*

The introductory keywords **do** and **once** of an *Internal* body correspond to different semantics for calls to the routine:

→ For details: "*PRECISE CALL SEMANTICS*", 23.17, page 652.

- With a **do** body the initialization and body are executed anew for each call.
- If routine *o* of class *C* has a **once** body (*o* is then called a “once routine”), the initialization and body are executed only for the first call to *o* applied to an instance of *C* during any given session. For every subsequent call on an instance of *C* during the same session, the routine call has no effect; if the routine is a function, the value it returns is the same as the value returned by the first call. Once routines are useful for “smart initialization” actions which must be applied the first time a certain structure is accessed, and for shared information. They help avoid the global variables of conventional programming languages.

→ "*ONCEROUTINES*", 23.14, page 641.

You can fine-tune the meaning of “once” by including *once keys* after the keyword **once**, as in **once** ("*THREAD*") to specify that the execution will take place once in each thread. Other predefined values include "*PROCESS*" (the default) and "*OBJECT*" (to require computation once for every instance). You can even define your own once keys and then reset the key, through *onces.reset* ("*YOUR_KEY*"), ensuring that the next call to any once routine using this key will execute its body again. The mechanism also makes it possible to define variable keys to be set from outside the Eiffel text proper, for example in an Ace file.

It is convenient to introduce precise terms:

Once routine, once procedure, once function

A **once routine** is an *Internal* routine *r* with a *Routine_mark* of the *Once* form.

If *r* is a procedure it is also a **once procedure**; if *r* is a function, it is also a **once function**.

Here is an example procedure (from the EiffelTime library) with all the optional components except **Obsolete** and **Rescue** clauses:



```

make_fine (h, m: INTEGER; s: DOUBLE)
    -- Set hour, minute and second to h, m and integer part of s;
    -- Set fractional_second to fractional part of s.
    require
        correct: is_correct_time (h, m, s, False)
    local
        s_trunc: INTEGER
    do
        s_trunc := s.truncated_to_integer
        fractional_second := s - s_trunc
        make (h, m, s_trunc)
    ensure
        hour_set: hour = h
        minute_set: minute = m
        fine_second_set: fine_second = s
    end

```

The various components and their respective roles are the following. All components except the **Feature_body** and the final **end** are optional.

- The text appearing immediately after the routine name and arguments, starting with **--**, is a **Header_comment** explaining the purpose of the routine. Other comments may be inserted at the end of any line; but this one has a special role, documenting the routine's interface. The "contract view" of a class retains header comments.
- The keyword **require** introduces an **Assertion**, called the **Precondition** of the routine. This expresses the conditions under which a call to the routine is correct. Here *is_correct_time* must be true for the arguments given. The **Identifier** *correct* is a label identifying that assertion.
- The **Local_declarations** clause, studied below, declares local variables used only within the routine body, and initialized anew on each call. Here *make_fine* uses a local variable *s_trunc* of type *INTEGER*.
- The **Feature_body** is here of the **Effective** kind, more specifically **Internal**, starting with **do** (the other possibility is **once**) and continuing with instructions — zero or more in the general case, here three.
- The keyword **ensure** introduces another **Assertion**, the **Postcondition** of the routine. This expresses the conditions that a routine call will ensure on return if called in a state satisfying the precondition. Here it states that a number of queries have been set from the values of the arguments.

← "DOCUMENTING THE CLIENT INTERFACE OF A CLASS", 7.9, page 212.

→ Chapter 9

→ Chapter 9



The example does not include a **Rescue** clause. If present, this describes what to do if an exception occurs during an execution of the routine. The absence of a **Rescue** clause has the same effect as the presence of a **Rescue** clause just consisting of a call to the procedure `default_rescue` of the universal class **ANY**. So the example routine could have been written equivalently as

→ The **Rescue** clause and procedure `default_rescue` are discussed in detail in chapter 26.

```
make_fine (h, m: INTEGER; s: DOUBLE)
    ... All other clauses as above ...
    rescue
        default_rescue
    end
```

In its original form, `default_rescue` has a null effect, but a class may redefine it to provide specific exception handling.

8.6 LOCAL VARIABLES AND RESULT



If present in a routine, a **Local_declarations** clause is the declaration of variable entities available only within the **Feature_body**; they are useful for the computation it describes, but their values do not need to be retained by the current object after a call to the routine.

The last example introduced just one local variable:



```
local
    s_trunc: INTEGER
```

used in the **do** clause to hold the value of `s.truncated_to_integer`, needed by two of the instructions.

In this example there is no need for an automatic initialization of the variable since the first instruction of the routine assigns it a value. Such an explicit assignment is not required; if the routine's execution accesses the value of the variable when it has not been assigned, **initialization rules** guarantee a well-defined initial value, for example 0 for integers and False for booleans.

→ “*Default Initialization rule*”, page 516.

The general structure of a **Local_declarations** clause is:



```
Local_declarations ≙ local [Entity_declaration_list]
```

The **Entity_declaration_list** may be absent, as we tolerate an empty **local** part — perhaps while you are refactoring your software and moving local variable declarations in and out.

In addition to the earlier constraint requiring all identifiers in an Entity_declaration_list to be different, we must avoid any ambiguity between local variables and features of the class: ← “VRED”, page 221



Local Variable rule

VRLV

Let *ld* be the Local_declarations part of a routine *r* in a class *C*. Let *locals* be the concatenation of every Identifier_list of every Entity_declaration_group in *ld*. Then *ld* is valid if and only if every Identifier *e* in *locals* satisfies the following conditions:

- 1 • No feature of *C* has *e* as its final name.
- 2 • No formal argument of *r* has *e* as its Identifier.

Most of the rules governing the validity and semantics of declared local variables also apply to a special predefined entity: **Result**, which may only appear in a function or attribute, and denotes the value to be returned by the function. The following definition of “local variable” reflects this similarity.



Local variable

The local variables of a routine include all entities declared in its Local_declarations part, if any, and, if it is a query, the predefined entity **Result**.

Result can appear not only in the Compound, Postcondition or Rescue of a function or variable attribute but also in the optional Postcondition of a constant attribute, where it denotes the value of the attribute and allows stating abstract properties of that value, for example after a redefinition. In this case execution cannot change that value, but for simplicity we continue to call **Result** a local “variable” anyway.

When applying validity and semantics rules, you must treat **Result** as an entity of the type declared for the enclosing function’s result. For example, this function from class CLOSED_FIGURE in EiffelVision treats *Result* as a local variable of type INTEGER:



```
fill_style_count: INTEGER
-- Number of defined fill styles for this figure
do
  Result := global_fill_style_count + local_fill_style_count
end
```


8.7 EXTERNALS

A routine may have a **Feature_body** of the **External** form, which means that its implementation is written in another language.

The following examples illustrate the form of an **External** body:



```

open_file (file_od: INTEGER; mode: CHARACTER)
    -- Open file_od in mode mode.
    require
        file_status (file_od) <= 0
    external
        "C"
    end

```

```

file_status (file_od: INTEGER): INTEGER
    -- Current status of file associated with file_od
    external
        "C"
    alias
        "_fstat"
    end

```

They enable other Eiffel elements to call a C procedure under the Eiffel name *open_file* and a C function under *file_status*.



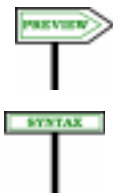
Such routines are viewed by the rest of a system as normal Eiffel routines; their only special property is that their execution, instead of being under the control of the Eiffel system to which they belong, is a call to some code generated by a compiler for the foreign language.

The second external routine of the example, a function, has a subclass of the form **alias** *external_name*, indicating that this function will be known through an Eiffel name, *file_status*, different from its name in the foreign language; by default the two would be the same. Here an alias is required since the C name, *_fstat*, begins with an underscore and so is not a valid Eiffel identifier.

The **External** mechanism include a wide set of possibilities; in particular, you may include inline C code directly, through the **alias** clause. A later chapter is entirely devoted to this mechanism.

→ Chapter 31.

8.8 TYPES OF INSTRUCTIONS



The **Internal** body of a non-deferred routine is a **Compound**, or sequence of instructions. As an introduction to the detailed study of instructions in ter chapters, here is an overview of the available variants. The syntax is:

Instructions	
Compound	\triangleq { Instruction ";" ... }*
Instruction	\triangleq Creation_instruction Call Assignment Assigner_call Conditional Multi_branch Loop Debug Precursor Check Retry

A **Compound** is a possibly empty list of instructions, to be executed in the order given. In the various parts of control structures, such as the branches of a **Conditional** or the body of a **Loop**, the syntax never specifies **Instruction** but always **Compound**, so that you can include zero, one or more instructions. → *Chapter 20.*

A **Creation instruction** creates a new object, initializes its fields to default values, calls on it one of the creation procedures of the class (if any), and attaches the object to an entity.

Call executes a routine. For the **Call** to yield an instruction, the routine must be a procedure. → *Chapter 23.*

Assignment changes the value attached to a variable. → *Chapter 22.*

An **Assigner call** is a procedure call written with an assignment-like syntax, as in $x.a := b$, but with the semantics of a call, as just a notational abbreviation for $x.set_a(b)$ where the declaration of a specifies an assigner command set_a . → *"ASSIGNER CALL", 22.12, page 607.*

Conditional, **Multi_branch**, **Loop** and **Compound** describe **control structures**, made out of instructions; to execute a control structure is to execute some or all of its constituent instructions, according to a schedule specified by the control structure. → *Control structures and Debug: chapter 17.*

Debug, which may also be considered a control structure, is used for instructions that should only be part of the system when you enable the *debug* compilation option.

Precursor enables you, in redefining a routine, to rely on its original implementation. → *Precursor: 10.24, page 299.*

Check is used to express that certain assertions must hold at certain moments during run time. → *Assertions and Check: chapter 9.*

Retry is used in conjunction with the exception handling mechanism. → *Exceptions and Retry: chapter 26.*

Correctness and contracts

9.1 OVERVIEW

Eiffel software texts — classes and their routines — may be equipped with elements of formal specification, called **assertions**, expressing correctness conditions.

Assertions play several roles: they help in the production of correct and robust software, yield high-level documentation, provide debugging support, allow effective software testing, and serve as a basis for exception handling. With advances in formal methods technology, they open the way to proofs of software correctness.

Assertions are at the basis of the **Design by Contract** method of Eiffel software construction.

This chapter describes assertions and the resulting notion of correctness of a class. It also specifies how the supporting development environment should help check correctness conditions at run time.

9.2 WHY ASSERTIONS?

One could write entire systems without assertions. Some Eiffel developers are even rumored to have done so. In fact, assertions have no effect on the semantics of correct systems — in theory, the only one that matters.

Do not look, however, for a SHORTCUT sign suggesting that you skip this chapter on first reading. Assertions are a key element of software development in Eiffel and omitting them would be renouncing a major benefit of the method.

Assertions serve to express the specification of software components: indications of *what* a component does rather than *how* it does it. This is essential information for building the components so that they will perform reliably, for using the components, and for validating them.

"Deviant Eiffel Programmers", in Proceedings of RACOON 13 (Report of Annual Conference on Object-Oriented Neuropsychiatry), Tahiti, 2005, pages 3456-3542.



The classes of the EiffelBase Library provide many examples of the use of assertions to express abstract properties of classes and routines. Consider the many descendants of class *CHAIN*, describing sequential data structures ("chains") such as lists. They enable clients to manipulate a cursor, allowed to go one position off the right and left edges of a chain, but no further. An assertion occurring in the **class invariant** of the corresponding classes expresses this property:

$$0 \leq \text{index}; \text{index} \leq \text{count} + 1$$

where *index* identifies the current cursor position, and *count* is the number of elements in the structure. The invariant must be guaranteed by every creation procedure of the class, and maintained by every exported routine.

In the same classes, a client may use exported procedures such as *start*, *finish*, *forth* and *back* to move the cursor. The bodies of these procedures depend on the implementation chosen (linked, array etc.) but many of their important properties are expressed by implementation-independent assertions, so that a version of *forth* will have the form

```
forth
    -- Move forward one position.
    require
        not_after: not after
    do
        ... Some appropriate implementation ...
    ensure
        moved: position = old position + 1
    end
```

The **require** and **ensure** clauses introduce a **precondition** and **postcondition**:

- The precondition states the condition under which *forth* is applicable: the cursor must not be "after" the right edge, as defined by the boolean function *after*. Any client calling *forth* must guarantee this condition.
- The postcondition states the property which the procedure must guarantee at the completion of any correct call: the cursor index will have been increased by one (the expression **old** position denotes the value of *position* as captured on routine entry). Any client may assume this condition after a call to *forth*.

The optional labels not_after and moved, specimens of Tag, serve as documentation and also for error messages. See below.

As these examples indicate, assertions are not instructions; they do not necessarily have an effect at execution time. Instead, they express properties that should be satisfied by the implementation. In other words, an assertion is a *description*, not a *prescription*.

Preconditions, postconditions, class invariants and other uses of assertions described below (in particular loop invariants) define **contracts** for the corresponding software elements: features, classes, loops.

For precise terminology: a *contract* is the specification of a software element; it is made of *assertions*. For example the contract of a feature comprises its precondition and postcondition; the contract of a class includes the contracts (as just defined) of all its features, plus the class invariant.

Contracts have important applications throughout software development. Among others:

- By helping developers to state precisely the formal properties of software elements, they enhance the correctness and reliability of the resulting software. The underlying theory of **Design by Contract**, views the construction of a software system as the fulfillment of many small and large contracts between clients and suppliers.
- Assertions may also be monitored at run time, providing a powerful tool for **testing** and **debugging** software.
- Contracts serve as the basis for automatic documentation tools, which produce abstract interface documentation of a class by extracting implementation-independent information from the class text.
- Rules on the fate of assertions in inheritance (invariant accumulation, precondition weakening, postcondition strengthening) provide a clear methodological framework for the proper use of inheritance and associated mechanisms of polymorphism and dynamic binding.

See “Object-Oriented Software Construction”.

← “-DOCUMENTING THE CLIENT INTERFACE OF A CLASS”, 7.9, page 212.

The first and second of these applications explain why the semantics of contracts, as defined later in this chapter, involves two aspects:

- The basic semantic role of contracts is to define the **correctness** of classes and their features. a class is correct if its implementation satisfies the contracts. Determining correctness is a matter of *mathematical proofs*, performed by people or by proof tools. Most of today’s development environments do not yet provide proof tools, but the notion of correctness is still essential to understand the role of contracts.
- Even in the absence of proofs, Eiffel environments must provide mechanisms for **run-time monitoring** of assertions, for the debugging and testing benefits just mentioned. That part of the semantics specifies when and how to evaluate assertion clauses during execution,

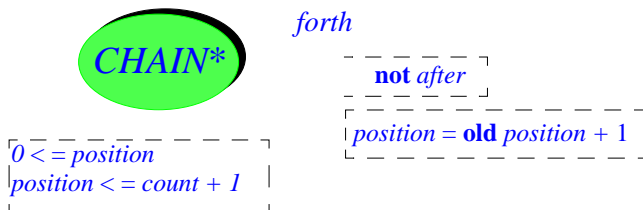
→ “THE CORRECTNESS OF A CLASS”, 9.12, page 252.

→ “RULES OF RUN-TIME ASSERTION MONITORING”, 9.13, page 253.

9.3 GRAPHICAL CONVENTION

The figure below illustrates the graphical representations that may be used to show routine preconditions, routine postconditions and class invariants.

For routines’ assertions each “drawer” figuratively opens on only one side: the routine obtains an input condition from the left drawer and delivers an output condition through the right drawer. The class invariant combines both conventions.



9.4 USES OF ASSERTIONS

Assertions appear in the following constructs:

- The Precondition and Postcondition parts of an **Attribute_or_routine**.
- The **Invariant** clause of a class.
- The **Check** instruction.
- The **Invariant** of a **Loop** instruction.

Attribute_or_routine:
page 143.

Class_declaration:
page 119.

Check: page 249.

Loop: page 495.

In addition, loops may have variants, which are integer expressions rather than assertions but play a closely related role.

All the constructs involving assertions are optional.

9.5 FORM OF ASSERTIONS

An **Assertion**, introduced by such keywords as **require** (for preconditions), **ensure** (for postconditions), **invariant** (for class and loop invariants) and **check** (for Check instructions) is made of one or more **Assertion_clause**, each based on a boolean expression, as in the precondition

```
require
  point_exists: ce /= Void
  positive_radius: ra > 0.0
```

from a routine with formal arguments *ce* and *ra*. This expresses that in a correct call to be correct the first argument must be non-void and the second argument must be positive. In this example each **Assertion_clause** is labeled by a **Tag** (**point_exists**, **positive_radius**).

The general form of an **Assertion**, and the syntax of constructs where it may appear, are:

A related construct is the loop Variant, described later in this chapter (9.14).

← On **Tag_mark**, see page =====.



Assertions	
Precondition	\triangleq require [else] Assertion
Postcondition	\triangleq ensure [then] Assertion [Only]
Invariant	\triangleq invariant Assertion
Assertion	\triangleq {Assertion_clause ";" ... }*
Assertion_clause	\triangleq [Tag_mark] Unlabeled_assertion_clause
Unlabeled_assertion_clause	\triangleq Boolean_expression Comment
Tag_mark	\triangleq Tag ":"
Tag	\triangleq Identifier



A **Boolean_expression**, as used in this syntax definition, is simply an Expression constrained to be of type **BOOLEAN**. → *Syntax on page 761 as part of the discussion of expressions.*

The semicolon is optional as **Assertion_clause** separator. This requires a non-production rule to avoid any ambiguity:

Syntax (non-production): Assertion Syntax rule

An **Assertion** without a **Tag_mark** may not begin with any of the following:

- 1 • An opening parenthesis "(".
- 2 • An opening bracket "[".
- 3 • A non-keyword **Unary** operator that is also **Binary**.

This rule participates in the achievement of the general Semicolon Optionality rule. Without it, after an **Assertion_clause** starting for example with the **Identifier** *a*, and continuing (case 2) with [*x*] it is not immediately obvious whether this is the continuation of the same clause, using *a* [*x*] as the application of a bracket feature to *a*, or a new clause that starts by mentioning the **Manifest_tuple** [*x*]. From the context, the validity rules will exclude one of these possibilities, but a language processing tool should be able to parse an Eiffel text without recourse to non-syntactic information. A similar issue arises with an opening parenthesis (case 1) and also (case 3) if what follows *a* is *-b*, which could express a subtraction from *a* in the same clause, or start a new clause about the negated value of *b*. The Assertion Syntax rule avoids this.

The rule does significantly restrict expressiveness, since violations are rare and will be flagged clearly in reference to the rule, and it is recommended practice anyway to use a **Tag_mark**, which removes any ambiguity.



Semicolons or not, the order of **Assertion_clause** components of an **Assertion** is significant. More precisely, the semantic specification below treats them as if they were separated by the **and then** binary boolean operator (replacing the semicolon if present). From that operator's own semantics this means that:

- The value of an **Assertion** is true if and only if every **Assertion_clause** in the **Assertion** has value true.
- If an **Assertion_clause** has value false, so has the whole **Assertion** in which it appears, even if the value of a subsequent clause is not defined.

→ *"SEMISTRIC
BOOLEAN OPERA-
TORS", 28.6, page 774.*

Because of the second of these properties, if an **Assertion_clause** b only makes sense when another, a , is true, you should write a before b . For example if you need a clause of the form $x.some_property$ and x is of a detachable type, and hence could be void, you should write the assertion as



```
x /= Void
x.some_property
```

where the first property implies that the second one, whether or not it holds, is meaningful. As another example, assuming an integer array a and an integer i , a routine precondition could read



```
require
  i >= a.lower
  i <= a.upper
  a [i] > 0
```

The last clause relates to the i -th item of a , defined only if i is within the array's bounds as expressed by the first two clauses (which we could also express more concisely as *valid_key* (i)). It would be incorrect here to reverse the order of clauses.

A few straightforward definitions are useful:

Precondition, postcondition, invariant

The **precondition** and **postcondition** of a feature, or the **invariant** of a class, is the **Assertion** of, respectively, the corresponding **Precondition**, **Postcondition** or **Invariant** clause if present and non-empty, and otherwise the assertion **True**.

So in these three contexts we consider any absent or empty assertion clause as the assertion **True**, satisfied by every state of the computation. Then we can talk, under any circumstance, of “the precondition of a feature” and “the invariant of a class” even if the clauses do not appear explicitly.

These are “local” assertions because unlike later “unfolded forms” they do not take into account the ancestor versions that, combined with the local clauses, will yield the full precondition, postcondition or invariant applicable to a feature or class.

→ “**REDECLARATION AND ASSERTIONS**”, 10.17, page 283, and also **UNFOLDING ASSERTIONS UNDER INHERITANCE** below.

Coming back to the syntax: an **Assertion_clause** may be preceded by a **Tag**, such as **point_exists** and **positive_radius** in the earlier example. Such tags are identifiers, with no validity constraint. They are useful for documentation purposes; in addition, they help produce precise error messages in the case, studied below, of run-time assertion monitoring. Although the **Tag** is optional and does not affect the semantics of correct programs, the style rule recommends including it for clarity.

← Page 232.

Finally, you will have noted that the syntax for `Unlabeled_assertion_clause` allows a clause that consists of just a `Comment`. This is useful for documenting conditions that you cannot or do not wish to express formally; for example:

```
require
  acyclic: -- The structure is not cyclic.
```

9.6 UNFOLDING ASSERTIONS UNDER INHERITANCE



Assertions have a close relationship with inheritance. In particular:

- The actual invariant applicable to a class includes, in addition to any clauses appearing in the class itself, all those from its ancestors.
- An inherited feature may change the assertions of its parent version, weakening the precondition through a `require else` clause and strengthening the postcondition through an `ensure then` clause.

These notions will be captured, in the discussion of feature adaptation, by the notion of *unfolded form* of an assertion, the result of including both immediate and inherited elements. Most of the semantic properties of contracts rely on the unfolded form. → “*Unfolded form of an assertion*”, page 287.

9.7 ASSERTIONS ON INDIVIDUAL FEATURES

We now look in more detail at the contracts governing a single feature. Class invariants, which apply to a whole class, will come next.

Preconditions and postconditions

The declaration of an `Attribute_or_routine` — deferred routine, effective routine with a `do` or `once` body, external routine, but also an attribute written with the explicit `attribute` keyword — may include a `Precondition`, a `Postcondition` or both.

Here is an example of a routine from class `CHAIN` in EiffelBase:



```
put_i_th (v: like first; i: INTEGER)
  -- Put item v at i-th position.
  require
    index_large_enough: i >= 1
    index_small_enough: i <= count
  deferred
  ensure
    not_empty: not is_empty
  end
```

The precondition expresses that no client should call the routine unless the actual argument for *i* is between 1 and *count*. The postcondition expresses that, after a successfully completed execution of the routine, *is_empty* (a boolean function of the same class) will yield false.

Each **Assertion_clause** of the **Precondition** and **Postcondition** has been labeled with a **Tag** such as `index_large_enough`.

The contract of a routine

A precondition and postcondition constitute a contract:



Contract, subcontract

Let *pre* and *post* be the precondition and postcondition of a feature *f*. The **contract** of *f* is the pair of assertions [*pre*, *post*].
A contract [*pre'*, *post'*] is said to be a **subcontract** of [*pre*, *post*] if and only if *pre* implies *pre'* and *post'* implies *post*.

Here “implies” is boolean implication. The notion of subcontract is important because it defines the clients’ perspective on permissible changes in a routine’s implementation. For a client of a class offering a routine



```

rout (...)
  require
    pre
  do
    ...
  ensure
    post
  end

```

what counts is the contract defined by the assertions: a client which achieves *pre* at call time is entitled to obtain *post* on return. Another routine *other_rou*t may be acceptable as a substitute for *rou*t if it still satisfies that contract. This does not necessarily mean that the contract of *other_rou*t must be the same as that of *rou*t: it may also be a subcontract [*pre'*, *post'*], since in that case any client’s call that satisfies the obligation defined by the original contract (*pre*) also satisfies the obligation of the new contract (*pre'*), and the benefits guaranteed by the new contract (*post'*) also entitle the clients to those guaranteed by the original (*post*).

The **constraint** on routine redeclaration will ensure that whenever a routine is redeclared (redefined, or effected if the original was deferred), the new specification is a subcontract of the original.

→ “**REDECLARATION AND ASSERTIONS**”, 10.17, page 283 and “**Redeclaration rule**”, page 313.

Constraints on routine assertions

For the contracts to be enforceable, preconditions and postconditions must satisfy constraints making them usable by clients.

The first constraint affects preconditions:



Precondition Export rule

VAPE

A **Precondition** of a feature r of a class S is valid if and only if every feature f appearing in every **Assertion_clause** of its unfolded form u satisfies the following two conditions for every class C to which r is available:

- 1 • If f appears as feature of a call in u or any of its subexpressions, f is available to C .
- 2 • If u or any of its subexpressions uses f as creation procedure of a **Creation_expression**, f is available for creation to C .

If (condition 1) r were available to a class B but its precondition involved a feature f not available to B , r would be imposing to B a condition that B would not be able to check for itself; this would amount to a secret clause in the contract, preventing the designer of B from guaranteeing the correctness of calls.

More on avoiding secret clauses below.

The rule applies to the *unfolded form* of a precondition, which will be defined as the fully reconstructed assertion, including conditions defined by ancestor versions of a feature in addition to those explicitly mentioned in a redeclared version.

The unfolded form (by relying on the “Equivalent Dot Form” of the expressions involved) treats all operators as denoting features; for example an occurrence of $a > b$ in an assertion yields $a.greater(b)$ in the unfolded form, where *greater* is the name of a feature of alias “>”. The Precondition Export rule then requires, if the occurrence is in a **Precondition**, that this feature be available to any classes to which the enclosing feature is available.

Condition 2 places the same obligation on any feature f used in a creation expression **create** $a.f(\dots)$ appearing in the precondition (a rare but possible case). The requirement in this case is “available for creation”.

The features mentioned in the constraint may include features of C and, for a complex precondition, features of other classes; for example, if the precondition of r , includes the expression

$$a.b(c) + d * (e.f(g).h)$$

where b, f, h are features of other classes, all these features must be available to B — as well as a, c, d , *plus alias* "+", *product alias* "*", e and g if all of these are features of C .



In addition to the Precondition Export rule:

- The Entity rule, studied as part of the [discussion](#) of entities, further restricts the kind of entities that may appear in an **Assertion**. For example **Result** may only appear in a postcondition. → "[Entity rule](#)", [page 513](#).
- If r is a creation procedure, the Creation Clause rule puts extra requirements on its precondition, expressed by the Creation Precondition rule and its notion of "creation-valid" precondition clause. → "[Creation Clause rule](#)", [page 548](#); "[Creation Precondition rule](#)", [page 547](#).

For postconditions, there is no rule corresponding to the Precondition Export rule; the Entity rule is sufficient. An **Assertion_clause** of a **Postcondition** may — unlike in a **Precondition** — refer to features with a different availability status. For example, the postcondition for routine *put_right* in class *LINKED_LIST* of EiffelBase includes the clause:

```
next.item = v
```



where v is a formal argument of the routine but the query *next* is only available to *LINKED_LIST* itself.



The reason for this difference of treatment between a **Precondition** and a **Postcondition** comes from the theory of Design by Contract. In the case of preconditions, as noted, using a secret query would make it impossible for clients to satisfy the contract. But including a secret query in a postcondition causes no harm to clients: they simply will not be able to rely on the corresponding properties, which indeed do not appear in the [contract view](#).

← "[DOCUMENTING THE CLIENT INTER-FACE OF A CLASS](#)", [7.9, page 212](#);

The difference in export status of the various entities involved in a precondition or postcondition explains the need for the following notion:



Availability of an assertion clause

An **Assertion_clause** a of a routine **Precondition** or **Postcondition** is **available** to a class B if and only if all the features involved in the Equivalent Dot Form of a are available to B .

This notion is necessary to define interface forms of a class adapted to individual clients, such as the incremental contract view ("short form").

“Old” expression

A special form of expression, the **Old** expression, is available in routine postconditions only. → Expressions are the topic of chapter 28.



In a postcondition clause, the expression **old** *exp* has the same type as *exp*; its value at execution time, on routine exit, is the value of *exp* as evaluated on routine entry in the current call.

An example appeared in the postcondition for *forth*. Here is another, ← Page 230. from routine *put* in class *FIXED_QUEUE* of EiffelBase; the routine inserts an element into a queue:



```

put (v: T)
    -- Add item v to queue.
    require
        not_full: not full
    do
        ...
    ensure
        count = old count + 1
        (old is_empty) implies (item = v)
        not empty
        array_item ((last - 1 + capacity) // capacity) = v
    end
  
```

// is the remainder operator on integers.

The highlighted postcondition clause indicates that if the queue was initially empty (**old** *is_empty*) the value at cursor position (given by *item*) will be the one just inserted. Operator **implies** is boolean implication.

The syntax of an **Old** expression is simply



“Old” postcondition expressions
 Old \triangleq **old** Expression

The validity constraint expresses that a **Postcondition** is the only permitted context for an **Old** expression:



Old Expression rule VAOX

An **Old** expression *oe* of the form **old** *e* is valid if and only if it satisfies the following conditions:

- 1 • It appears in a **Postcondition** part *post* of a feature.
- 2 • It does not involve **Result**.
- 3 • Replacing *oe* by *e* in *post* yields a valid **Postcondition**.



Result is otherwise permitted in postconditions, but condition 2 rules it out since its value is meaningless on entry to the routine. Condition 3 simply states that **old** e is valid in a postcondition if e itself is. The expression e may not, for example, involve any local variables (although it might include **Result** were it not for condition 2), but may refer to features of the class and formal arguments of the routine.

The semantic rule follows from the above informal explanations:

Old Expression Semantics, associated variable, associated exception marker

The effect of including an **Old** expression oe in a **Postcondition** of an effective feature f is equivalent to replacing the semantics of its **Feature_body** by the effect of a call to a fictitious routine possessing a local variable av , called the **associated variable** of oe , and semantics defined by the following succession of steps:

- 1 • Evaluate oe .
- 2 • If this evaluation triggers an exception, record this event in an **associated exception marker** for oe .
- 3 • Otherwise, assign the value of oe to av .
- 4 • Proceed with the original semantics.

The recourse to a fictitious variable and fictitious operations is in the style of “unfolded forms” used throughout the language description. The reason for these techniques is the somewhat peculiar nature of the **Old** expression, used at postcondition evaluation time, but pre-computed (if assertion monitoring is on for postconditions) on entry to the feature.

The matter of exceptions is particularly delicate and justifies the use of “associated exception markers”. If an **Old** expression’s evaluation triggers an exception, the time of that exception — feature entry — is not the right moment to start handling the exception, because the postcondition might not need the value. For example, a postcondition clause could read

$((x \neq 0) \text{ and } (\text{old } x \neq 0)) \text{ implies } (((1 / x) + (1 / (\text{old } x))) = y)$

If x is 0 on entry, **old** $x \neq 0$ will be false on exit and hence the postcondition will hold. But there is no way to know this when evaluating the various **Old** expressions, such as $1 / \text{old } x$ on entry. We must evaluate this expression anyway, to be prepared for all possible cases. If x is zero, this may cause an arithmetic overflow and trigger an exception. This exception should not be processed immediately; instead it should be remembered — hence the associated exception marker — and triggered only if the evaluation of the *postcondition*, on routine exit, attempts to evaluate the associated variable; hence the following rule.

The “associated variable” is defined only for effective features, since a deferred feature has no **Feature_body**. If an **Old** expression appears in the postcondition of a deferred feature, the rule will apply to effectings in descendants through the “unfolded form” of the postconditions, which includes inherited clauses.

Like any variable, the associated variable *av* of an **Old** expression raises a potential initialization problem; but we need not require its type to be self-initializing since the above rule implies that *ov* appears in a Certified Attachment Pattern that assigns it a value (the value of *oe*) prior to use.

→ “Self-initializing type”, page 515.

There remains to define precisely the value of the “associated variable”:

Associated Variable Semantics

As part of the evaluation of a postcondition clause, the evaluation of the associated variable of an **Old** expression:

- 1 • Triggers an exception of type **OLD_EXCEPTION** if an associated exception marker has been recorded.
- 2 • Otherwise, yields the value to which the variable has been set.

“Only” clause

With the **Old** expression we have a way to express the effect of a routine by specifying how some properties of the object after the routine’s execution relate to values captured before the execution.

This technique can be used in particular to express that the values of some queries (attributes or functions) do *not* change through the application of the routine; it suffices to use a postcondition of the form

ensure

q = **old** *q*

r = **old** *r*

... Other similar clauses ...

... Other postcondition clauses ...

Although it does the job, this technique has some disadvantages:

- Often, a routine will change only a few queries. In the above style, its postcondition will have many “similar clauses”, one for each query that it doesn’t affect.
- You have to go through the class, for each routine, to make sure you don’t forget any such queries.
- When you add a new query unrelated to existing routines, you have to add a “similar clause” to every one of these routines!

- Things become even more messy with inheritance: if you add a new query s in a descendant and do not redefine a feature f , most likely f will not modify s , but its original postcondition obviously cannot state that; to express this property, you would have to redefine f for the sole purpose of adding the clause $s = \mathbf{old} s$ to its postcondition!

→ In the form of an ensure then clause; see “REDECLARATION AND ASSERTIONS”, 10.17, page 283.

These observations show that along with the mechanism for expressing how a routine affects the value of certain queries, we need a complementary notation to express what queries it does **not** affect. The **only** postcondition clause achieves this goal. In a postcondition you may include one (and only one) clause of the form

only q, r, s

to state that *all queries other than q, r, s* are left untouched by the enclosing routine. In other words, this is an abbreviation for postcondition clauses

$q_1 = \mathbf{old} q_1$
 ...
 $q_n = \mathbf{old} q_n$

where q_1, \dots, q_n are all the queries other than q, r, s . The “unfolded form” defining the validity and semantics of an **Only** clause will express this.

The syntax is straightforward:



“Only” postcondition clauses
 $\mathbf{Only} \triangleq \mathbf{only} [\mathbf{Feature_list}]$

The syntax of assertions indicates that an **Only** clause may only appear in a **Postcondition** of a feature, as its last clause.

← Construct Postcondition, page 232.

Those other postcondition clauses let you specify how a feature *may* change specific properties of the target object, as expressed by queries. You may also want — this is called the **frame problem** — to restrict the scope of features by specifying which properties it *may not* change. You can always do this through postcondition clauses $q = \mathbf{old} q$, one for each applicable query q . This is inconvenient, not only because there may be many such q to list but also, worse, because it forces you to list them all even though evolution of the software may bring in some new queries, which will not be listed. Inheritance makes matters even more delicate since such “frame” requirements of parents should be passed on to heirs.

An **Only** clause addresses the issue by enabling you to list which queries a feature may affect, with the implication that:

- Any query *not* listed is left unchanged by the feature.
- The constraints apply not only to the given version of the feature but also, as enforced by the following rules, to any redeclarations in descendants (specifically, to their effect on the queries of the original class).

The syntax allows omitting the **Feature_list**; this is how you can specify that the routine must leave *all* queries unchanged (it is then known as a “*pure*” routine).

To express the validity and semantics of an **Only** clause, we must clarify how inheritance affects it. What does our example, **only *q, r, s*** appearing in the postcondition of a feature *f* of a class *C*, mean in a proper descendant *D*, which may add its own queries? Two cases are possible:

- *D* does not redeclare *f*. Then we must understand **only *q, r, s*** as we did in *C*: *f* may modify no query of *D*, whether from *C* or new in *D*.
- We may want to allow *f* to modify some features of *D* — say *t* and *u* — in addition to *q, r* and *s*. In that case we’ll redeclare *f* with a new postcondition clause, introduced by **ensure then**, of the form **only *t, u***. Such an **Only** clause appearing in a declaration means: “the feature may only affect, *among queries introduced since any preceding Only clause*, the queries *t* and *u*”. The effect then is cumulative, as if there had been a single **Only** clause of the form **only *q, r, s, t, u***.

Here are the rules stating these properties. First, the validity:



Only Clause rule

VAON

An **Only** clause appearing in a **Postcondition** of a feature of a class *C* is valid if and only if every **Feature_name** *qn* appearing in its **Feature_list** if any satisfies the following conditions:

- 1 • There is no other occurrence of *qn* in that **Feature_list**.
- 2 • *qn* is the final name of a query *q* of *C*, with no arguments.
- 3 • If *C* redeclares *f* from a parent *B*, *q* is not a feature of *B*.

Another condition, following from the **syntax**, is that an **Only** clause appears at the last element of a **Postcondition**; in particular, you may not include more than one **Only** clause in a postcondition. ← *Construct Postcondition, page 232.*

First, two useful notions of “unfolding”. First we need to include inherited features:

DEFINITION

Unfolded feature list of an Only clause

The **unfolded feature list** of an **Only** clause appearing in a **Postcondition** of a feature f in a class C is the **Feature_list** containing:

- 1 • All the feature names appearing in its **Feature_list** if any.
- 2 • If f is the redeclaration of one or more features, the final names in C of all the features whose names appear (recursively) in their unfolded Only clauses.

For an immediate feature (a feature introduced in C , not a redeclaration), the purpose of an **Only** clause of the form

only q, r, s

is to state that f may only change the values of queries q, r, s .

In the case of a redeclaration, previous versions may have had their own **Only** clauses. Then:

- If there was already an **Only** clause in an ancestor A , the features listed, here q, r and s , must be new features, not present in A . Otherwise specifying **only** q, r, s would either contradict the **Only** clause of A if it did not include these features (thus ruling out any modification to them in any descendant), or be redundant with it if it listed any one of them.
- The meaning of the **Only** clause is that f may only change q, r and s *in addition* to inherited queries that earlier **Only** clauses allowed it to change.

Note that this definition is mutually recursive with the next one.

The notion of unfolded feature list enables us to interpret an **Only** clause as a sequence of postcondition clauses asserting that the feature — double negation! — does *not* change any of the the *non*-listed features from the current class:

Unfolded Only clause

The **unfolded Only clause** of a feature f of a class C is a sequence of **Assertion_clause** components of the following form, one for every argument-less query q of C that does not appear in the unfolded feature list of the **Only** clause of its **Postcondition** if any:

$q = (\text{old } q)$

This will make it possible to express the semantics of an **Only** clause through a sequence of assertion clauses stating that the feature may change the value of no queries except those explicitly listed.

Note the use of the equal sign: for a query q returning a reference, the **Only** clause states (by *not* including q) that after the feature's execution the reference will be attached to the same object as before. That object might, internally, have changed. You can still rule out such changes by listing in the **Only** clause other queries reflecting properties of the object's *contents*.

9.8 CLASS INVARIANTS

The next category of assertion use is the class invariant, determined by the last clause of a class text, **Invariant**. Unlike a **Precondition** or **Postcondition** which characterizes a single feature, the class invariant applies to an entire class — more precisely, to all its exported features.



The invariant specifies properties which any instance of the class must satisfy at every instant at which the instance is observable by clients.

The class *CHAIN* quoted above has the following **Invariant** clause (with assertion tags removed from brevity):



```
deferred class CHAIN feature
...
invariant
  -- Definitions:
    empty = (count = 0)
    off = ((position = 0) or (position = count + 1))
    isfirst = (position = 1)
    islast = (not empty and (position = count))
  -- Axioms:
    count >= 0
    position >= 0; position <= count + 1
    empty => (position = 0)
    (not off) implies (item = i_th (position))
  -- Theorems:
    (is_first or is_last) implies (not empty)
end
```

As an example, the first `Assertion_clause` states that in all observable states the result of `empty` (a boolean function) called on a chain is true if and only if the value of `count` called on the same chain is zero. The second and third of those marked as `Axioms` state that the value of `position`, an integer attribute, always remains between 0 and the value of `count` plus one.

This `Invariant` has been divided into “Definitions”, expressing that certain queries may be defined in terms of others, “Axioms”, expressing constraints on the features, and “Theorems”, expressing properties which may be deduced from other clauses. This classification helps for readability but has no semantic consequence.

As already noted, the semantics of a class’s invariant — to define correctness and specify assertion monitoring — doesn’t just rely on the invariant clauses listed in the class itself but includes ancestors’ invariants, through the notion of “unfolded form”. The unfolded form is the concatenation of the clauses of its parents’s invariants, themselves unfolded in the same way, and its own clauses.

This ensures the consistency of the `type view` of inheritance. In particular, inheritance permits substitution of instances: if C is a descendant of B , instances of C will also be instances of B . But the soundness of this notion requires any semantic obligation on instances of B , as expressed by B ’s invariant, to be also applicable to instances of C . Hence the definition of the invariant of a class as including all invariant clauses from ancestors.

← On the two views of inheritance see “[OVERVIEW](#)”, 6.1, page 169.

9.9 THE CONSISTENCY OF A CLASS

Invariants and the notion of routine contract make it possible to define the consistency of a class, part of its *correctness*, one of its two characteristic semantic properties.



A class will be said to be consistent if its implementation satisfies the correctness requirements expressed by the preconditions on the routines of the class, the postconditions, and the class invariant.

The following notations serve to define this notion precisely. They are not part of Eiffel, but mathematical conventions used to talk **about** Eiffel classes and their semantic properties.

- If r is a routine, do_r denotes its body, pre_r its precondition, and $post_r$ its postcondition.
- If C is a class, INV_C denotes its class invariant.

WARNING: These are mathematical notations, not Eiffel text.

- If P and Q are assertions and A is an instruction or compound, the notation $\{P\} A \{Q\}$ expresses the property that whenever A is executed in a state in which P is true, the execution will terminate in a state in which Q is true. This is a standard concept from the theory of programming languages, taken here in a “total correctness” meaning:

Hoare triple notation (total correctness)

In definitions of correctness notions for Eiffel constructs, the notation $\{P\} A \{Q\}$ (a mathematical convention, not a part of Eiffel) expresses that any execution of the **Instruction** or **Compound** A started in a state of the computation satisfying the assertion P will terminate in a state satisfying the assertion Q .

If P , A and Q are extracted from the text of a routine with arguments, the Hoare triple $\{P\} A \{Q\}$ will be considered to hold if and only if it holds for all possible values of the formal arguments.

Earlier conventions enable us to assume that every **Attribute_or_routine** has both a **Precondition** and a **Postcondition**, and that every **Class** has an **Invariant**, by considering any missing clause as an implicit form for **require True**, **ensure True** or **invariant True**. In addition, we will take inheritance into account by considering the unfolded forms of these assertions, integrating inherited properties.

← “Precondition, postcondition, invariant”, – page 234.

→ “Unfolded form of an assertion”, – page 287.

These conventions make it possible to define class consistency:

DEFINITION

Class consistency

A class C is **consistent** if and only if it satisfies the following conditions:

- 1 • For every creation procedure p of C :

$$\{pre_p\} do_p \{INV_C \text{ and then } post_p\}$$

- 2 • For every feature f of C exported generally or selectively:

$$\{INV_C \text{ and then } pre_f\} do_f \{INV_C \text{ and then } post_f\}$$

where INV_C is the invariant of C and, for any feature f , pre_f is the unfolded form of the precondition of f , $post_f$ the unfolded form of its postcondition, and do_f its body.

The mathematical symbol \wedge represents boolean conjunction.

Class consistency is one of the most important aspects of the *correctness* of a class: adequation of routine implementations to the specification. The other aspects of correctness, studied below, involve **Check** instructions, **Loop** instructions and **Rescue** clauses.

→ “Correctness (class)”, – page 253.

9.10 CHECK INSTRUCTIONS

Another use of assertions is the **Check** instruction, of the form

```

check
    "Some_assertion" -- May include zero or more clauses
note -- This part is optional
    Tag: "Explanation"
end

```

This lets you express that a certain property, captured by **Some_assertion**, will be satisfied whenever execution reaches the instruction. As the name indicates, the instruction is there to require that some mechanism “check” that the property indeed holds. The mechanism in question may be a human reader, a mechanical program prover (if possible), or the execution itself. (We have not seen yet what effect, if any, assertion constructs may have on execution, and must wait a few more sections for an answer.)

→ The reader who does not want to wait may preview [“RULES OF RUN-TIME ASSERTION MONITORING”, 9.13, page 253.](#)

A common use of a **Check** instruction is just before a routine call, to express one or both of two properties without which the call’s execution couldn’t make sense:

- For a **Qualified_call**, the target is attached (not void).
- The routine’s precondition is satisfied.

As an example of the first case, the body of procedure *remove_left* in the class *LINKED_LIST* of EiffelBase contains the following extract:



```

previous := item (position - n - 1)
check
    previous /= Void
note
    why: "Value at position - n - 1 cannot be void"
end
previous.put_right (active)

```

In that class, *item* (*i*) yields the element at position *i*. The **Check** instruction expresses that the value at *position - n - 1* is not void, and so can be used as the target of the call to *put_right* that follows.

The **note** part is optional but recommended: as here, it lets you explain, through a **why** entry, the reasoning that leads you to expect that the assertion will hold.

← [“ANNOTATING A CLASS”, 4.8, page 122.](#)

Another style recommendation illustrated here is to indent the **Check**, to separate it from instructions that perform actual steps of the algorithm.

As an example of the second case, the body of procedure *put* in the same class contains:



```

if i = 1 then
    lt.put_right (first_element)
else
    check
        i - 1 >= 1
        i - 1 <= count
    note
        why: "See condition of if and invariant"
    end
    left_neighbor := item (i - 1)
    ...
end

```

This should be understood in light of the precondition for *item* (*j*):

```

require
    index_large_enough: j >= 1
    index_small_enough: j <= count

```

The **Check** instruction expresses that the call to *item* satisfies the precondition thanks to the context (the **Conditional** instruction) and the invariant.



More generally, a **Check** instruction is useful in the following situation. You know that the proper execution of a certain computation requires some consistency condition (such as a call's target being attached, or a precondition satisfied). To do your job properly, you so design the context of the computation as to guarantee the desired condition. If your reasoning and its consequence (that the consistency condition will be satisfied) are not obvious to a reader of the software text, a **Check** instruction will clarify that you did not overlook your obligations.

In the presence of assertion monitoring as discussed below, the **Check** will also help detect the mistake if you did make one after all.

The general form of a **Check** instruction is:



Check instructions

Check \triangleq **check** Assertion [Notes] **end**

The **Notes** part is intended for expressing a formal or informal justification of the assumption behind the property being asserted.

The following definition expresses the semantics:



Check-correct

An effective routine r is **check-correct** if, for every **Check** instruction c in r , any execution of c (as part of an execution of r) satisfies its **Assertion**.

9.11 LOOP INVARIANTS AND VARIANTS



Our next application of assertions uses them to guarantee the correctness of loops. It uses an assertion, the **loop invariant**, as well as an integer expression, the **loop variant**. The variant is not syntactically an assertion, but plays a closely related role.

The invariant determines the properties ensured by the loop on exit; the variant guarantees that the loop's execution terminates.

Here is an example of these constructs in a loop from routine *search_child* (which looks for a certain node among the children of the current node) in the EiffelBase class *LINKED_TREE*:



```

from
    go_before
invariant
    0 <= child_position; child_position <= arity + 1
until
    child_after or else (j = i)
loop
    child_forth
    if (sought = child) then j := j + 1 end
variant
    arity - child_position + 1
end

```

This discussion refers to instructions of a **Loop**: the **from** clause or **Initialization**, executed once at the start of the loop; the iteration or **loop** clause, executed zero or more times until the **Exit_condition**, introduced by **until**, holds.

→ “[LOOP](#)”, 17.7, page 494.

The invariant expresses a property of *child_position*, the index of the child being looked at: *child_position* will remain between 0 and *arity* (the number of children) plus one. Stating that this property is a loop invariant means asserting that the **Initialization** ensures it, and that every iteration preserves it. This is indeed the case:

- The **Initialization**, *go_before*, moves the child cursor to the node's first child, or to position 0 if there is no child.

- In the iteration, *child_forth*, which moves the child cursor right by one position, is only executed when the property *child_after* is not satisfied (since this condition is part of the *Exit_condition*).

The variant is an integer expression, $arity - child_position + 1$, which is always non-negative and decreases on every iteration. This guarantees that the loop will terminate.

Here now are the precise rules governing these constructs, starting with the syntax. **Invariant** and **Variant** appear in the syntax of **Loop**, appearing in the chapter on control structures. A loop invariant is a specimen of **Invariant**, also applicable to class invariants and given earlier in this chapter. A loop **Variant** (which contains an integer expression, not an assertion) has the following syntax: → Page 495.
← Page 232.



Variants

Variant \triangleq **variant** [Tag_mark] Expression

The optional **Tag_mark** labels the variant in the same way as for an **Assertion_clause**. The variant must satisfy a simple constraint:



Variant Expression rule VAVE

A **Variant** is valid if and only if its variant expression is of type **INTEGER** or one of its sized variants.

The sized variants are **INTEGER_X** and **NATURAL_X** where *X* is an explicit length, for example **INTEGER_16**.

→ “**INTEGERS**”, 30.6, page 820.

Associated terminology:

Loop invariant and variant

The **Assertion** introduced by the **Invariant** clause of a loop is called its **loop invariant**. The **Expression** introduced by the **Variant** clause is called its **loop variant**.

Semantically, the invariant **INV** of a correct loop satisfies two properties:

- The loop’s Initialization (**from** clause) ensures the truth of **INV**.
- Any iteration started in a state that does not satisfy the **Exit_condition** but satisfies **INV** terminates with **INV** true again.

As a result of these properties, the invariant will still be satisfied on loop exit, at which point the **Exit_condition** will hold. Their conjunction is the output condition of the loop. From a theoretical viewpoint, the goal of the loop is to achieve this condition, and the looping process reaches it by successive approximations.

You may view the variant as an estimate of the remaining distance to the goal. The variant *VAR* of a correct loop satisfies two properties:

- The **Initialization** sets *Var* to a non-negative value.
- Any iteration started in a state that does not satisfy the **Exit_condition** decreases the value of *VAR* while keeping it non-negative.

As a result of these properties, since the variant is an integer expression, the iterations may not go on forever.

The following definition captures this correctness semantics:

← For the Hoare triple notation $\{P\} A \{Q\}$ see page 247 above.



Loop-correct

A routine is **loop-correct** if every loop it contains, with loop invariant *INV*, loop variant *VAR*, **Initialization** *INIT*, **Exit condition** *EXIT* and body (Compound part of the **Loop_body**) *BODY*, satisfies the following conditions:

- 1 • $\{\text{true}\} \text{INIT} \{ \text{INV} \}$
- 2 • $\{\text{true}\} \text{INIT} \{ \text{VAR} \geq 0 \}$
- 3 • $\{ \text{INV and then not EXIT} \} \text{BODY} \{ \text{INV} \}$
- 4 • $\{ \text{INV and then not EXIT and then } (\text{VAR} = v) \} \text{BODY} \{ 0 \leq \text{VAR} < v \}$

Conditions **1** and **2** express that the initialization yields a state in which the invariant is satisfied and the variant is non-negative. Conditions **3** and **4** express that the body, when executed in a state where the invariant is satisfied but not the exit condition, will preserve the invariant and decrease the variant, while keeping it non-negative. (*v* is an auxiliary variable used to refer to the value of *VAR* before *BODY*'s execution.)

In keeping with the general use of the word “contract” — for features, for classes — the invariant and variant of a loop (only the former an assertion) are said to define the contract of the loop.

9.12 THE CORRECTNESS OF A CLASS

In connection with the various uses of assertions, we have seen how a class may be “correct” in several partial ways:

- **Consistent**: every feature satisfies its precondition and postcondition, every creation procedure ensures the invariant, every exported feature preserves the invariant.
- **Check-correct**: the assertion of every **Check** instruction holds.
- **Loop-correct**: loops preserve their invariants and decrease their variants.

Exceptions will lead to one more variant: a class is *exception-correct* if in the case of a non-retried exception, leading to a failure of the enclosing routine, the “rescue clause” will restore the class invariant. → “EXCEPTION CORRECTNESS”, 26.9, page 702.

The combination of these properties yields the full notion of correctness:



Correctness (class)

A class is **correct** if and only if it is consistent and every routine of the class is check-correct, loop-correct and exception-correct.



Do not confuse correctness and validity. Correctness is a semantic notion, expressing that the implementation of a class matches its specification. Validity is simply the property that a construct is well-formed, such as a routine call with the proper number and type of actual arguments. The correctness of a software element is a meaningless notion unless the element is valid.

← “CORRECTNESS”, 2.10, page 99.



Ideally, an Eiffel environment should come with tools which can prove or disprove the correctness of a class as defined here. This is still beyond the reach of most of today’s environments, although the technology is progressing quickly. To understand classes and contracts, however, it is essential to refer to this notion of class correctness, even if you have to assess correctness by manual examination of the class rather than through proof tools.

As the next best thing to proofs, Eiffel environments must support run-time monitoring of contracts, as described next.

9.13 RULES OF RUN-TIME ASSERTION MONITORING

Complementing the notion of class correctness, run-time monitoring provides the second facet of assertion semantics.



Contracts, as expressed through assertions and loop variants, express correctness requirements on software texts. If we view them not just as specification (as in the definition of class correctness) but as language constructs, they raise the same semantic question as any other construct: what’s the run-time effect? In the execution of a *correct* system — the only case that, in principle, matters — every assertion will always be satisfied at the times when it has to: a precondition on every call to its routine, a postcondition on return and so on. So in theory it should not be necessary to define the semantics of assertions evaluation: for a correct class, the effect of evaluating an assertion is irrelevant, since the value is always true! This may be called the paradox of assertion semantics.

The paradox is only theoretical. In practice, short of proofs as discussed above, one must often accept the prospect that a system may contain errors — may not be correct. Then assertions provide crucial help in detecting these errors and suggesting corrections. By directing the run-time system to evaluate assertions and variant clauses, and to trigger an exception if it detects a violation, you check the consistency between what the software does (the feature implementations) and what you think it does (the contracts): you let the tools call your bluff. This gives a remarkable tool for debugging, testing and maintaining software systems.

Eiffel implementations are indeed required to provide, usually through a compilation option, mechanisms to evaluate contract elements — assertions and loop variants — during the execution of a system. We now review the details of this requirement.

Associated boolean expression

We must first clarify what it means to evaluate an assertion. As a language construct, the notion of assertion derives from boolean expressions. Like a boolean expression, an assertion describes a property that, for the computation captured in a certain state, is true or false.

Two possibilities available for writing assertions — specifically, postconditions — do not exist in boolean expressions: **Old** expressions and **Only** clauses. Their semantics, however, does reduce in the end to that of boolean expressions thanks to the associated “unfolded forms”:

- **old** *exp* stands for an extra variable (the “associated variable” of the **Old** expression) that would have been initialized to the value of *exp* on entry to the routine. ← ““Old” expression”, page 239.
- **only** q_1, \dots, q_n stands for a set of assertions of the form $t = \mathbf{old} t$, one for every query t that is not one of the q_n . ← ““Only” clause”, page 241.

We can capture these equivalences through a notion of unfolded form:

Local unfolded form of an assertion

The **local unfolded form** of an assertion a — a **Boolean expression** — is the Equivalent Dot Form of the expression that would be obtained by applying the following transformations to a in order:

- 1 • Replace any **Only** clause by the corresponding unfolded Only clause.
- 2 • Replace any **Old** expression by its associated variable.
- 3 • Replace any clause of the **Comment** form by **True**.

The unfolded form enables you to understand an assertion, possibly with many clauses, as a single boolean expression. The use of **and then** to separate the clauses indicates that you may, in a later clause, use an expression that is defined only if an earlier clause holds (has value true).

This unfolded form is “local” because it does not take into account any inherited assertion clauses. This is the business of the full (non-local) notion of unfolded form of an assertion, introduced in the discussion of redeclaration.

→ “Unfolded form of an assertion”, page 287.

The Equivalent Dot Form of an expression removes all operators and replaces them by explicit call, turning for example $a + b$ into $a.\textit{plus}(b)$. This puts the result in a simpler form used by later rules.

If an **Only** clause is present, we replace it by its own unfolded form, a sequence of **Assertion_clause** components of the form $q = \textit{old } q$, so that we can treat it like other clauses for the assertion’s local unfolded form. Note that this unfolding only takes into account queries explicitly listed in the **Only** clause, but not in any **Only** clause from an ancestor version; inheritance aspects are handled by the normal unfolding of postconditions, applicable after this one according (as noted above) to the general notion of unfolded form of an assertion

The syntax permits a **Comment** as **Unlabeled_assertion_clause**. Such clauses are useful for clarity and documentation but, as reflected by condition 3, cannot have any effect on run-time monitoring.

← Page 232.

Assertion monitoring

SEMANTICS

An Eiffel environment must make it possible to evaluate assertions — that is to say, the boolean expressions resulting from their unfolding as just defined — during execution.

For a correct system, as noted, such evaluation will have no effect on execution semantics, except through possible side effects of the functions called by assertions.

For an incorrect system, if an assertion evaluates to true, it has no further effect on the outcome of the computation. If it evaluates to false, it will trigger an exception, disrupting the normal flow of computation.

→ Chapter 26 discusses exceptions.

The first notion is *evaluation* of an assertion or loop variant. For a variant — an integer expression — the meaning is clear; for assertions we must be more precise:

DEFINITION

Evaluation of an assertion

To **evaluate** an assertion consists of computing the value of its unfolded form.

This defines the value of an assertion in terms of the value of a boolean expression, as given by the discussion of expressions.

→ “*Expression Semantics (strict case)*”, page 773 and “*Operator Expression Semantics (semistrict cases)*”, page 777

Should assertions be evaluated? Not necessarily. You may define various levels of *monitoring*, including no evaluation at all:

DEFINITION

Assertion monitoring

The execution of an Eiffel system may evaluate, or **monitor**, specific kinds of assertion, and loop variants, at specific stages:

- 1 • Precondition of a routine *r*: on starting a call to *r*, after argument evaluation and prior to executing any of the instructions in *r*'s body.
- 2 • Postcondition of a routine *r*: on successful (not interrupted by an exception) completion of a call to *r*, after executing any applicable instructions of *r*.
- 3 • Invariant of a class *C*: on both start and termination of a *qualified* call to a routine of *C*.
- 4 • Invariant of a loop: after execution of the **Initialization**, and after every execution (if any) of the **Loop_body**.
- 5 • Assertion in a **Check** instruction: on any execution of that instruction.
- 6 • Variant of a loop: as with the loop invariant.

Unlike other semantic definitions of this book the rule is in “may” rather than “must” mode, describing run-time checks that you will wish to enable or disable depending on the circumstances. The possibilities are detailed next.

The following definition describes how assertion monitoring can catch cases of software incorrectness:

DEFINITION

Assertion violation

An **assertion violation** is the occurrence at run time, as a result of assertion monitoring, of any of the following:

- An assertion (in the strict sense of the term) evaluating to false.
- A loop variant found to be negative.
- A loop variant found, after the execution of a **Loop_body**, to be no less than in its previous evaluation.

To simplify the discussion these cases are all called “*assertion violations*” even though a variant is not technically an assertion.

Assertions affect the semantics of system execution only in the case of assertion violations:



Assertion semantics

In the absence of assertion violations, assertions have no effect on system execution other than through their evaluation as a result of assertion monitoring.

An assertion violation causes an exception of type *ASSERTION_VIOLATION* or one of its descendants.



The result of the exception is detailed in the corresponding chapter. Any exception, when it occurs, interrupts the execution of some routine that has been started but not yet finished: the *recipient* of the exception. In accordance with the principles of Design by Contract: → “*SEMANTICS OF EXCEPTION HANDLING*”, 26.10, page 702

- For a precondition violation, the recipient is the caller: it has not observed its obligations, and any attempt to execute the routine would be meaningless, and perhaps harmful. In this case the error — the *bug* — is on the client side.
- For a postcondition violation, the recipient is the routine itself: it is unable to fulfill its obligations. The bug is on the supplier side.

The definition of “exception cases”, part of exception semantics, provides the full specification for these two kinds of violation and all others. → Page 706.

Levels of assertion monitoring

As noted for the definition of assertion monitoring, the execution “may” monitor exceptions but does not always have to do so. For a program that is known to be correct — through other means, for example a *proof* that it meets its contracts — you may prefer to avoid the overhead of monitoring. An Eiffel development environment, however, must offer you at least basic monitoring facilities.

A number of levels of monitoring are predefined:

Implementations may — and usually do — provide more combinations than the four required by the last part of the rule.



It is common to provide *precondition* checking only (variant 2) as the default. Checking preconditions on routine entry avoids disasters, since a *Routine_body* might attempt to perform erroneous or impossible actions when executed in a state which does not satisfy the routine’s precondition, but normally causes only a modest performance penalty. More extensive monitoring is especially useful for quality assurance, maintenance, debugging, testing and regression analysis.

Assertion monitoring levels

An Eiffel implementation must provide facilities to enable or disable assertion monitoring according to some combinations of the following criteria:

- Statically (at compile time) or dynamically (at run time).
- Through control information specified within the Eiffel text or through outside elements such as a user interface or configuration files.
- For specific kinds as listed in the definition of assertion monitoring: routine preconditions, routine postconditions, class invariants, loop invariants, **Check** instructions, loop variants.
- For specific classes, specific clusters, or the entire system.

The following combinations must be supported:

- 1 • Statically disable all monitoring for the entire system.
- 2 • Statically enable precondition monitoring for an entire system.
- 3 • Statically enable precondition monitoring for specified classes.
- 4 • Statically enable all assertion monitoring for an entire system.

Invariant and qualified calls



As noted in the definition of assertion monitoring, a class invariant will only be checked for **qualified** calls. If r is a routine of a class C , a call in dot-notation is qualified if it is of the form ← *Case 3, page 256.*

$a.r(\dots)$

with an explicit target, here a .

An operator expression using a binary operator, such as $a + b$, is also a qualified call since it is an abbreviation for a call $a.plus(b)$, assuming a feature *plus alias* "+". Since the difference is of syntax only, we can limit ourselves here to the case of calls with an explicit dot.

→ *"THE EQUIVALENT DOT FORM", 28.8, page 780. Chapter 23 discusses qualified and unqualified calls in details.*

Whether it appears in the text of a routine of C or in another class, such a qualified call will cause the invariant to be evaluated (both before and after the call).

A routine of C may also contain an **unqualified** call, written just

$r(\dots)$

where the target is not *a* any more but the current object. Such an unqualified call will not cause an invariant check. → “*Current object, current routine*”, page 649.

The form *Current.r(...)*, which has the same semantics in the absence of assertions, is qualified and so will trigger an invariant check.

Feature adaptation

10.1 OVERVIEW

Chapter [6](#) introduced inheritance as a module enrichment technique. You inherit from a class out of sheer mercenary interest: you want its features. But that doesn't necessarily mean accepting all these features at face value.

A key attraction of the inheritance mechanism is that it lets you tune inherited features to the context of the new class. This is known as feature adaptation. The present discussion covers the principal mechanisms, leaving to a [later one](#) some important complements related to repeated inheritance.

→ Chapter [16](#) presents repeated inheritance.

This chapter is the longest of this book, which should not be a surprise since it explores in full detail some of the most fascinating aspects of object technology: how to play mix and match with software components, taking advantage of the best features of existing classes while refining, adapting or overriding what is not exactly suited to your new need. Only a few basic concepts are involved, but they interact in diverse and powerful ways.

So make sure you have a comfortable armchair and a big cup of coffee, and for the 50 pages of this chapter be prepared to question, implement, override, rename, merge or otherwise wring all those features that your ancestors, for better or worse, bequeathed to you.

There are actually 40 more pages of wringing in chapter [16](#).

10.2 TERMINOLOGY: REDECLARATION, REDEFINITION, EFFECTING

Our major focus will be the two **redeclaration** mechanisms that help adapt inherited features to the local context of a class:

- **Redefinition**, which may change an inherited feature's original implementation, signature or specification.
- **Effecting**, which provides an implementation (or **effective** version) for a feature that did not have one in the parent. The parent's version, deprived of any implementation, but with a signature and specification, is said to be **deferred**; deferred features play an important role in analysis and design, which this chapter will explain.

The term "effecting" sometimes surprises at first, but achieves consistent terminology: to effect a feature is to make it effective.

The purpose of redefining a feature is often to extend (rather than discard) its original implementation. We will see how the **precursor** mechanism enables you, in a redefinition, to reuse and extend the original version.

Two closely related facilities, which the discussion will address in detail, are the possibility of **undefining** an inherited feature, to forget its original implementation, and of merging abstractions by **joining** two or more features inherited from different parents.

Another adjacent concept is **repeated inheritance**, which enables a class to inherit twice or more from a given ancestor, letting the designer control what happens to the common feature heritage. This topic is important enough to deserve a chapter of its own, coming only later in this book, after the conformance chapter, since repeated inheritance rules rely extensively on those of type conformance. → *Chapter 16.*

Although with the present chapter the major language constructs involving inheritance will have been introduced, we are still missing an important part of the picture. To grasp the full extent and practicality of the techniques introduced below, you will need to understand *polymorphism* and *dynamic binding*, studied in subsequent chapters. Together, these notions are responsible for some of the most powerful characteristics of the object-oriented method. → “*POLYMORPHISM*”, 22.11, page 606; “*DYNAMIC BINDING*”, 23.12, page 638.

10.3 REDECLARING INHERITED FEATURES: WHY AND HOW



A class inheriting from another may add new features of its own. But what about the old ones? So far the presentation has assumed that an heir will be happy enough to obtain every inherited feature “as is” from a parent. To be sure, the heir may *rename* the feature, but this does not change it; the effect is simply to make it available to the client’s dependents under a name that is better suited to the local context. ← “*RENAMING*”, 6.9, page 183.

Inheritance offers more. When you inherit a set of features, you may want to adapt those whose original *specification* or *implementation* did not take advantage of the heir’s specific properties.

Redefinition is the basic method for achieving such an adaptation. By redefining an inherited feature you may give it a new implementation, a new signature, or a new set of assertions, as long as you follow the applicable rules to ensure that the new version remains compatible with the old one as seen by clients. You may even redefine a function into an attribute, switching from an algorithmic representation to one that simply stores feature values. Every proper descendant of a class may provide its own alternative redefinition.

In some cases, the original form of a routine does not provide any default implementation at all; this is an explicit invitation for proper descendants to offer various implementations. Such unimplemented features, and the classes that introduce them, are said to be **deferred**; proper descendants may then **effect** those features (make them **effective**).

In the software construction process, classes and features may in fact remain deferred for a long time, providing a high-level notation for system analysis and design.

The basic terminology has already been previewed:

DEFINITION

Redeclare, redeclaration

A class **redeclares** an inherited feature if it redefines or effects it. A declaration for a feature f is a **redeclaration** of f if it is either a redefinition or an effecting of f .



This definition relies on two others, appearing below, for the two cases: *redefinition* and *effecting*.

Be sure to distinguish *redeclaration* from *redefinition*, the first of these cases. Redeclaration is the more general notion, redefinition one of its two cases; the other is *effecting*, which provides an implementation for a feature that was deferred in the parent. In both cases, a redeclaration does not introduce a new feature, but simply overrides the parent's version of an inherited feature.

redeclaration:

In the case of a redefiniti

Getting the full power of deferred features requires two more mechanisms:

- Sometimes a class will be able to merge two or more features that it inherits from separate parents; in so doing the class combines several abstractions into one. This is the join mechanism.
- In some cases, as you inherit an effective feature from a parent, you may want to discard the inherited implementation altogether, recanting all the sins of its earlier effective life. This is the process of undefinition, which turns an effective feature into a born-again deferred feature.

The following sections explore redefinition, deferred features, undefinition and join. The discussion will first explain these facilities and their role in software analysis, design and implementation. The second part of the chapter, which you may skip on first reading, gives the more formal set of corresponding syntactic rules and validity constraints, together with the resulting semantic definitions.

The formal part starts with 10.25, page 306.

10.4 FEATURE ADAPTATION CLAUSES

For a start, let us just refresh our memory as to the syntactical context of this discussion: the **Inheritance** clause of a class declaration, which may contain one or more **Parent** parts. Here is a simplified form of the beginning of class **TWO_WAY_TREE** in EiffelBase:

← Another descendant of **TREE**, class **FIXED_TREE**, served to illustrate inheritance basics in "**AN INHERITANCE PART**", 6.2, page 169



```
note
... (Notes clause omitted)...

class TWO_WAY_TREE [T] inherit
  TREE [T]
    redefine
      higher, ...
    end
  BI_LINKABLE [T]
    rename
      ... (Rename subclause omitted)...
    redefine
      put_between
    end
  TWO_WAY_LIST [like Current]
    rename
      ... (Rename subclause omitted)...
    redefine
      first_child, update_after_insertion,
      duplicate, merge_right, merge_left
    end
feature
  ... Rest of class omitted ...
```

Each **Parent** part is relative to one of the class's parents and may include a **Feature_adaptation** subclause (optional, but present for all three parents above). Here again is the corresponding syntax:



```
Inheritance ≙ inherit Parent_list
Parent_list ≙ "{Parent ";" ... }
Parent ≙ "Class_type
         [Feature_adaptation]
Feature_adaptation ≙ [Rename]
                    [New_exports]
                    [Undefine]
                    [Redefine]
                    [Select]
                    end
```

← The original presentation of this syntax is on page 171.

The **Rename** and **New_exports** clauses have been discussed in [previous chapters](#). The next sections explain **Redefine**, **Undefine** and **Select**.

← “[RENAMING](#)”, 6.9, page 183; “[Adapting the export status of inherited features](#)”, ., page 204.

10.5 WHY REDEFINE?

The first mechanism to study is feature redefinition, which allows you to change some aspects of an inherited feature.



Assume you write a class **C** that describes a specific variant of the concepts covered by an existing class **B**. **C** will be an heir of **B**. You may find that, for this variant, the inherited version of a certain feature **f** is not appropriate any more. This sets the stage for redefining **f** in **C**.

Besides its name, a feature is characterized by three properties:

← “[FEATURE DECLARATIONS: SYNTAX](#)”, 5.10, page 140.

- The feature has a **signature**, defined by the number and type of its arguments and result, if any.
- It either is deferred or has an **implementation**, including the choice between attribute or routine, external or not, and for a non-external routine the **Routine_body**, **Local_declarations** and **Rescue**.
- It has a **specification**, defining the feature's **contract**: **Precondition** and **Postcondition** (for routines only).

A routine may also have a **Header_comment** and an **Obsolete clause**, which a redefinition may change.

A feature redefinition may affect one or more of these three aspects. In general, a change of specification implies a change of implementation.

There are two possible reasons, *correctness* and *efficiency*, for redefining a feature:

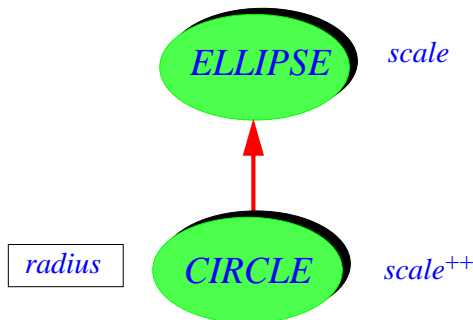
- The original version may perform actions or compute results that are incorrect for the new class, for example because they do not update some of the new attributes.
- If the original version is still appropriate, it may not be efficient enough, because it fails to take advantage of specific properties of the new class.

Signature redefinition falls in the correctness category: the types of arguments or results, as originally declared, are not appropriate for the new class. *Implementation* redefinition may be for correctness, efficiency or both. A change of *specification* involves correctness since it means the redefined version offers a new contract to its clients.

10.6 REDEFINITION EXAMPLES

To get a good feel for redefinition, let us look at a pair of simple examples illustrating each of the two purposes cited.

As a case of redefinition for correctness, assume a class *CIRCLE*, inheriting from *ELLIPSE* and adding an attribute *radius*:



*Inheritance
structure for
circles and
ellipses*



As always, it is useful to check that we are not misusing inheritance. Here there is hardly any doubt that the structure is right: every circle may be viewed as an ellipse that happens to have only one focus.

Let *scale* be the procedure that scales a figure by a certain ratio. Since attribute *radius* is not present in *ELLIPSE*, the version of *scale* inherited from *ELLIPSE* does not update the value of that attribute. Class *CIRCLE* must redefine *scale* to make sure that it updates not just the attributes inherited from *ELLIPSE*, but also the specific *CIRCLE* attributes such as *radius*. (The problem would not arise if *radius* was a function, defined in terms of attributes inherited from *ELLIPSE*, rather than an attribute.)

As illustrated by the figure, the graphical convention for a redefined feature uses two plus signs after the feature's name, as in *scale++*.

Now an example of redefinition for efficiency. Class *CIRCLE* may redefine as follows the function *contains* which determines whether a point is inside a closed figure:



```
contains (p: POINT): BOOLEAN
    -- Is p inside circle?
require
    point_exists: p /= Void
do
    Result :=(origin.distance (p) <= radius)
ensure
    ... Postcondition omitted ...
end
```

ELLIPSE has a version of *contains* too. Because an ellipse is a more general figure than a circle, the *ELLIPSE* version is more complex than the above; it would still be correct for circles, but less efficient since it does not take advantage of the special properties of circles. Redefinition solves the problem.

10.7 THE REDEFINITION CLAUSE

Whether for correctness or efficiency, the redefinition of a feature must be explicitly announced in a *Redefine* subclause of the *Feature_adaptation* for the corresponding parent, as in



```
class CIRCLE inherit
    ELLIPSE
    rename
        ...
    redefine
        scale, contains,...
    end
... Rest of class omitted...
```

The names given in the *Redefine* subclause must be the final names of features inherited from the given parent. (In other words, these are the names *after* any renaming; this is easy to remember since the *Rename* clause always appears before *redefine* and other feature adaptation clauses.) ← “Final name” was defined on page 186.

Such a **Redefine** subclause allows — and requires — class *CIRCLE* to include (in a **Feature_clause**) new feature declarations, such as given above, for *scale*, *contains* and others listed after the keyword **redefine**. These declarations will override the ones inherited from the parent, here *ELLIPSE*. Without the **Redefine** subclause, such declarations would make *CIRCLE* invalid, since it would now have two features called *scale*, *contains* etc., a case of invalid **name clash**. → “**NAMECLASHES**”, 10.23, page 297

To discuss redefinition it will be convenient to refer to the “precursor” of an inherited feature — its original form in the parent:



Precursor (initial definition)

If a class inherits a feature from a parent, either keeping the feature unchanged or redefining it, the parent’s version of the feature is called the **precursor** of the feature.



With the mechanisms seen so far, every feature of a parent yields a feature in the heir; so every inherited feature has **one** precursor. Mechanisms explored later — joining of deferred features, and sharing under repeated inheritance — may result in the merging of two or more parent features into just one heir feature. This will require extending the definition to account for features having more than one precursor. → “**Precursor (joined features)**”, page 315. See also the more formal definition, page 473.

10.8 REDEFINITION IN THE SOFTWARE PROCESS



(This section introduce no new language concept but broadens the discussion by presenting methodological aspects.)

Before proceeding with more technical aspects of redefinition, it is useful to reflect a little on the implications of this notion for object-oriented software engineering. Feature redefinition is part of the answer to a major software engineering issue: reconciling reusability with extendibility.

In software, it is seldom satisfactory to reuse an element exactly as it is; often, you must also adapt it to a specific context. With redefinition, as suggested by the simple examples above, you can keep those features that are still appropriate for the new context, while overriding the implementations of those which need to be adapted.

The ability to change the signature of an inherited routine, studied below, is also essential to the smooth functioning of Eiffel’s type system.

It is useful to compare this technique with another of the mechanisms for adapting an inherited feature: renaming. The distinction to keep in mind is between a *feature* and a *feature name*: ← “**RENAMING**”, 6.9, page 183.

- A feature of a class is a certain operation (routine or attribute) applicable to instances of the class. The feature is normally passed on to heirs, except for redeclaration, which allows an heir to substitute another feature.
- Every feature of a class has a **final name** relative to that class, called just its “feature name” if there is no ambiguity. This is the name used by the class, its clients and heirs to refer to the feature. The name is normally passed on to heirs, except for renaming, which allows an heir to substitute another name for the same feature.

Redefinition and renaming serve complementary purposes:

Redefinition and renaming

Redefinition changes the feature, but keeps its name.
Renaming keeps the feature, but changes its name.

You may want to apply both mechanisms to a given feature, to change both the feature and its name:



```
class B inherit
  A
    rename
      f as new_f
    redefine
      new_f
    end
  feature
    ... Rest of class omitted ...
```



Remember that once you have renamed a feature the only name that makes sense for it in the rest of the class, past the **Rename** clause, is the new name, which becomes its final name in *C*, here *new_name*. In particular, the **Redefine** subclause — as well as **Undefine** — only refers to the new name. So in this example it would have been invalid to write

```
redefine f
```

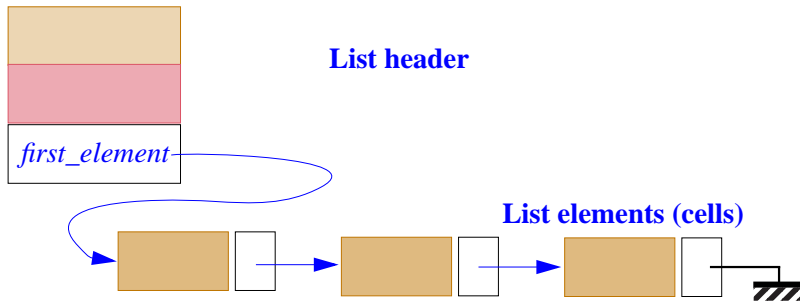
since *f* is not the name of a feature that *C* inherits from *B* (unless the **Rename** subclause separately renames another inherited feature to *f*).

10.9 CHANGING THE SIGNATURE

The preceding example redefinitions affected the implementation, for either correctness or efficiency reasons. Here now is an example where we need to change the signature of an inherited feature. That feature is an attribute, so its signature only includes the attribute’s type.



Consider class *LINKED_LIST [T]* in EiffelBase, representing one-way linked lists of objects of type *T* (the formal generic parameter).



**One-way
linked list**

One of the attributes of class *LINKED_LIST* is a reference to the first element of a list:

```
first_element: LINKABLE [T]
```

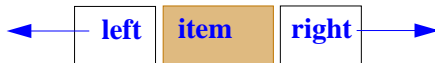
The actual class text uses LINKABLE [like first]. This doesn't affect the discussion.

The type of the corresponding objects, *LINKABLE [T]*, represents list cells, chained to their right neighbors:



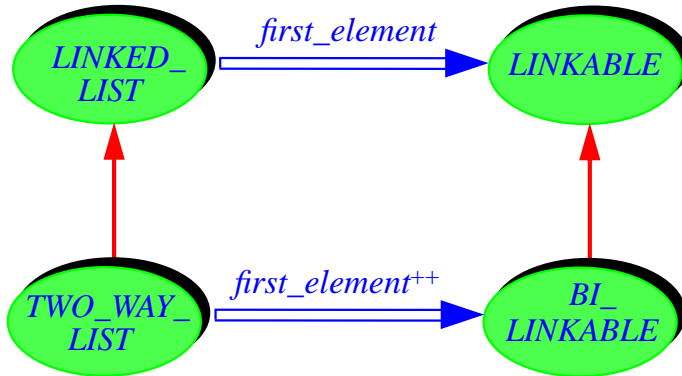
**Linkable list
cell**

Various proper descendants of *LINKED_LIST* support variants of the linked list data structure. An immediate heir is *TWO_WAY_LIST*, which, instead of linkables, uses “bi-linkables”, chained not just to their successors but also to their predecessors:



**Bi-linkable list
cell**

Class *BI_LINKABLE* is itself an heir from *LINKABLE*:



Signature redefinition

Clearly, the *first_element* of a *TWO_WAY_LIST* should not just be a linkable any more, but a bi-linkable. Hence the need to redefine that attribute, which will appear in *TWO_WAY_LIST* as

```
first_element: BI_LINKABLE [T]
```

The redefinition of *first_element* into a *BI_LINKABLE* in *TWO_WAY_LIST* follows the rule (given in detail below) requiring that any change of type in a redeclaration replace the original with a type that conforms to it (by being based on a descendant class, as *BI_LINKABLE* for *LINKABLE*).

→ “REDECLARATION AND TYPING”, 10.16, page 280

In this example, the redefined feature is just an attribute. There is often a concomitant need to change the types of routine arguments. For example, the insertion routine *put_element* may be declared in *LINKED_LIST* as

```
put_element (lt: LINKABLE[T] ; i: INTEGER) is...
```

put_element is secret since clients of the list classes never explicitly manipulate linkables, only objects of type T.

Clearly, class *TWO_WAY_LIST* needs to adapt *put_element* to give it a first argument *lt* of type *BI_LINKABLE [T]*. A redefinition of *put_element* will achieve this.

10.10 THE NEED FOR ANCHORED DECLARATIONS



Cases such as the redeclaration of the argument *lt* of *put_element* are so frequent in inheritance hierarchies that they warrant a special mechanism, bypassing the need for explicit redefinition. Rather than the above, the signature of *put_element* as declared in *LINKED_LIST* is

```
put_element (lt: like first_element; i: INTEGER) is...
```

meaning that *lt* has the same type as *first_element*: type *LINKABLE* [*T*] in *LINKED_LIST* and, in any proper descendant of this class, the new type, if any, to which *first_element* has been redefined. This mechanism, known as **anchored declaration**, is discussed in detail in a subsequent chapter. It is a form of implicit signature redefinition.

→ “*ANCHORED TYPES*”, 11.10, page 339

10.11 DEFERRED FEATURES

Feature redefinition, as just studied, lets you override the implementation, signature or specification of a feature that already had an implementation in a proper ancestor.



In some cases, the designer of that ancestor could not provide such a default implementation, or did not want to. It is possible to declare a feature without choosing an implementation by making it **deferred**. This transfers to proper descendants the responsibility for providing an implementation through a new declaration, called an **effecting** of the feature.

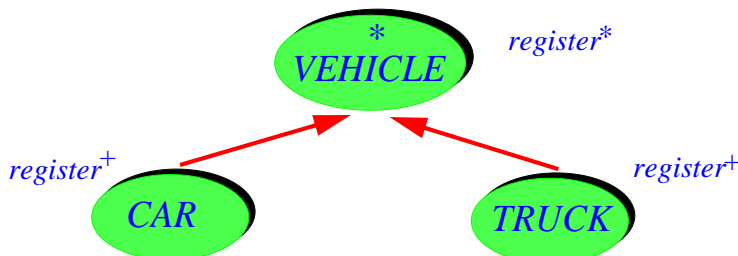
Although similar in many ways to redefinition, this case is more a “definition” (without the *re*) of the feature, since there was no original implementation in the parent. Accordingly, a class that effects a feature inherited as deferred will not list it in a **Redefine** clause.

Some terminology:

- A feature that is not deferred — meaning it has an implementation, either as an attribute or as a non-deferred routine — is **effective**.
- The terms “deferred” and “effective”, originally defined for features, extend to classes: a class is deferred if it has at least one deferred feature; otherwise (if all its features are effective) the class is effective.

Although sufficient for the time being, these definitions will be made more rigorous below. → “*Effective, deferred feature*”, page 309

In graphical representations of system structures, both deferred features and deferred classes will be marked by an asterisk *. Their effectings, as other forms of redeclaration, are marked with a plus sign +.



*Deferred class,
deferred
feature, and
effectings*



As noted above, a class designer may decide to declare a feature as deferred because of either **inability** or **refusal** to provide an implementation. These two cases correspond to the two major uses of deferred routines and classes:

- 1 • You may want to write a class describing an abstract notion, covering several possible implementations. Then you cannot write an effective class, which would require that you provide full implementation information. Some of the features of such a class, and hence the class itself, will be deferred.
- 2 • In other cases, whether or not you have enough information to give the implementation, you prefer to concentrate on the abstract properties of a class and its features, postponing implementation concerns to later.

The next two sections explore these two applications.

10.12 DEFERRED CLASSES FOR DESCRIBING ABSTRACTIONS



The first application of deferred classes supports a central aspect of the Eiffel method, resulting from the use of inheritance as a classification mechanism. Often, classes appearing towards the top of inheritance hierarchies represent general categories, for which various proper descendants will provide specific implementations. The higher-level classes should usually be deferred.



The EiffelBase Library contains numerous such cases. A typical example is class *TREE*, describing the most general notion of tree, independent of any representation. Specific implementations are described by proper descendants of that class, such as *FIXED_TREE* and *TWO_WAY_TREE*, both sketched earlier. Class *TREE* contains a number of deferred features describing operations that cannot be made more precise without committing to a representation. Typical of these is the procedure

On *FIXED_TREE* see [6.2, page 169](#); on *TWO_WAY_TREE*, [10.4, page 263](#).

```

child_put (v: like item)is
    -- Put item v at active child position.
    require
        not_child_off: not child_off
    deferred
    ensure
        replaced: child.item = v
    end

```

which replaces by *v* the value stored in the “active child” (the child at current cursor position) of the current node.

The keyword **deferred**, indicating that the routine is deferred, comes in lieu of an **Effective** body introduced by **do**, **once** or **external**. As the example shows, the **Precondition** and **Postcondition** clauses may still be present; they characterize the semantics of the routine, which all descendant implementations must preserve (in a manner explained below).

Here are two further examples from other ISE Libraries.

EiffelVision contains numerous classes representing various geometrical figures, some simple, some composite. They are all descendants of a deferred class **FIGURE**, usually through one of its heirs **OPEN_FIGURE** and **CLOSED_FIGURE**, still deferred themselves.

See "Reusable Software" for details about these examples.

EiffelParse provides tools for analyzing programs or other structured texts. To build a parser for a particular language, you write classes describing the abstract structure of that language's constructs; for example, a parser for Eiffel will contain classes **EIFFEL_CLASS**, **ROUTINE**, **INSTRUCTION** etc. All such classes are descendants of the deferred class **CONSTRUCT**, through one of three heirs of **CONSTRUCT** describing three kinds of construct (the same as in the Eiffel syntax descriptions of this book):

← "**PRODUCTIONS**", 2.5, page 88.

- **AGGREGATE** describes constructs with a fixed number of parts. For example, in a parser for Eiffel, a class describing the syntax of a **Loop** (where the parts are an **Initialization**, an **Invariant**, a **Variant** and a **Loop_body**) would be written as an heir to **AGGREGATE**.
- **CHOICE** describes constructs whose specimens are chosen from a number of possible constructs. For example an Eiffel **Instruction** is a **Creation**, or a **Call**, or an **Assignment** etc.; the corresponding class in a parser would be an heir of **CHOICE**.
- **SEQUENCE** describes constructs with a variable number of components of the same kind, such as an Eiffel **Feature_declaration_list**, which may consist of zero or more specimens of **Feature_declaration**.

→ The syntax for **Loop** is on page 495.

← The syntax for **Instruction** is on page 228.

← The syntax for **Feature_declaration** is on page 137.

CONSTRUCT is almost fully deferred. The three heirs listed, although still deferred, are "less" deferred since they provide effective routines for parsing the corresponding types of constructs.

As you will remember, it is not possible to have a feature both deferred and frozen, since frozen features may never be redeclared, and deferred features are born for the very purpose of redeclaration.

← "**Feature Declaration rule**", page 162

10.13 DEFERRED CLASSES FOR SYSTEM DESIGN AND ANALYSIS

In the preceding examples, deferred classes were abstracted from effective ones, by removing implementation aspects. In other cases, deferred classes initially exist independently of any implementation. This is the second of the two major applications of deferred classes.

This situation — mentioned earlier as a case of the designer not *wanting* to consider any implementation — arises in particular out of the use of Eiffel as a tool for **system analysis and design**.

At *design* time, you are concerned with the architecture of a system, not its implementation; deferred classes provide an ideal way to express the abstract properties of an architecture, including contracts, without making decisions about representation or algorithms

At a stage even more remote from implementation concerns, deferred classes are an *analysis* tool: to model and analyze a certain category of real world objects, you may write fully deferred classes that capture the abstract properties of those objects. Not only are such classes independent of any implementation; they may in fact be independent of any computerization. It is indeed possible through deferred classes to describe in Eiffel many natural or artificial systems, whether or not they involve computers and software, as long as their structure and semantics are well understood.

"Fully deferred class" means that all the class's features are deferred. In general, a class is deferred as soon as it has one deferred feature, even if some of its other features are effective.

Object-oriented systems analysis may be defined as the discipline of describing systems of any kind through collections of fully deferred classes, connected by client and heir relations (capturing system structure) and characterized by preconditions, postconditions and invariants (capturing system semantics). Although a detailed presentation of these topics falls beyond the goal of this book, the following class sketch should enable you to form a general idea of O-O system analysis.

Extracted from the hypothetical description of a chemical plant, it illustrates the gist of the method, in particular its use of contracts to characterize the known abstract properties of a set of objects. As noted, such a specification is independent from any computer implementation — although it will of course serve as an ideal basis for the software design and implementation process if computerization does occur.



deferred class TANK feature

```

fill
    -- Fill tank with liquid
require
    in_valve.open
    out_valve.closed
deferred
ensure
    in_valve.closed
    out_valve.closed
    is_full
end

```

```
... Other deferred features, such as:  
empty, is_full, is_empty, in_valve, out_valve  
gauge, maximum, ...  
invariant  
  is_full = ((0.97 * maximum <= gauge) and  
             (gauge <= 1.03 * maximum))  
  ... Other invariant clauses ...  
end
```

10.14 EFFECTING A DEFERRED FEATURE

Unless you are using Eiffel just as a modeling language, and do not plan to build software for the system that you first described using deferred classes, you will eventually give these classes proper descendants that **effect** (redeclare as effective) the features they inherit in deferred form.

Any class *C* that inherits a deferred feature from one of its parents may provide a declaration making the feature effective in *C*. (This is a possibility, not an obligation; if the designer of *C* elects to leave some or all of the inherited features deferred, *C* itself will still be a deferred class.)

Effecting a feature is similar to redefining an inherited feature. Here you will not list the feature in a **Redefine** clause since it was not “defined” in the first place.

→ The [“Redeclaration rule”, page 313](#), states what exactly must appear in the **Redefine** clause.

As an example of effecting, one of the many proper descendants of *TREE* that effect *child_put* above is *TWO_WAY_TREE*, where the redeclaration, describing the routine’s implementation for this particular representation, looks like this:

← The deferred version of *child_put* was on page 273.



```

child_put (v: like item)
    -- Make v the value of the node at active child position;
    -- if current node is leaf, create active child with value v.
require else
    is_leaf_or_not_off: (not is_leaf) implies (not child_off)
local
    node: like parent
do
    if is_leaf then
        create node.make (v)
        put_child (node)
        child_start
    else
        child.put (v)
    end
ensure then
    set: child_item = v
end

```



Note the new form of the precondition and postcondition clauses. The precondition of the effective version is the boolean “or” of the original (deferred) routine’s precondition and of the assertion given in the **require else** clause; the new postcondition is the boolean “and” of the original postcondition and of the assertion given in the **ensure then** clause. This is part of the general Redeclaration rule, as given below.

→ [“REDECLARATION AND ASSERTIONS”, 10.17, page 283](#); [“Redeclaration rule”, page 313](#).

For an effecting, as with the redeclaration of *put_child* here, you will not list the feature in a **Redefine** clause.

10.15 PARTIALLY DEFERRED CLASSES AND PROGRAMMED ITERATION

As defined above, a class is deferred as soon as it has at least one deferred feature. But nothing requires it to be **all** deferred: it may contain a combination of deferred and effective features.



This yields one of the most powerful techniques of Eiffel development: producing partially deferred classes which capture what you know for sure about the behaviors and data structures characterizing a certain application area, while leaving open what you do not yet know and what is open to individual variation. You will describe the known aspects through effective features, the variable ones through deferred routines. In particular, an effective routine, covering a known general behavior, may call one or more deferred features, which stand for the variable components of that behavior.



A typical application of this technique appears in many user-interface building systems, where the application software is under the control of an outside loop, sometimes called an **event loop**, which controls the overall scheduling of individual operations: detecting input events, processing these events, updating the screen etc. The event loop is the same for all applications, but each application will define its own version of the individual operations. To implement this scheme elegantly, you may write a deferred class covering the properties of all applications of a certain type, with an effective routine that serves as event loop and calls deferred routines representing the individual operations. Each specific application will then effect these routines, according to its own needs, in a proper descendant of the deferred class.

This scheme is an attractive alternative to the “call-back” mechanisms present in lower-level programming languages.

→ On call-back mechanisms see also [31.8](#), [page 833](#), indicating how to enable an existing call-back mechanism, implemented in another language, to call Eiffel routines.

Another important application of the same idea is illustrated by the **iteration** classes of EiffelBase. These classes provide various iteration mechanisms on arbitrary structures: linear iteration (forward only), two-way iteration, tree iteration (preorder, in order, postorder). For example, class *LINEAR [G]* has iteration procedures such as



```

until_do (action: PROCEDURE [ANY, G];
         test: PREDICATE [ANY, G])
    -- Starting at beginning of structure, apply action to
    -- every item up to but excluding first satisfying test.

do
  from
    start
  until
    after or else test.item ([item])
  loop
    action.call ([item])
    forth
  end
ensure
  found_if_not_after: not after implies test.item ([item])
end

```

The actual implementation in the library class is slightly different as it takes advantage of other iteration procedures.



In this procedure, *action* and *test* are **agents**: objects representing operations to be applied. They both take an argument of type *G*, representing a list item; *action* is a procedure that processes such an item, *test* a boolean-valued function (predicate) that determines whether a certain property is true of the item. A typical call, using *your_integer_list* of type *LIST [INTEGER]* — where EiffelBase's *LIST* is indeed a descendant of *LINEAR* — is:

→ Chapter 27 discusses in detail the notion of agent and its application to iteration.



```

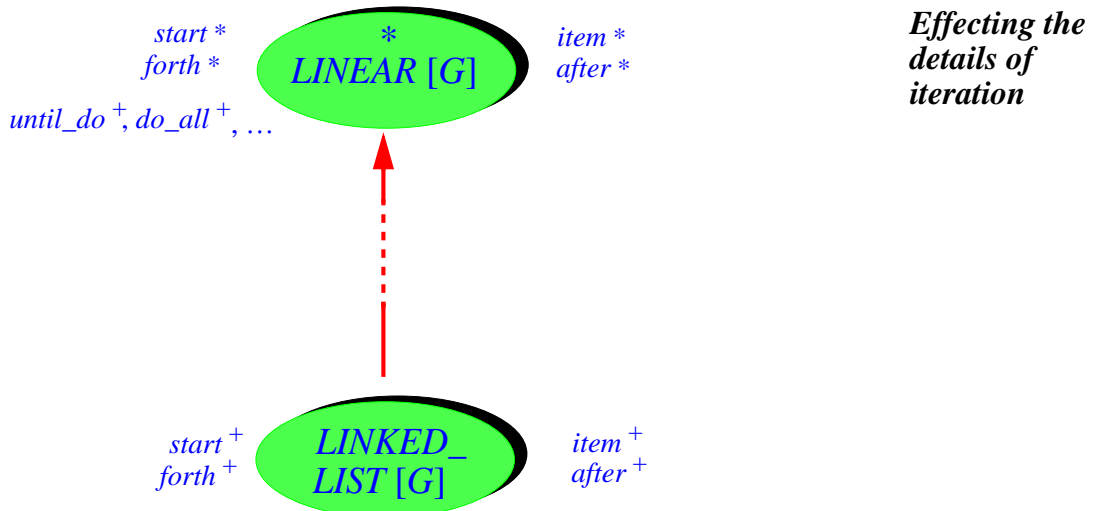
your_list.until_do (~ {INTEGER}.print, ~ is_positive)

```

using two agent arguments, one built from procedure *print* as applicable to class *INTEGER* and the other from a function *is_positive* assumed to be available in the current class to determine whether an integer is positive. This call will print the initial elements of the list, if any, up to and excluding the first positive one.

Along with *until_do*, traversal classes such as *LINEAR* and their descendants provide other iterators: *do_until*, *do_all*, *while_do*, *do_while*, *do_if*, *exists*, *for_all*.

LINEAR is a very general deferred class, requiring its effective descendants to provide features representing basic traversing steps: *start* to start traversal, *forth* to advance by one position, *item* to yield the item at cursor position, *after* to find out if the traversal has passed the last item. All the classes of EiffelBase and other libraries that describe traversable data structures such as chains, lists and many others are its descendants.



Effective procedures such as *do_until* define traversal patterns. Deferred features such as *start* and *item* describe the ingredients to be used in any particular application of these patterns.

To provide an actual iteration mechanism over a certain concrete structure — such as *LINKED_LIST* or *CIRCULAR_LIST* — it suffices to inherit from *LINEAR* or another of the traversal classes, and to effect the deferred features to describe the specific machinery of iteration processing on the chosen structure: how to start an iteration, move on to the next element, access the current element, and determine end of traversal, based on the specific implementation retained.

10.16 REDECLARATION AND TYPING

The two redeclaration mechanisms studied so far in this chapter, redefinition and effecting, share many properties; both are ways to refine the original declaration of an inherited feature, and both are subject to the same constraints.

Two important properties apply in both cases:

- The type constraint, which we will now explore informally.
- The rule on semantics of updated assertions, studied in the next section.

The formal version of these combined properties is the Redeclaration rule, [→ “Redeclaration rule”, page 313.](#)
given in full later.

First, the type constraints. Let f be a precursor (parent’s version) of an inherited feature. Assume that the signature of f (in the parent) is

[A, B], [C]



Recall that the first part, here [A, B], lists the arguments types for a routine (it is empty for an attribute), and that the second part, here C, lists the result type for an attribute or a function (it is empty for a procedure). [← “THE SIGNATURE OF A FEATURE”, 5.13, page 148.](#)

Then the Redeclaration rule will state that if you redeclare f into a new feature, the new signature must conform to the precursor’s signature.

Conformance, a key concept of the type system, is discussed in detail in a [later chapter](#), but the basic idea is straightforward: a type conforms to another if its base class is a descendant of the other’s; a signature conforms to another if it has the same number of arguments and results and every type in the first signature conforms to its counterpart in the other. For example, the signature. [→ See chapter 14 on conformance, particularly “EXPRESSION AND SIGNATURE CONFORMANCE”, 14.4, page 386.](#)



[X, Y], [Z]

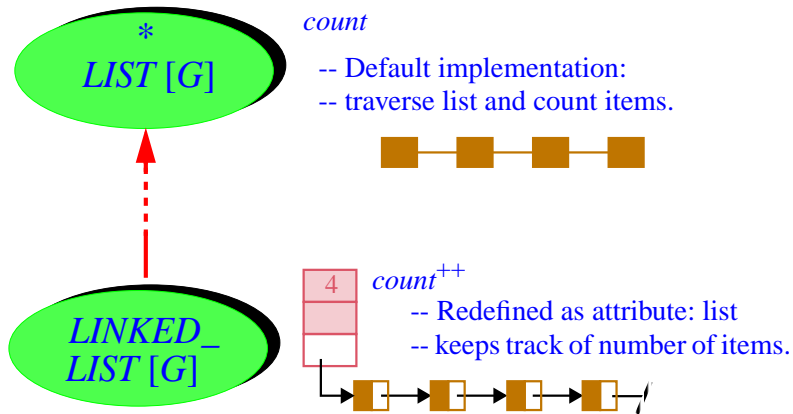
will conform to the above if type X conforms to A , Y to B and Z to C .

This rule means in particular that a redeclaration may not change the number of arguments and results, and may only replace types of arguments or results by conformant types. You can obtain the effect of changing the number of arguments and results by using [tuples](#). [→ Chapter 13.](#)

The Redeclaration rule also prohibits the redeclaration of an attribute into a function. It is permitted, however, to redeclare a function into an attribute; in this case the preceding constraint implies that the precursor function must have been without arguments (otherwise the new signature could not conform). The attribute used for the redeclaration may be variable or constant.



Redeclaring a function into an attribute is a useful and common practice. Here is a typical case. Feature *count*, present in most classes of EiffelBase, gives the number of elements of a structure. Classes high in the inheritance graph, such as *LIST*, the deferred class describing lists independently of any representation choice, declare *count* as a function, which traverses a structure to count its elements. The implementation of effective descendants such as *LINKED_LIST* keeps a record of a list's element count in the list header; these descendants accordingly redefine *count* into an attribute.



Redefining a function into an attribute



This is typical of why you may want to redefine a function into an attribute. A class *B* (*LIST* in this example) has a function *f* that computes some information about the corresponding objects (in the example, the number of items in a list). You devise a new implementation, represented by a descendant *C* of *B*, that keeps the information up to date in a field of the object, represented by an attribute of *C*. (In the example, *C* is *LINKED_LIST*, which keeps a record of the number of items in the list header.) In most object-oriented languages, you would have to define this attribute as a new feature of the class, and redefine *f* into a function that returns its value. But there is no need for two separate features, since they represent the same information: in Eiffel, *C* will simply redefine *f* into an attribute.

This is all in line with the Uniform Reference principle, which states that attributes and functions without arguments should be indistinguishable from the outside, as they are just two alternative ways to provide a query, differing in implementation technique, not relevance to clients.

In implementing such a scheme, *C* must ensure that the value of the query will always be up to date when clients access it; this means that any procedure whose execution may have an effect on the query's value must be redefined in *C* to update the attribute. (In our example, *LINKED_LIST* must redefine all the procedures that insert or remove items, to make sure they increment or decrement *count*.) To make sure that you don't forget any

such redefinition, take a look at procedure postconditions: in well-written classes, the postcondition of any procedure should indicate whether the procedure has any effect on any particular query. For example the postcondition of *remove*, which deletes an item from a list, will have a clause of the form $count = \mathbf{old\ count} - 1$. This signals that together with any redefinition of *count* into an attribute there must be a redefinition of *remove* to include the instruction $count := count + 1$ or equivalent.

Sometimes the *B* version of *f* is deferred; this is the case in the above example if instead of *LIST* we consider its ancestor *SEQUENTIAL*, where *count* is deferred. (Deferred features are syntactically treated as routines, although if they have no arguments they are just features for which we have refused to choose yet between attribute and routine implementations.)

Why then (in spite of the Uniform Reference principle) does the type constraint prohibit the reverse form of redefinition – changing an attribute into a function? One of the reasons is that we would be unable, were this permitted, to make sense of certain routines inherited from parents. Assume class *B* with features



<i>a</i> : <i>INTEGER</i> ;	-- <i>a</i> is an attribute
<i>set_a</i> is do <i>a</i> := 0 end	-- <i>set_a</i> assigns to <i>a</i>

Then if *C*, an heir of *B*, were allowed to redefine *a* into a function, but did not redefine *set_a*, there would be no way to execute *set_a* applied to instances of *C*, since one may not assign to a function. For the same reason, it is not permitted to redefine a variable attribute into a constant attribute.

10.17 REDECLARATION AND ASSERTIONS

The other fundamental property of redeclaration governs the **Precondition** and **Postcondition** clauses of a redeclared routine. Such assertions, if present, may not be of the basic forms using just **require** and **ensure**; instead they must use **require else** and **ensure then**. Consider a routine redeclaration. If it contains new assertion clauses, they must be of the form

← See chapter 9 about **Precondition** and **Postcondition** clauses and their semantics in the absence of redeclaration.

require else <i>alternative_precondition</i>
ensure then <i>extra_postcondition</i>

expressing the new assertions as a variation on the precursors' assertions.

What kind of variation? Consider a routine redeclaration and let pre_1, \dots, pre_n be the precursors' preconditions and $post_1, post_n$ be the precursors' postconditions. (Remember that in most practical cases there is only one precursor, so that n is 1; only with a join of deferred features may there be two or more precursors.) Assume that new assertion clauses are present, of the above form. Then the redeclared routine will be considered to have the precondition and postcondition.

With sharing in repeated inheritance, there may also be two or more precursors, but this is not a case of redeclaration. See the definition of "inherited features" on page 470.

alternative_precondition **or else** pre_1 **or else** ... **or else** pre_n
extra_postcondition **and then** $post_1$ **and then** ... **and then** $post_n$

In other words, the precondition is or-ed with the original preconditions, and the postcondition is and-ed with the original postconditions. For the precondition, the use of operator **or else** rather than plain **or** guarantees that the assertion is defined, with value true, whenever one of the operands has value true, even if a subsequent one is not defined; similarly, **and then** for postconditions guarantees that any false operand makes the whole assertion false even if a subsequent one is not defined.

→ or else and and then are the "semi-strict" versions of plain or and and. See "SEMIS-TRICT BOOLEAN OPERATORS", 28.6, page 774.



If the assertion clauses are missing in a redeclaration, the convention is that the redeclared routine is considered to have *False* as *alternative_precondition* for an absent **Precondition** part and *True* as *extra_postcondition* for an absent **Postcondition**. Because of the rules of boolean algebra, this means keeping the corresponding precursor assertions. (Or-ing a boolean value with **false**, or and-ing it with **true**, does not change the condition.)



The use of **require else** and **ensure then** in a redeclared routine reflects an important part of the Design by Contract method underlying Eiffel. Redeclaring a routine means subcontracting to a descendant the job which clients originally entrusted to the precursor. A good subcontractor will do as well as better for clients as agreed in the original contract (involving the precursor). This means:

See "Object-Oriented Software Construction" and "Design by Contract" (references in appendix D) and the notion of subspecification in .page 236.

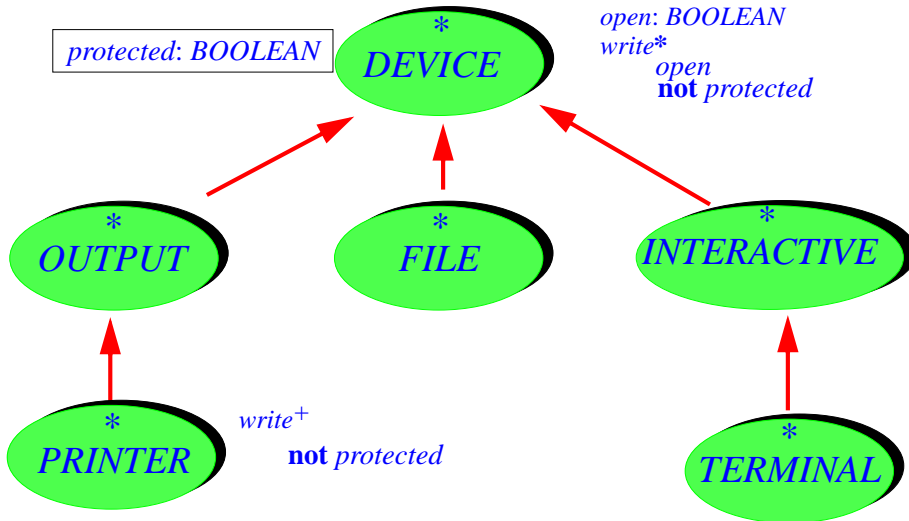
- Keeping or weakening the precondition, so as not to impose any new requirements on the original clients.
- Keeping or strengthening the postcondition, so as to return a result that is as good as what was originally promised to the clients.

The or-ing and and-ing automatically guarantee these rules, since p **or else** q is always weaker than or equal to p , and p **and then** q is always stronger than or equal to p .

A condition is stronger than or equal to another if it implies it, in the sense of boolean implication. "Weaker than or equal" is the inverse relation.

Examples of strengthening the postcondition routine are very common. In fact, almost any redefinition of a routine's implementation, or effecting of a deferred routine, will do something more — such as updating new attributes —, leading to a postcondition stronger than the original. The added properties should appear in the **ensure then** clause.

As an example of weakening the precondition, assume the inheritance hierarchy illustrated below. Procedure *write*, in *DEVICE*, has two clauses in its **Precondition**: the device must be open, and it must not be protected. Examples of devices are output devices, interactive devices and files.



**Precondition
weakening**

Assume that printers, a kind of device, may not be write-protected. (The invariant of class *PRINTER* should include the clause *not protected*.) The precondition of *write* for *PRINTER* may then be weakened to just *open*.

To achieve this, just include in the redefined version of *write* in *PRINTER* the **Precondition**

require else open

The above semantic rule gives, as actual precondition:

open or else (not protected and then open)

which has the same value as just *open*.



If a declaration introduces an immediate feature — in other words, it's not a redeclaration — the **require else** and **ensure then** forms are still permitted, having the same effect as just **require** and **ensure**.

← “*SYNONYMS AND MULTIPLE DECLARATION*”, 5.18, page 159



Since the longer forms are normally intended for redeclarations, you might expect a validity constraint which makes them invalid for an immediate feature. But there is no such constraint, among other reasons because this tolerance makes it easy to declare *synonym features* of which one is immediate and the other inherited. A declaration may be of the form



```

inherited, immediate
require else
    pre
do
    ...
ensure then
    post
end
  
```

where *inherited* is a feature inherited from a parent, for which this declaration will be a redefinition or effecting, but *immediate* is a new feature. The **require else** and **ensure then** form are compulsory because of *inherited*. But they also work for *immediate*, being understood as **require** and **ensure**.



Remember that there is no tolerance in the reverse direction: for a redeclaration, only the **require else** and **ensure then** forms are permitted.

----- UPDATE --- Assertion declaration, as we have now studied it, complements another property involving the combination of assertions and inheritance: the definition of “invariant of a class” as containing not only the local **Invariant** clause, but also any others inherited from parents. Together with the rules just seen on assertions of redeclared routines, this ensures that inheritance and redeclaration maintain the fundamental semantic properties of a class and its features, as expressed by the assertions.



We need to consider one more case in the combination of redeclaration and assertions. What happens, when you redefine a function without arguments into an attribute, to the function's assertions if any? Since an attribute has no precondition, we may consider that the precondition is changed to *True*; this is consistent with the preceding discussion since *True* is weaker than any other assertion. For a postcondition, the situation is different: the only way to express that the attribute's possible values will

satisfy the corresponding condition (with the attribute's name substituted for *Result*) is to make it part of the invariant of the class. The definition of class invariants took care of this by stating that the redefinition of a function into an attribute automatically adds the adapted postcondition to the invariant of the redefining class, replacing any occurrence of *Current* by the attribute name. So if a function was of the form



```

last_value: INTEGER
do
    ...
ensure
    Result >= 0
end

```

and a descendant *C* of its class of origin redefines *last_value* into an attribute, the invariant of *C* will automatically include the clause

```

last_value = 0

```

--- ADD DISCUSSION OF EFFECT OF REDECLARATION ON "ONLY" POSTCONDITION CLAUSES

10.18 RULES ON INHERITED ASSERTIONS

Unfolded form of an assertion

The **unfolded form** of an assertion *a* of local unfolded form *ua* in a class *C* is the following Boolean_expression:

- 1 • If *a* is the invariant of *C* and *C* has *n* parents for some $n \geq 1$: *up*₁ **and** ... **and** *up*_{*n*} **and then** *ua*, where *up*₁, ... *up*_{*n*} are (recursively) the unfolded forms of the invariants of these parents, after application of any feature renaming specified by *C*'s corresponding Parent clauses.
- 2 • If *a* is the precondition of a redeclared feature *f*: the combined precondition for *a*.
- 3 • If *a* is the postcondition of a redeclared feature *f*: the combined postcondition for *a*.
- 4 • In all other cases: *ua*.

The unfolded form of an assertion is the form that will define its semantics. It takes into account not only the assertion as written in the class, but also any applicable property inherited from the parent. The “local unfolded form” is the expression deduced from the assertion in the class itself; for an invariant we “and then” it with the “and” of the parents, and for preconditions and postconditions we use “combined forms”, defined next, to integrate the effect of **require else** and **ensure then** clauses, to ensure that things will still work as expected in the context of polymorphism and dynamic binding.

The earlier definitions enable us to talk about the “precondition of” and “postcondition of” a feature and the “invariant of” even in the absence of explicit clauses, by using **True** in such cases. This explains in particular why case 1 can mention “the invariants of” the parents of *C*.

Assertion extensions

For a feature *f* of a class *C*:

- If *C* redeclares *f* with a non-empty **Precondition** (starting with **require else**), the **precondition extension** of *f* in *C* is the corresponding **Assertion**.
- If *C* redeclares *f* with a non-empty **Postcondition** (starting with **ensure then**), the **postcondition extension** of *f* in *C* is the corresponding **Assertion**.

In all other cases, the precondition extension of *f* in *C* is **False** and the postcondition extension of *f* in *C* is **True**.

These are the forms that routines can use to override inherited specifications while remaining compatible with the original contracts for polymorphism and dynamic binding. **require else** makes it possible to weaken a precondition, **ensure then** to strengthen a postcondition, under the exact interpretation explained next.

Covariance-aware form of an assertion extension

The **covariance-aware form** of an inherited assertion a is:

- 1 • If the enclosing routine has one or more arguments x_1, \dots, x_n redefined covariantly to types U_1, \dots, U_n : the assertion $((x_1: U_1) y_1$ **and** ... **and** $\{x_n: U_n\} y_n)$ **and then** a' where y_1, \dots, y_n are fresh names and a' is the result of substituting y_i for each corresponding x_i in a .
- 2 • Otherwise: a .

A covariant redefinition may make some of the new clauses inapplicable to actual arguments of the old type (leading to “catcalls”). The covariance-aware form avoids this by ignoring the clauses that are not applicable. The rule on covariant redefinition avoid any bad consequences.

Combined precondition, postcondition

Consider a feature f redeclared in a class C . Let f_1, \dots, f_n ($n \geq 1$) be its versions in parents, pre_1, \dots, pre_n the covariance-aware forms of (recursively) the combined preconditions of these versions, and $post_1, \dots, post_n$ the covariance-aware forms of (recursively) their combined postconditions.

Let pre be the precondition extension of f if defined and not empty, otherwise **False**.

Let $post$ be the postcondition extension of f if defined and not empty, otherwise **True**.

The **combined precondition** of f is the **Assertion**

$(pre_1$ **or** ... **or** $pre_n)$ **or else** pre

The **combined postcondition** of f is the **Assertion**

(old pre_1 **implies** $post_1)$

and ... **and**

(old pre_n **implies** $post_n)$

and then $post$

The informal rule is “perform an *or* of the preconditions and an *and* of the postconditions”. This indeed the definition for “combined precondition”. For “combined postconditions” the informal rule is sufficient in most cases, but occasionally it may be too strong because it requires the old postconditions even in cases that do *not* satisfy the old preconditions, and hence only need the new postcondition. The combined postcondition as defined reflects this property.

10.19 UNDEFINING A FEATURE

You may redefine an inherited feature; you may also, if it was effective, *undefine* it.

As the Redeclaration rule will express precisely, you may not use redeclaration to turn an effective feature into a deferred one, discarding its inherited implementation. In other words, redeclaration cannot decrease the “effectiveness level” of a feature: it can take the status of an inherited feature from deferred to deferred (redefinition), effective to effective (redefinition), or deferred to effective (effecting), but never from effective to deferred.

→ Clause 5 of the Redeclaration rule, page 313.

In some cases, however, this is desirable; when inheriting a feature, you may wish to give it back its virginity, by pretending you inherited it as deferred, even though its precursor (the parent’s version) is in fact effective.

Undefinition serves this goal. To undefine one or more effective features inherited from a parent, just list them in the **Undefine** subclause of the corresponding **Parent** part, as in



```
class C inherit
  B
  rename
  ...
  undefine
    f, g, h
  redefine
  ...
  ... Other subclauses of Feature_adaptation ...
end
... Other parents and rest of class...
```

*To remember this order, note that all subclauses except **Rename** refer to features by their final names, so **Rename** should come first. Since, as seen next, an undefined feature may then be redefined, **Undefine** must come before **Redefine**.*

In the optional subclauses of a **Feature_adaptation**, **Undefine** comes after **Rename** and **New_exports**, and before **Redefine**.

Here f , g , h must be features that are effective in B . The effect of the above **Undefine** subclause is that C obtains these features from B as if they had been deferred rather than effective in that class; the process does not change the features' signature and specification.

It is possible to apply both undefinition and redefinition to the same inherited feature; this is useful if you want to make an inherited feature deferred and also change its signature or specification, as in



```
class E inherit
  B
      undefine
        f
      redefine
        f
end
feature
  f(x: U) is deferred end
end
```

where the B version of f had an argument of type T rather than U , assuming (as required by the Redeclaration rule) that U conforms to T .

This leads to a precise definition of the inherited status of a feature:



Inherited as effective, inherited as deferred

An inherited feature is **inherited as effective** if it has at least one precursor that is an effective feature, and the corresponding **Parent** part does not undefine it.

Otherwise the feature is **inherited as deferred**.

10.20 REDEFINITION AND EFFECTING

We can now define precisely the two variants of redeclaration:

Effect, effecting

A class **effects** an inherited feature f if and only if it inherits f as deferred and contains a declaration for f that defines an effective feature.

Effecting a feature (making it *effective*, hence the terminology) consists of providing an implementation for a feature that was inherited as deferred. No particular clause (such as **redefine**) will appear in the **Inheritance** part: the new implementation will without ado subsume the deferred form inherited from the parent.



Redefine, redefinition

A class **redefines** an inherited feature f if and only if it contains a declaration for f that is not an effecting of f .

Such a declaration is then known as a **redefinition** of f

← Two feature names are “the same” if they are identical or differ only by letter case. See “[Same feature name, same operator, same alias](#)”, page 153.

Redefining a feature consists of providing a new implementation, specification or both. The applicable **Parent** clause or clauses must specify **redefine** f (with f 's original name if the new class renames f .)

Redefinition must keep the inherited status, deferred or effective, of f :

- It cannot turn a deferred feature into an effective one, as this would fall be an effecting.
- It may not turn an effective feature into a deferred one, as there is another mechanism specifically for this purpose, *undefinition*. The Redeclaration rule enforces this property.

As defined earlier, the two cases, effecting and redefinition, are together called *redeclaration*.

10.21 THE JOIN MECHANISM

The notion of deferred feature yields a useful technique: *feature join*, allowing a class to merge several inherited features into just one.



The join mechanism supports an important aspect of object-oriented architecture design: the fusion of abstractions. The abstractions that need to be combined will come from different hierarchies of deferred classes.

The EiffelBase library, based on combinations of three such hierarchies, provides typical opportunities for such fusion. The hierarchies correspond to complementary classification criteria for general-purpose “container” data structures:

- **Storage**, characterizing the representation properties of a container structure (fixed size, variable size but bounded, unbounded but finite, potentially infinite).
- **Access**, characterizing the methods through which clients store and retrieve elements (in last-in-first-out for stacks, through a key for hash tables etc.).
- **Traversal**, characterizing ways of exploring the container exhaustively (forward, backward, postorder, preorder and others.).

You can obtain a particular type of effective container by multiple inheritance from classes of these three categories. For example, a “fixed-size list” has fixed-size storage, access by index and other techniques, and forward traversal.

A container data structure, such as a queue or a hash table, serves to store and retrieve objects. Some of the most important kinds of container data structure are covered by the classes of EiffelBase. See “Reusable Software” for details.

In this process of combining abstractions, it will often be useful to merge inherited deferred routines if they correspond to the same notion in the descendant. For example, the deferred EiffelBase class *CHAIN* (describing sequential structures such as lists) inherits from two deferred classes that both have an *item* function returning the item at cursor position:

- *ACTIVE*, from the Access hierarchy, describe structures with a client-controlled “cursor” position. Procedures are available to move the cursor to various elements. In this class, *item* denotes the value of the element at cursor position.
- *BIDIRECTIONAL*, from the Traversal hierarchy, describe structures that are sequentially traversable both forward and backward. In this class, *item* denotes the value of the current element at each step of a traversal operation.

Class *CHAIN* combines these two concepts and inherits both *item* functions. Normally, this would be considered a name clash, which we would have to remove through renaming. But here the clash is harmless, in fact desired, since for a *CHAIN* the two concepts are compatible. If the features were effective, we would have to choose between conflicting implementations; but they are both deferred, so we have no such problem. We can simply merge — “join” — them into one.

→ “NAME CLASHES”,
10.23, page 297.

It is valid, then, to write *CHAIN* as heir to both *BIDIRECTIONAL* and *ACTIVE* even without renaming the deferred *item* routines, which will yield a single deferred routine in *CHAIN*:



```
deferred class CHAIN [T] inherit
  BIDIRECTIONAL [T]
    -- BIDIRECTIONAL has a deferred routine item
  ...
  ACTIVE [T]
    -- ACTIVE has a deferred routine item
  ...
... Other parents and rest of class text omitted ...
```

Here is another interesting application. Occasionally you will need to effect an inherited procedure to do nothing at all. For example a descendant of a general-purpose iteration class, as studied earlier in this chapter, might not need a particular initialization operation, provided in the ancestor by a procedure *prepare*. You can manually effect *prepare* into a procedure that does nothing. But it is simpler to use a join with the procedure *do_nothing* from class *ANY*, whose implementation faithfully respects its name:



```
class SIMPLE_ITERATOR inherit
  GENERAL_ITERATOR
    rename prepare as do_nothing end
... Other parents and rest of class text omitted ...
```

That's all you have to do: renaming *prepare* causes a join with *do_nothing* and the associated effecting.



To be joined, inherited features must have the same final name in the class that performs the join. In the above case both precursors were called *item* in the parents, so no particular action was required from the designer of class *CHAIN* with respect to their names. In other cases you might want to join two deferred features that have different names, say *f* and *g*, in the respective parents. You should then use renaming to make sure that the features are inherited under the same final name:

```
-- C may be deferred or not (see below)
... class C inherit
  A
    rename
      f as new_name
    ...
  end
  B
    rename
      g as new_name
    ...
  end
```



If *C* inherits and joins two or more deferred features, the net result for *C* is as if it had inherited a single deferred feature. In the absence of further action from *C*, that feature remains deferred. *C* may of course provide an effective declaration, killing several abstract birds with one concrete stone by using a single redeclaration to effect several features inherited as deferred.

← “Inherited as deferred” was defined (page 291) to mean: either coming from deferred precursors, or explicitly undefined.

More generally, *C* may treat the result of the join as it would any other inherited deferred feature. *C* may in particular redefine the feature to change its signature while leaving it deferred. In that case *C* must list all the inherited features in the **Redefine** subclauses of their respective **Parent** parts.



The join mechanism imposes easily justifiable conditions on features to be joined in this way: they must be deferred (after possible undefinition, as detailed in the next section), inherited under the same name (after possible renaming), and equipped with the same signature (after possible redeclaration). The formal rule expressing these requirements is the Join rule, described later in this chapter.

→ “Join rule”, page 319.

10.22 MERGING EFFECTIVE FEATURES

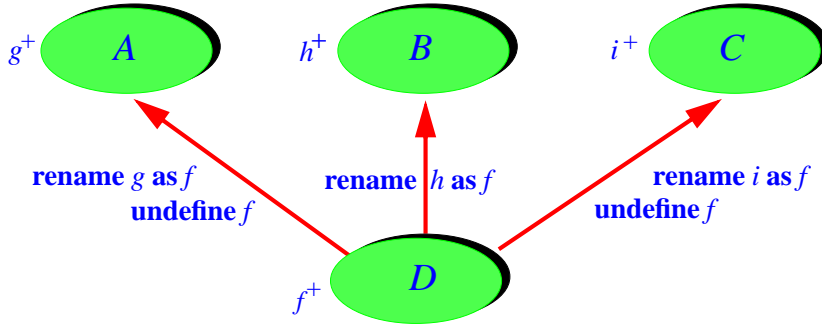


As introduced so far, the join mechanism applies only to deferred routines. The reason is obvious: an attempt by a class *D* to join two effective features inherited from parents of *D* may yield an ambiguous result in the absence of a clear universal criterion for choosing one of the two inherited implementations over the other.

What happens, however, if when you design *D* you *do* know which of the versions you want to override the other in *D*? Then the merging should not raise any particular problem.

The undefinition mechanism makes this possible. Here is an illustration of the scheme, used in this case to join three features:

← “UNDEFINING A FEATURE”, 10.19, page 290.



Merging and overriding

We want to merge the three inherited features by renaming all of them into a single name, *f*. But the originals were all effective, yielding three implementations of which we may retain only one in *D*. To discard the *A* and *C* implementations, *D* undefines them, leaving the *B* version as the undisputed victor.

In the simplest case, there are only two competing features in parents *B* and *C*, and they already had the same name *f* in these parents. If you want the *B* version to take over in *D* all you need is to undefine the *C* version:



```
class D inherit
  B
  -- B has an effective feature f
  C
  undefine f end
feature
  ...
end
```

Although *f*'s precursor in *B* was effective, the undefinition causes *f* to be “inherited as deferred” from *C*. The *B* version provides an effecting.



An application of this technique will appear in repeated inheritance when a class inherits conflicting versions of the same feature, and the class designer wants to retain only one of these versions.

→ See the beginning of 16.5, page 442.

The general rule is the natural one (although we must wait until a full definition of repeated inheritance to express it rigorously): inheriting two or more features under the same name may only be invalid — a case of **name clash** — if more than one is inherited as effective. If, after possible undefinition, they are all deferred, or all deferred except for one effective version, then we have a valid case of join, since there is no conflict of implementations: we have either no implementation or one. In the latter case the effective version will serve as common implementation for all the features inherited as deferred. *→ The next section discusses name clashes.*

The examples have illustrated one way to reconcile conflicting effective versions from parents: undefine all but one of them. This is like a competition where one of the rivals win. There is another way — as in business or in war — to resolve a competition: a new entrant overcomes everyone else. The technique here will be to use **redefinition** rather than undefinition: redefine all the conflicting inherited versions into a new one. The last example becomes:



```

class D inherit
  B
    redefine f end
  C
    redefine f end
feature
  f
    do
      ... "Redefined algorithm" ...
    end
  ...
end

```

As you may have noted, it actually doesn't make any difference here if we replace either or even both of the **redefine** keywords by **undefine**. If we undefine one of the features, the other takes over, but gets redefined. If we undefine both, they are inherited as deferred, and hence joined; but then the declaration of *f* effects both.

10.23 NAME CLASHES

Now that we have seen the join mechanism we are in a position to define precisely the notion of **name clash** of features under multiple inheritance, and see what kinds of name clashes are permitted. From the previous section we know the rough form of the rule: a name clash in a class *D* between two or more inherited features will be OK, leading to a join of all of them, if they all have compatible signatures (so that we may indeed join them into a common version) and, taking any undefinitions into account:

- Either the resulting features, except possibly one, are all deferred.
- Or if this is not the case, meaning that two or more versions remain effective in *D*, then *D* redefines all of them into a common version, as in the example class text above.

The discussion of repeated inheritance will also add a permissible case: the “false alarm” resulting from features that come from different parents but are really the *same feature* inherited from a common ancestor. In the absence of conflicting redefinitions this can cause no trouble.

Let’s see the precise form of the rule. The general guideline is the **no overloading principle**, dictated (although it may at first sound like an advertisement for a mutual fund) by criteria of clarity and simplicity.

Overloading — the possibility for a single name to denote several features within the context of a given class — defeats the principles of object technology, running into conflict with the more powerful forms of *dynamic* overloading provided by polymorphism and dynamic binding. Introducing in-class overloading is probably the biggest mistake that one can make in the design of an O-O language.

In the absence of inheritance, the no-overloading principle is easy to enforce: all the features declared in a class must have different names. With single inheritance, we add the rule that no inherited feature may have the same final name as a feature of the class; renaming provides an easy way to correct any such potential conflict. With multiple inheritance, this last rule must still apply between the class and each of its parents, but in addition we have to take into account the case of conflicts between names of features in the parents themselves. This is what we call a name clash:



Name clash

A class has a **name clash** if it inherits two or more features from different parents under the same final name.

Since final names include the identifier part only, aliases if any play no role in this definition.

Name clashes would usually render the class invalid. Only three cases may — as detailed by the validity rules — make a name clash permissible:

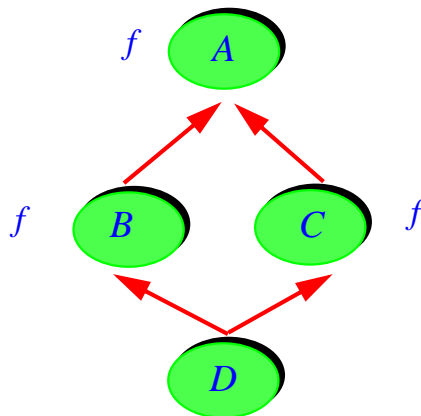
- At most one of the clashing features is effective.
- The class redefines all the clashing features into a common version.
- The clashing features are really the same feature, inherited without redeclaration from a common ancestor.

This property is not expressed as a separate validity constraint since it follows from the Join rule given at the end of this chapter, and the complementary mechanisms discussed in the repeated inheritance chapter.

In the first permissible case, the clash involves only one implementation, or none; if the signatures are compatible, we may join all the features into a single one, with no particular difficulty (and without departing from the no-overloading principle). The second case is similar: joining through redefinition.

In the third case, we don't have a real clash at all, only the appearance of one, as if being scared in an empty house by a moving figure that turns out to be our own reflection in the mirror. This case arises out of repeated inheritance (as studied in a later chapter) in the situation represented on the figure:

→ Chapter 16 explores repeated inheritance.



D seems to inherit two features *f* from both its parents *B* and *C*, but they are not really different features, simply the same feature inherited from a common ancestor *A*, and not redeclared anywhere in the process. As we may expect, and the rules of repeated inheritance will state precisely, *D* inherits a single feature *f*, so this case causes no difficulty. Outside of these three cases, however, a name clash is always prohibited. In the typical situation

A name clash that isn't really one.

```
class C inherit
  A
  B
... Rest of class omitted ...
```

where both *A* and *B* have a feature with the same name *fname*, class *C* will be invalid. It's quite easy to get rid of the name clash:

- Often you will want the features to remain distinct in *C*, because they indeed correspond to different operations; their sharing of a common name is just an unfortunate coincidence, a kind of pun. Then you will simply rename one, or both.
- Sometimes, however, it's not just a pun: in *C* you really want the clashing features to be merged into one. Then, if the signatures are compatible, you can rely on the join mechanism by undefining either one; the other's implementation will take over. You may also undefine both, leaving *C* or one of its own proper descendants in charge of effecting the joined result.

10.24 ADDING TO INHERITED BEHAVIOR: PRECURSOR

The last mechanism of this chapter, **Precursor**, simplifies writing a routine's redefinition when the new implementation relies on the original one.

The need for a precursor mechanism

In studying redefinition we have seen that you can override a routine's inherited implementation (as well as its signature and contract). The new implementation may be completely different from the original one; but fairly often it just extends it, performing the same actions as the original plus some others, with a redefinition of the form

You may redefine features of all kinds, but this section only applies to routines.

```
your_routine
do
  "Something else"
  "Whatever the original version did"
  "Yet something else"
end
```

If you need new actions only after the original processing there is no “Something else”; if only before, there is no “Yet something else”.

A typical example would be the redefinition, in a proper descendant *TAXABLE_INVESTMENT* of a class *INVESTMENT*, of the procedure *sell*, where the new version performs what the original version did but must also compute the tax penalty associated with the sale of a stock.

The **Precursor** mechanism provides you, when writing redefinitions of this kind, with a simple way to include “Whatever the original version did” in the actions of the new routine body. Without **Precursor**, you would have two ways of achieving the intended effect:

- You could simply repeat the original algorithm in the body of the new version, in lieu of the line that reads “Whatever the original version did” above. This works but has the usual effects of code duplication: making the software bigger and less readable (since the reader doesn’t immediately realize that a certain element is not original but the verbatim replication of something else); tediousness for the developer; and, most damaging, the need to remember, if the original changes, that you must update all duplicates as well, with the risk of forgetting some.

As you will have noted, one of the recurring fetures of the Eiffel method is its phobia of unnecessary replication. Genericity, inheritance and other reuse mechanism are all intended to make sure that what needs to be said is said well, and said once.

- You can also use the replication mechanisms of repeated inheritance, studied in the [relevant chapter](#), to keep a duplicate of the original feature, along with the redefined version. This approach avoids the drawbacks of the preceding technique; it was indeed the recommended method in early versions of Eiffel, and remains appropriate in some cases. For most common applications, however, it is overkill, and **Precursor** provides a simpler solution.

→ [“KEEPING THE ORIGINAL VERSION OF A REDEFINED FEATURE”, 16.8, page 451](#)

Precursor basics and examples

That solution is in fact disarmingly easy: in a routine redefinition, **Precursor** stands for what was written above as “Whatever the original version did”. The form of the construct is simply the reserved word *Precursor*, followed by a list of actual arguments if any. So you can write the first example sketch above, for a routine with no arguments, as just

```

your_routine
do
    “Something else”
    Precursor
    “Yet something else”
end

```

For a routine with arguments, the redefinition might look like



```

sell (share_count: REAL; selling_price: PRICE)
    -- Record sale of share_count shares at selling_price.
  do
    Precursor (share_count, selling_price)
    compute_tax
  end

```

These examples illustrate how to use the *Precursor* reserved word: exactly like you would use the feature name (*new_version* and *sell* in these examples) for a new call to the corresponding routine, such as the calls *new_version* and *sell* (*share_count*, *selling_price*). Appearing in the body of the routine, these calls would be recursive — leading in fact, as written, to infinite recursion —, but instead of the feature name we use *Precursor* which yields the desired effect, calling the original version.

In the *sell* example the arguments to *Precursor* are the same as the formal arguments to the procedure, *share_count* and *selling_price*, meaning that you call the precursor with the same arguments that were passed to you. This is not a general requirement; in other circumstances you may pass to *Precursor* any arguments of the appropriate types.

These two examples use procedures. The mechanism works just as well with functions:



```

profit (share_count: REAL; selling_price: PRICE): AMOUNT
    -- Profit from sale of share_count shares at selling_price.
  do
    Result := Precursor (share_count, selling_price)
               - tax_penalty (share_count, selling_price)
  end

```

The result of the *Precursor* call is the result returned by a call to the original version of the routine, with the given arguments.

The *Precursor* construct is valid only in the case illustrated by these examples: the body of the redefinition of a routine, in which it denotes the original implementation in the parent.



Outside of this case a class may not refer to ancestor versions of its features (as provided by the “super” variables of some object-oriented languages, notably Smalltalk) because this would impair the consistency of the notion of class. A class is entirely defined by its features; how these features were arrived at through inheritance is internal information, and the original versions are not part of the information associated with the class. (They will often violate the contracts associated with the class, in particular by failing to maintain its invariant, typically stronger than the ancestors’ invariants.) The only justification for accessing a parent version is that we may need the old implementation to define a new one, through the **Precursor** construct.

The precursor of a redefined feature is *not* a feature of the current class. If you do want to keep both the original and the redefinition as features of the class, you can, but you have to use a different mechanism: repeated inheritance, as explained in a later chapter.

→ “KEEPING THE ORIGINAL VERSION OF A REDEFINED FEATURE”, 16.8, page 451

Choosing between multiple precursors

In ordinary cases, as illustrated by the examples, a redefined routine has only one effective precursor. In studying the join of routines, however, we have seen that it is possible for a routine declaration to be the redefinition of two or more parent versions (precursors). If you use a **Precursor** construct in such a case you will need to specify which precursor you want, by listing its name. Instead of just *Precursor (arguments)* the syntax in that case will be *Precursor {PARENT} (arguments)*, where *PARENT* is the name of one of the parent classes from which we are redefining the feature.

The earlier join example illustrates the case of multiple precursors:

← Page 296.



```
class D inherit
  B
  redefine f end
  C
  redefine f end
feature
  f
  do
    ... “Redefined algorithm” ...
  end
  ...
end
```

In the “Redefined algorithm” a precursor call of the form *Precursor (arguments)* is invalid, because it leaves open the obvious question “Which precursor do you mean: the version from *A*, or from *B*?”.

The qualified form removes the ambiguity: you should write either one of

```
Precursor {B} (arguments)
Precursor {C} (arguments)
```

You may, in fact, include both of these in the redefinition's body if you need to reuse both parents' original implementations to define the new one.



The form with explicit qualification, *Precursor* {*PARENT_NAME*}, is valid even in the absence of ambiguity. It is usually preferable to use this form in all cases since it clarifies the context and helps identify errors if you change parents. This is part of the style guidelines.



With these observations we have enough to introduce the formal properties of the *Precursor* construct. (They will mark the beginning of the formal part of this chapter; since it will introduce no new construct or technique, but only provide precise definitions of the concepts seen informally so far, you may on first reading skip to the next chapter.)

Precursor specification

The syntax of the *Precursor* construct covers the variants seen in the preceding examples:



```
Precursor ≙ Precursor [Parent_qualification] [Actuals]
Parent_qualification ≙ "{" Class_name "}"
```

← The definition of *Parent_qualification*, repeated here for clarity, originally appeared with *Clients* on page 208.

For the validity and semantics, we avoid introducing special rules — which would repeat many of the properties of calls — by relying on our usual *unfolding* language definition technique: we just pretend that we were clever enough, in the parent class, to keep a duplicate of the original feature, by relying on a *synonym* feature:

Relative unfolded form of a Precursor

In a class *C*, consider a *Precursor* specimen *p* appearing in the redefinition of a routine *r* inherited from a parent class *B*. Its **unfolded form relative to *B*** is an *Unqualified_call* of the form *r'* if *p* has no *Actuals*, or *r' (args)* if *p* has actual arguments *args*, where *r'* is a fictitious feature name added, with a **frozen** mark, as synonym for *r* in *B*.

In other words, we will talk about the **Precursor** call as if the declaration of r in B , instead of just

```
r (a: T; ...) ... do ... Body ...end
```

had been written with a frozen synonym

```
r, frozen r' (a: T; ...) ... do ... Body ... end
```

The **rule** on multiple declarations implies that this is equivalent to having declared independent features with an identical **Body**. Because r' is frozen, it retains the original semantics of r , in the context of the new class C ; this is exactly what we want to describe the validity and semantics of **Precursor**.

← “*Unfolded form of a possibly multiple declaration*”, page 159.

Here indeed is the validity:



Precursor rule

VDPR

A **Precursor** is valid if and only if it satisfies the following conditions:

- 1 • It appears in the **Feature_body** of a **Feature_declaration** of a feature f .
- 2 • If the **Parent_qualification** part is present, its **Class_name** is the name of a parent class P of C .
- 3 • Among the features of C 's parents, limited to features of P if condition 2 applies, exactly one is an effective feature redefined by C into f . (The class to which this feature belongs is called the **applicable parent** of the **Precursor**.)
- 4 • The unfolded form relative to the applicable parent is, as an Unqualified_call, argument-valid.

In addition:

- 5 • It is valid as an **Instruction** if and only if f is a command, and as an **Expression** if and only if f is a query.



This constraint also serves, in condition [3](#), as a definition of the “applicable parent”: the parent from which we reuse the implementation. Condition [4](#) relies on this notion.

Condition [1](#) states that the **Precursor** construct is only valid in a routine redefinition. In general the language definition treats functions and attributes equally (*Uniform Access* principle), but here an attribute would not be permissible, even with an **Attribute** body.

Because of our interpretation of a multiple declaration as a set of separate declarations, this means that if **Precursor** appears in the body of a multiple declaration it applies separately to every feature being redeclared. This is an unlikely case, and this rule makes it unlikely to be valid.

← “*SYNONYMS AND MULTIPLE DECLARATION*”, *5.18, page 159*.

Condition [2](#) states that if you include a class name, as in *Precursor {B}*, then *B* must be the name of one of the parents of the current class. The following condition makes this qualified form compulsory in case of potential ambiguity, but even in the absence of ambiguity you may use it to state the parent explicitly if you think this improves readability.

Condition [3](#) specifies when this explicit parent qualification is required. This is whenever an ambiguity could arise because the redefinition applies to more than one effective parent version. The phrasing takes care of all the cases in which this could happen, for example as a result of a join.

Here is a more verbose form of clause [3](#), obtained from a mathematical specification. Let *PAR* be the set of classes defined as follows: if the **Parent_qualification** part is present, *PAR* is the single-element set containing the class whose name is listed in that **Parent_qualification**; otherwise *PAR* is the set of all parents of *C*. Let *REDEF* be the set of all the effective routines, from classes belonging to *PAR*, of which *r* is a redefinition. Then *REDEF* has exactly one element.

Condition [4](#) simply expresses that we understand the **Precursor** specimen as a call to a frozen version of the original routine; we must make sure that such a call would be valid, more precisely “argument-valid”, the requirement applicable to such an **Unqualified_call**.

A **Precursor** will be used as either an **Instruction** or an **Expression**, in the same way as a call to (respectively) a procedure or a function; indeed **Precursor** appears as one of the syntax variants for both of these constructs. *Pages 228 and 761.* So in addition to being valid on its own, it must be valid in the appropriate role. Condition [5](#) takes care of this.



This property really belongs to the validity of instructions and expressions, but having a single clause here saves two full-fledged validity rules in the respective chapters: “It is valid to use a **Precursor** as an **Instruction** if and only if its unfolded form is a call to a procedure”, and “It is valid to use a **Precursor** as an **Expression** if and only if its unfolded form is a call to a function”.

The definition of the “relative” unfolded form didn’t necessarily yield a valid call; in fact it serves, in clause 4, to determine validity. If as a result we know we have a valid **Precursor**, we can define an unfolded form that is not relative any more:

Unfolded form of a Precursor

The **unfolded form** (absolute) of a valid **Precursor** is its unfolded form relative to its applicable parent.

The semantics follows immediately:



Precursor semantics

The effect of a **Precursor** is the effect of its unfolded form.

As usual, semantics is only defined for valid specimens, so it may legitimately use the “absolute” unfolded form.

10.25 REDEFINITION AND UNDEFINITION RULES



The agenda for the remainder of this chapter is to provide the precise rules for syntax, validity and semantics of the mechanisms seen so far — all feature adaptation mechanisms except for those involving repeated inheritance. As already noted, this will introduce no new techniques, so you may prefer on first reading to skip the rest of this chapter.

Let us begin with the straightforward syntax and validity of **Undefine** and **Redefine** subclasses. It will do no harm to repeat here (again) the general structure of **Inheritance** clauses:



Inheritance parts

```
Inheritance  $\triangleq$  "inherit Parent_list
Parent_list  $\triangleq$  "{Parent ";" ... }
Parent  $\triangleq$  "Class_type [Feature_adaptation]
Feature_adaptation  $\triangleq$  [Rename]
                    [New_exports]
                    [Undefine]
                    [Redefine]
                    end
```

This syntax appeared first on page 171.

← See page 183 for Rename and 209 for New_exports.

The clauses involved in the present discussion are **Undefine** and **Redefine**.

Here is the syntax of **Redefine**:



Redefinition

Redefine \triangleq **redefine** Feature_list

The following constraint applies to **Redefine** subclauses:



Redefine Subclause rule *VDRS*

A **Redefine** subclause appearing in a **Parent** part for a class *B* in a class *C* is valid if and only if every **Feature_name** *fname* that it lists (in its **Feature_list**) satisfies the following conditions:

- 1 • *fname* is the final name of a feature *f* of *B*.
- 2 • *f* was not frozen in *B*, and was not a constant attribute.
- 3 • *fname* appears only once in the **Feature_list**.
- 4 • The **Features** part of *C* contains one **Feature_declaration** that is a redeclaration but not an effecting of *f*.
- 5 • If that redeclaration specifies a deferred feature, *C* inherits *f* as deferred.

In this definition:

- The final name of an inherited feature (clause 1) is its name as it results from possible renaming (the **Feature_name** part only, not including any **Alias**). ← “*FEATURES AND THEIR NAMES*”, 6.10, page 185.
- A feature is “frozen” (clause 2) if it has been declared with the keyword **frozen** in its class of origin. The purpose of such a declaration is precisely to forbid any redefinition of the feature in descendants, guaranteeing that the exact original implementation remains in place. ← “*FEATURE DECLARATIONS: SYN-TAX*”, 5.10, page 140.
- A feature is a constant attribute (clause 2) if it is declared with a clause of the form **is v**, where *v* is **Manifest_constant**. ← “*HOW TO RECOGNIZE FEATURES*”, 5.12, page 145.
- The condition for a redeclaration to be valid (clause 4) appears later in this chapter; in particular, the new signature must conform to the original’s, and you may not redeclare an attribute into a function. → “*REDECLARATION RULES*”, 10.28, page 312.
- If *C* provides an effective version of a feature that it inherits as deferred, this is a case of effecting, and hence of redeclaration, but not of redefinition; as a consequence, clause 4 indicates that the feature must not appear in the **Redefine** subclause. → *Effecting is defined precisely in the next section.*

As to the semantics:



Redefinition semantics

The effect in a class C of redefining a feature f in a Parent part for A is that the version of f in C is, rather than its version in A , the feature described by the applicable declaration in C .

This new version will serve for any use of the feature in the class, its clients, its proper descendants (barring further redeclarations), and even ancestors and their clients under dynamic binding.

The syntax of an **Undefine** clause is similar to that of a **Redefine**:



Undefine clauses

Undefine \triangleq **undefine** Feature_list

The constraint is also similar:



Undefine Subclause rule

VDUS

An **Undefine** subclause appearing in a Parent part for a class B in a class C is valid if and only if every **Feature_name** $fname$ that it lists (in its **Feature_list**) satisfies the following conditions:

- 1 • $fname$ is the final name of a feature f of B .
- 2 • f was not frozen in B , and was not an attribute.
- 3 • f was effective in B .
- 4 • $fname$ appears only once in the **Feature_list**.
- 5 • Any redeclaration of f in C specifies a deferred feature.

--- EXPLAIN LAST CLAUSE ---

and the semantics:



Undefinition semantics

The effect in a class C of undefining a feature f in a Parent part for A is to cause C to inherit from A , rather than the version of f in A , a deferred form of that version.

→ This also applies to clients of proper ancestors, under dynamic binding. *"DYNAMICBINDING"*, 23.12, page 638

10.26 DEFERRED AND EFFECTIVE FEATURES AND CLASSES

The discussion has already referred informally to features being “deferred” or “effective” in a class. We can now make these notions precise, and use the opportunity to define what it means to “effect” a feature

DEFINITION

Effective, deferred feature

A feature f of a class C is an **effective feature** of C if and only if it satisfies either of the following conditions:

- 1 • C contains a declaration for f whose `Feature_body` is not of the `Deferred` form.
- 2 • f is an inherited feature, coming from a parent B of C where it is (recursively) effective, and C does not undefine it.

f is **deferred** if and only if it is not effective.

As a result of this definition, a feature is deferred in C not only if it is introduced or redefined in C as deferred, but also if its precursor was deferred and C does not redeclare it effectively. In the latter case, the feature is “inherited as deferred”.

← “Inherited as effective, inherited as deferred”, page 291.

The definition captures the semantics of deferred features and of their effecting. In case 1 it’s clear that the feature is effective, since C itself declares it as either an attribute of a non-deferred routine. In case 2 the feature is inherited; it was already effective in the parent, and C doesn’t change that status.

← “UNDEFINING A FEATURE”, 10.19, page 290.

In case 1 the declaration may be for a new (*immediate*) feature, or it may be a redeclaration of an inherited feature, deferred in the parent but made effective in C . This is known as an *effecting*:

DEFINITION

Effecting

A redeclaration into an effective feature of a feature inherited as deferred is said to **effect** that feature.

Some validity constraints, seen below, apply to this case: the effective feature must satisfy the Redeclaration rule, and if there are two or more deferred features among the lot, this is a *join*, governed by the Join rule.

It is possible under this definition for a redeclaration to effecting *several* inherited features. The only other case in which we permit inheriting several features with the same name without renaming is sharing under repeated inheritance. Here too we don’t have a real name clash, as long as at most one of the features is effective and they satisfy the two applicable rules (Redeclaration and Join).

→ “Repeated Inheritance Consistency constraint”, page 466.

Effecting may follow one three schemes:

- 1 • You may write *C* as heir to a class *B* where *f* is deferred, and provide an effecting of *f* in the form of a **Feature_declaration** in the **Features** part of *C*. This is the most common use of deferred features and effecting. ← As illustrated in “[EFFECTING A DEFERRED FEATURE](#)”, 10.14, page 276.
- 2 • You may want to inherit a specification from one parent *A* and the corresponding implementation from another *B*. In this case *A* will provide a deferred feature and *B* an effective feature with compatible signature; if they have the same final name in *C*, the *B* version will serve as effecting of the *A* version. In this case there is no new feature declaration in *C*. ← As illustrated by the figure “[Merging and overriding](#)”, page 295.
- 3 • *C* may also undefine a parent’s effective feature, and use an effective feature (inherited from a parent, or introduced or redefined in *C* itself) to provide an implementation. This is less common, but provides the mechanism for merging effective features, with one of the implementations overriding the others, as in one of the earlier examples. ← [Class D](#), page 295.

The above defines the meaning of “deferred” and “effective” for features. These qualifiers carry over to the classes that contain these features:

Deferred class property

A class that has at least one deferred feature must have a **Class_header** starting with the keyword **deferred**. The class is then said to be **deferred**.

This includes a validity requirement and a definition, both of which follow from the the original discussion of classes:

- The requirement to declare the class as **deferred** as soon as it has deferred feature is not a new validity constraint, but just repeats what the Class Header rule said — except that now, as a result of the definitions in this chapter, we have a precise definition of “deferred feature” (introduced as deferred, or inherited as deferred and not effected). ← “[Class Header rule](#)”, page 126.
- As to the definition, it follows from the Class Header rule combined with the original definition of “deferred class”, which stated that a class is deferred if its **Class_header** starts with **deferred**. That was a purely syntactic criterion; now we have a more meaningful one, reminding us that a class is deferred whenever it has a deferred feature. ← “[Expanded, frozen, deferred, effective class](#)”, page 127.

The reverse — that a class is effective if all its features are effective — is usually true, but not always since you have the option of declaring it explicitly as **deferred**, to specify that it remains abstract and not directly instantiatable. Hence the precise phrasing of the complementary property:

Bla bla bla =====

Effective class property

A class whose features, if any, are all effective, is effective unless its `Class_header` starts with the keyword **deferred**.



It is not an error to declare a class **deferred** if it has no deferred features; the effect is simply that clients are not able to create direct instances. It is indeed sometimes useful to introduce a class that cannot be directly instantiated; for example the designer may intend the class to be used only through inheritance. The technique to achieve this is simply to state the abstract nature of the class by declaring it **deferred** even if all its features are effective.



As a summary, remember that you **must** declare a class as

```
deferred class C ...
```

as soon as it has a deferred feature f — not only if f is introduced in C as deferred, but also if C inherits it as deferred and does not effect it.

For an effective class, you will just use one of

```
class C...
expanded class C...
reference class C...
```

This is not necessarily the beginning of the class text itself since there may be a [Notes](#) clause first.

10.27 ORIGIN AND SEED

Two useful definitions follow from the discussion of redeclaration. [Chapter 6](#) defined the **origin** of a feature introduced in class C as C itself.

We can now generalize this to arbitrary features, inherited as well as immediate. The associated notion is a feature's **seed**, its original version. These notions, which will be especially useful in the discussion of repeated inheritance, are defined as follows.

← This is a refinement of the initial definition of "origin" on page [133](#), which only covered case [1](#) of the present definition.



Origin, seed

Every feature f of a class C has one or more features known as its **seeds** and one or more classes known as its **origins**, as follows:

- 1 • If f is immediate in C : f itself as seed; C as a origin.
- 2 • If f is inherited: (recursively) all the seeds and origins of its precursors.

→ "[SHARING AND REPLICATION](#)", [16.4](#), page [436](#).

The origin, a class, is “where the feature comes from”, and the seed is the version of the feature from that origin. In the vast majority of cases this is all there is to know. With repeated inheritance and “join”, a feature may result from the merging of two or more features, and hence may have more than one seed and more than one origin. That’s what case 2 is about.



-----If this is your first reading, do not let yourself be troubled by case 2, which refers to repeated inheritance. As soon as you have read the first three sections of the repeated inheritance chapter, the context in which case 2 occurs should be quite clear.

→ Chapter 16.

The origin of a feature is the most remote ancestor from which the feature comes, and its seed is its original form in that ancestor.

None of the reincarnations that the feature may have gone through along the inheritance part as a result of redefinition, effecting or renaming may affect its seed and its origin.

10.28 REDECLARATION RULES



(The rest of this chapter gives the formal rules applying to feature redeclaration. The essential concepts have already been seen, so you may safely skip to the next chapter on first reading.)

---- REMOVE ALL THIS!!! According to the earlier definitions, case ---- ← “*Effective, deferred feature*”, page 309 and “*Effecting*”, page 309.
 -- is an effecting. Case ----- is an effecting for deferred *f* and effective *g*, a redefinition if they are both deferred or both effective. Clause 5 of the constraint below will preclude the other apparent possibility: *f* effective, *g* deferred.

In case -----, the text of *C* does not contain any declaration for *f*, but some other inherited feature *g* (which must come from a different parent) effects *f*. It is convenient to treat this implicit and somewhat special case as a redeclaration, along with the explicit and more common case -----.

The above definition says nothing about validity: case ---- simply states that if a declaration uses the name of an inherited feature, we must treat it as a redeclaration (valid or not) of that feature, not as the declaration of a new, or *immediate*, feature. Here is the rule that determines when a redeclaration (explicit or implicit) is valid:



Redeclaration rule

VDRD

Let C be a class and g a feature of C . It is valid for g to be a redeclaration of a feature f inherited from a parent B of C if and only if the following conditions are satisfied.

- 1 • No effective feature of C other than f and g has the same final name.
- 2 • The signature of g conforms to the signature of f .
- 3 • The Precondition of g , if any, begins with **require else** (not just **require**), and its Postcondition, if any, begins with **ensure then** (not just **ensure**).
- 4 • If the redeclaration is a redefinition (rather than an effecting) the Redefine subclause of the Parent part for B lists in its Feature_list the final name of f in B .
- 5 • If f is inherited as effective, then g is also effective.
- 6 • If f is an attribute, g is an attribute, f and g are both variable, and their types are either both expanded or both non-expanded.
- 7 • f and g have either both no alias or the same alias.
- 8 • If both features are queries with associated assigner commands fp and gp , then gp is the version of fp in C .



Condition [1](#) prohibits name clashes between effective features. For g to be a redeclaration of f , both features must have the same final name; but no other feature of the class may share that name. This is the fundamental rule of **no overloading**.

No invalidity results, however, if f is deferred. Then if g is also deferred, the redeclaration is simply a redefinition of a deferred feature by another (to change the signature or specification). If g is effective, the redeclaration is an effecting of f . If g plays this role for more than one inherited f , it both joins and effects these features: this is the case in which C kills several deferred birds with one effective stone.

← *The bird-shooting was on page [294](#).*

Condition [2](#) is the fundamental type compatibility rule: signature conformance. In the case of a join, g may be the redeclaration of more than one f ; then g 's signature must conform to all of the precursors' signatures.

→ *See details below: "[RULES ON JOINING FEATURES](#)", [10.29](#), [page 315](#).*

Signature conformance permits *covariant* redefinition of both query results and routine arguments, but for arguments you must make the new type detachable — $?U$ rather than just U — to prevent “catcalls”.

Condition [3](#) requires adapting the assertions of a redeclared feature, as governed by rules given [earlier](#).

← *“[REDECLARATION AND ASSERTIONS](#)”, [10.17](#), [page 283](#).*

Condition [4](#) requires listing f in the appropriate **Redefine** subclause, but only for a redefinition, not for an effecting. (We have a redefinition only if g and the inherited form of f are both deferred or both effective.) If two or more features inherited as deferred are joined and then redefined together, **every one of them** must appear in the **Redefine** subclause for the corresponding parent.

← *“[Redefine, redefinition](#)”, [page 292](#).*

Condition [5](#) bars the use of redeclaration for turning an effective feature into a deferred one. This is because a specific mechanism is available for that purpose: undefinition. It is possible to apply both undefinition and redefinition to the same feature to make it deferred and at the same time change its signature.

← *As noted: see class [E](#), [page 291](#).*

Condition [6](#) prohibits redeclaring a constant attribute, or redeclaring a variable attribute into a function or constant attribute. It also precludes redeclaring a (variable) attribute of an expanded type into one of reference type or conversely. You may, however, redeclare a function into an attribute — variable or constant.

Condition [7](#) requires the features, if they have aliases, to have the same ones. If you want to introduce an alias for an inherited feature, change an inherited alias, or remove it, redeclaration is not the appropriate technique: you must rename the feature. Of course you can still redeclare it as well.

Condition [8](#) applies to assigner commands. It is valid for a redeclaration to include an assigner command if the precursor did not include one, or conversely; but if both versions of the query have assigner commands, they must, for obvious reasons of consistency, be the same procedure in C .



In earlier versions of the language, there was an extra condition, prohibiting a redeclaration from changing an **External** feature into an **Internal** one or conversely. Although initially justified by the original conventions on external features, this had become just an implementation constraint with no remaining conceptual justification.

→ On **External routines**, see chapter [31](#), especially “[BASICS OF EXTERNAL ROUTINES](#)”, [31.5](#), page [828](#).



Note, however, that redefining an external routine into a non-external one will usually cause a small performance penalty for the *original* (non-redefined) version, as the Eiffel compiler will probably have to call the external routine through an Eiffel wrapper.

10.29 RULES ON JOINING FEATURES

The last constraint that we need to examine governs the validity and semantics of the join mechanism, used to merge two or more features, of which at most one is effective, by inheriting them under the same name.

It is useful first to extend the notion of precursor:



Precursor (joined features)

A **precursor** of an inherited feature is a version of the feature in the parent from which it is inherited.

Bla bla bla

← The definition for in the non-join case was on page [268](#). A final, more formal definition covering both cases will appear on page [473](#) at the end of the repeated inheritance chapter.

precursors.

DEFINITION

Transposition to a class or type

The **transposition** to a class C of a specimen s appearing in a ancestor A of C is the specimen obtained from s by replacing every expression by its Equivalent Dot Form, then:

- 1 • Replacing the arguments of any Call by (recursively) their transposition to C .
- 2 • If s is part of the declaration of a feature g replicated in C along a certain repeated inheritance path, replacing any Feature_name used as name of the feature of an unqualified call or as anchor of an anchored type by the name resulting from any renaming of the feature along that path.
- 3 • Replacing any Feature_name used as name of the feature of an unqualified call or as anchor of an anchored type, if case 2 does not apply, by the result of any renaming along applicable inheritance paths.
- 4 • In every qualified call of target t , replacing t by (recursively) its transposition t' to C and the feature of the call by (recursively) its transposition to the type of t' in C .
- 5 • In every Non_object_call of target type T , replacing T by (recursively) its transposition T' to C and the feature of the call by (recursively) its transposition to T' .
- 6 • For every entity e , other than an attribute, such that s includes a declaration for e , replacing every occurrence of e by a fresh identifier not used in C .
- 7 • If an ancestor B of C has a parent type P of base class A , replacing every occurrence of any generic parameter G of A by (recursively) the transposition to C of the application to G of P 's generic substitution.

The transposition to a type T of a specimen s appearing in a ancestor of the base class C of T is the result of applying the generic substitution of T to the class transposition of s to C .

DEFINITION

Transposition

The **direct transposition** to a class B of a specimen s appearing in a parent class A of B is the specimen obtained from s by replacing every expression by its Equivalent Dot Form, then:

- 1 • Replacing the arguments of any Call by (recursively) their direct transposition to B .
- 2 • If s is part of the declaration of a feature g replicated in B along a certain repeated inheritance path, replacing the name of the feature of any unqualified call by the name of the feature as resulting from any renaming along that path.
- 3 • In every unqualified call of feature f whose feature name fn appears in a Rename_pair of the form fn as gn in a Parent part for A , such that case 2 does not apply, replacing fn by the identifier of gn .
- 4 • In every qualified call of target t , replacing t by (recursively) its class transposition t' to B and the feature of the call by (recursively) its transposition to the type of t' in B .
- 5 • In every Non_object_call of target type T , replacing T by (recursively) its class transposition T' to B and the feature of the call by (recursively) its transposition to T' .
- 6 • For every entity e , other than an attribute, such that s includes a declaration for e , replacing every occurrence of e by a fresh identifier not used in B .
- 7 • Replacing every occurrence of a formal generic parameter of A by the generic substitution of B 's parent type of base class A .

The **class transposition** to a class C of a specimen s appearing in an ancestor A of C is:

- 8 • If A and C are the same class: s .
- 9 • If A is a parent of an ancestor B of C : (recursively) the transposition to C of the direct transposition of s to B .

The **transposition** to a type T of a specimen s appearing in an ancestor of the base class C of T is the result of applying the generic substitution of T to the class transposition of s to C .

The first part (cases [3](#) and [7](#)) defines transposition to an heir (direct descendant). Cases [8](#) and [9](#) generalize this to any descendant. Recall that a descendant C of A is either A itself (case [8](#)) or, recursively, a descendant of an heir B of A (case [9](#)).

Case takes care of feature renaming. Because

Without the join mechanism there was just one precursor; but a feature resulting from the join of two or more deferred features will have all of them as precursors.

DEFINITION

Unfolded redeclaration

Consider a feature f of a class A . The **unfolded redeclaration** of f in an heir C of A is a **Feature_declaration** defined as follows:

- 1 • If C redeclares f , the declaration of f in C .
- 2 • Otherwise, a **Feature_declaration** for a feature with the same extended name, the same signature as f and the same Assigner_mark if any, both transposed to C , and an Attribute_or_routine consisting solely of:
 - If f is deferred, a **Feature_body** of the **Deferred** kind.
 - If f is an effective routine, a **do** clause whose **Compound** reads just **Precursor** (if f is a procedure) or **Result := Precursor** (if f is a function), followed by the parenthesized list of formal arguments if any.
 - If f is an attribute, an **attribute** clause whose **Compound** reads just **Result := Precursor**.

Here now is the validity constraint for joining features:



Join rule

VDJR

It is valid for a class C to inherit two different features under the same final name under and only under the following conditions:

- 1 • If both are inherited as effective, C redefines both into a common version.
- 2 • If both are inherited as deferred, the unfolded redeclaration in C of each of them is a valid redeclaration of the other.
- 3 • Otherwise, the unfolded redeclaration in C of the one inherited as effective is a valid redeclaration of the one inherited as deferred.

THE FOLLOWING INFORMATIVE TEXT NEEDS UPDATING.

The Join rule indicates that joined features must have exactly the same signature — argument and result types.

→ “[Repeated Inheritance rule](#)”, page 438;
 → “[Repeated Inheritance Consistency constraint](#)”, page 466.
 ← “[Redeclaration rule](#)”, page 313.



What matters is the signature after possible redefinition or effecting. So in practice you may join precursor features with different signatures: it suffices to redeclare them using a feature which (as required by [point 2 of the Redeclaration rule](#)) must have a signature conforming to all of the precursors’ signatures.

→ A signature conforms to another if every type in it conforms to the corresponding type in the other. See “[EXPRESSION AND SIGNATURE](#)” ← Join-cum-effecting was described on page 294.

If the redeclaration describes an effective feature, this is the case of both joining and effecting a set of inherited features. If the redeclaration describes a feature that is still deferred, it is a redefinition, used to adapt the signature and possibly the specification. In this case, [point 4](#) of the Redeclaration rule requires every one of the precursors to appear in the **Redefine** subclause for the corresponding parent.

Condition [1](#) mentions “redeclaration or effecting”. These two cases are not exclusive: an effecting — turning a feature f , inherited as deferred from a parent of C , into an effective one — can result from a new declaration of f in C , but also from a “join” of f with an effective feature inherited under the same name from another parent.

In any case, nothing requires the precursors’ signatures to conform to each other, as long as the signature of the version in C conforms to all of them. This means you may write a class inheriting two deferred features of the form

$f(p: P): T \dots$
 $f(t: Q): U \dots$

and redeclare them with

$$f(x: ? R): V \dots$$

provided R conforms to both P and Q and V to both T and U . No conformance is required between the types appearing in the precursors' signatures (P and Q , T and U).

The assumption that the features are “different” is important: they could in fact be the same feature, appearing in two parents of C that have inherited it from a common ancestor, without any intervening redeclaration. This would be a valid case of repeated inheritance; here the rule that determines validity is the Repeated Inheritance Consistency constraint. The semantic specification (sharing under the Repeated Inheritance rule) indicates that C will have just one version of the feature.

Conditions 1 and 2 of the Join rule are consistency requirements on aliases and on assigner commands. The condition on aliases is consistent with condition 7 of the Redeclaration rule, which requires a redeclaration to keep the alias if any; it was noted in the comment to that rule that redeclaration is not the appropriate way to add, change or remove an alias (you should use renaming for that purpose); neither is join. The condition on assigner commands ensures that any Assigner_call has the expected effect, even under dynamic binding on a target declared of a parent type.

The following figure illustrates a valid case, in which all types involved are non-generic classes (so that conformance is just inheritance). U is an heir of P , but for the second argument the relation is in the other direction: Q is an heir of V . Then a redeclaration into a feature of signature $[U, Q]$, $[R]$ will be valid.

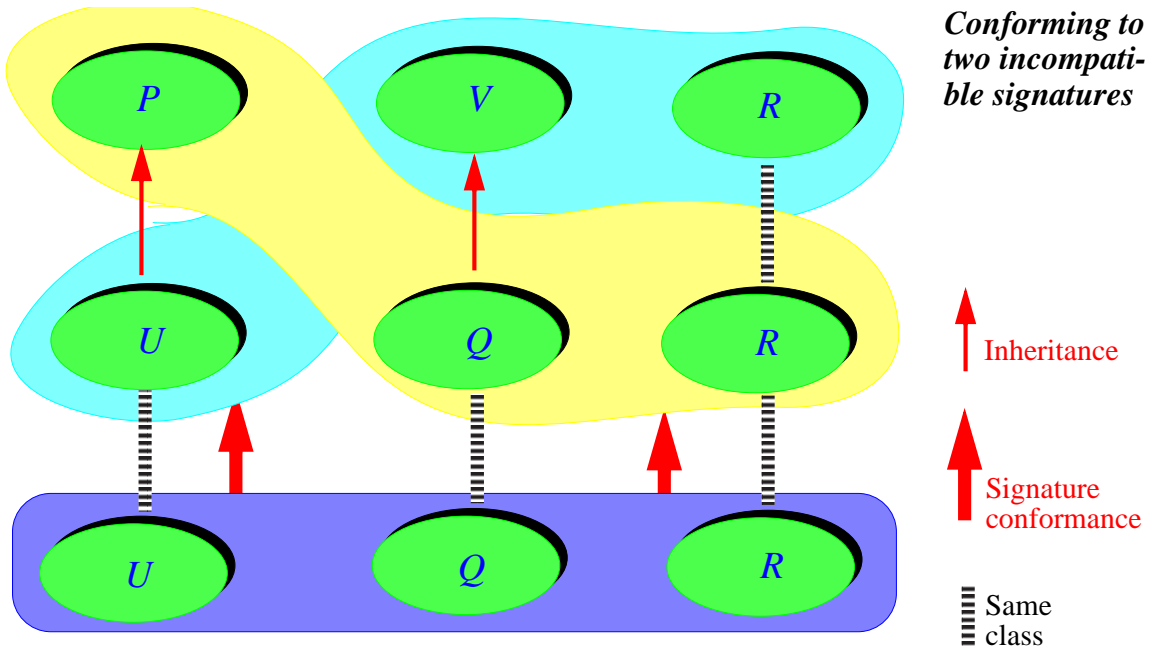
This takes care of the validity of the join mechanism. The last rule gives the precise properties of the resulting feature:



Join Semantics rule

Joining in a class C two or more inherited features with the same final name under the terms of the Join rule yields a single feature of C defined as follows:

- 1 • If at least one of these features is effective: its unfolded redeclaration in C .
- 2 • Otherwise: the unfolded redeclaration in C of any of them.



***** TO BE REDONE *****The rule covers three cases:

- An explicit redeclaration, which serves as a redeclaration of all the joined precursors, and gives them a new signature (which must conform to all their signatures per the Join rule), body (since it serves as “unfolded redeclaration” in point *****) and assertions (point *****).
- No redeclaration, with precursors all deferred, all having the same signature; they are then merged into a single deferred feature.
- No redeclaration, with one effective feature and the others deferred, all with the same signature; the effective feature then serves as effecting of the others.

In the absence of a redeclaration, point***** states that the new feature has no specific precondition and postcondition. It will still, however, have a **combined precondition** and a **combined postcondition** obtained from the precursors’ assertions. In the case of a redeclaration, the combined precondition and postcondition also include the assertions, if any, of the redeclared version.

Point ***** leaves the concatenation order unspecified.

In point ***** , there can be at most one effective precursor because of the Join rule.

In point ***** (corresponding to a rare case) language processing tools should produce an obsolescence message for the class performing the join, but the resulting feature is not itself obsolete.

← “*OBSOLETE FEATURES*”, 5.21, page 165.

Types

11.1 OVERVIEW

Types describe the form and properties of objects that can be created during the execution of a system. The type system lies at the heart of the object-oriented approach; the use of types to declare all entities leads to more clear software texts and permits compilers to detect many potential errors and inconsistencies before they can cause damage.

This chapter — complemented by the next two, which address generic types and tuple types — presents the type system.

11.2 THE ROLE OF TYPES

Every object is an instance of some type. (More precisely, it is a *direct instance* of exactly one type; thanks to the inheritance mechanism it may also be an instance of other, more general types.) Class texts may refer to eventual run-time objects through the software elements that denote values: constants, attributes, function calls, formal routine arguments, local variables, and expressions built from such elements.

Typing in Eiffel is static. For software developers, this means four practical properties:

- Every element denoting run-time values is *typed*: it has an associated type, limiting the possible types of the attached run-time objects.
- This type is immediately clear — to a human reader or to a language processing tool — from the element itself or the surrounding software text. For a manifest constant, such as the Integer [421](#), the type follows from the way the constant is written; in all other cases it is a consequence of a type declaration, made compulsory by the validity rules of the language.
- Non-atomic constructs impose complementary validity constraints, defining admissible type combinations. For example, an assignment requires the type of the source to conform to the type of the target.

- Since the constraints are defined as conditions on the software text, language processing tools such as compilers or static analyzers may check the type consistency of a system **statically**, that is to say, just by examining the system’s text, without making any attempt at execution.

This *explicit* and *static* approach to typing has a number of advantages. It makes software texts easier to read and understand, since developers, by declaring the types of entities, reveal how they intend to use them. It enables compilers and other tools to catch many potential errors by detecting inconsistencies between declarations and actual uses. It gives compilers information that helps them generate much more efficient code than would be possible with an untyped (or more weakly typed) language.

Typing in Eiffel is taken seriously. Many languages that claim to be statically (or even “*strongly*”) typed allow developers to cheat the type system, enticing them into sordid back-alley deals sometimes known as *casts*. No such cheating exists in Eiffel, where the typing rules suffer no exception. This is essential if we want to have any trust in our software. The only price to pay for this added security is the need to declare entities explicitly and to observe validity constraints — obligations which are even easier to justify if you observe that the type system, far from being a hindrance to the developer’s freedom of expression, helps in the production of powerful and readable software systems.

It should be noted, however, that some conceptual issues, having to do with *covariance* and *descendant hiding* can cause type problems in certain borderline cases. The [chapter on type checking](#) discusses them.

→ [Chapter 25](#).

The present chapter and the next two (on generic types and tuple types) explore the basic forms of types and their properties. This will not exhaust, however, the issue of typing, which pervades most of the discussions of this book. To understand the type system fully, you will need important complements provided by two separate chapters:

- The discussion of **conformance** will explain how a type may be used in lieu of another, and its instances in lieu of that other’s instances.
- The presentation of the **type checking** policy will show how the typing policy defines the fundamental validity constraints on the most important computational construct — feature call.

→ *Conformance is the topic of [chapter 14](#). [Chapter 23](#) covers calls; on type checking, see [chapter 25](#).*

11.3 WHERE TO USE TYPES

You will need to write a type — a specimen of the construct **Type** — in the following contexts:

- 1 • To declare the result type of an attribute or function: construct **Declaration_body**. ← *Syntax*: page [141](#).
- 2 • To declare the arguments of a routine or inline agent: construct **Formal_arguments**, defined in terms of **Entity_declaration_list**. ← *Page 220*; see also *Inline_agent*, p. [751](#).
- 3 • To declare a local routine entity: construct **Local_declarations** (also defined in terms of **Entity_declaration_list**). ← *Page 225*.
- 4 • To indicate that a class has a certain parent: construct **Parent**, as part of **Inheritance**. ← *Page 171*.
- 5 • To specify actual generic parameters, as explained in the next chapter: construct **Actual_generics**. → *Page 350*.
- 6 • To specify a generic **Constraint**, also in the next chapter: construct **Constraint**, part of **Formal_generics**. → *Page 357*.
- 7 • To indicate an explicit creation type in a creation instruction or expression: construct **Explicit_creation_type**. → *Page 551*.
- 8 • To choose from a set of instructions, based on an expressions's type, in a **Multi_branch**. → *Choice*, page [485](#).
- 9 • To specify the parameters (component types) of a **Tuple_type**. → *Page 372*.
- 10 • To declare the type of a target of a **Call_agent**. → *Agent_actual*, page [752](#).
- 11 • To specify target conversion for infix operators. → *Page 771*.
- 12 • To call a feature without a target, in a **Non_object_call**. → *Page 626*.

As an example of the first three cases, here is the beginning of a possible function declaration:



```
total_occupied_area (wl: LIST [WINDOW] ): RECTANGLE
-- Smallest rectangle that covers the representations
-- of all windows in wl
local
  xmin, ymin, xmax, ymax: REAL
... Rest of routine omitted ...
```

In this example and all the others, types are easy to recognize: apart from keywords such as **like**, they use all-upper-case names.

The function has a result (case [1](#)) of type **RECTANGLE**, probably a reference type, and one argument (case [2](#)) of type **LIST [WINDOW]**, a “generically derived” reference type. It uses four local variables (case [3](#)) of type **REAL**, a basic expanded type. The use of **WINDOW** as actual generic parameter to **LIST** provides an example of case [5](#). → *See chapter 12 about generically derived types.*

The following class beginning uses types in its two **Parent** parts (case 4):

```
class DISPLAY_STATE inherit
  LIST [WINDOW]
  INPUT_MODE
  ...
```

An example of case 6 is the use of type **ADDABLE** in a class text starting with

```
class MATRIX [G -> ADDABLE ] ...
```

which states that any actual generic parameter must conform to **ADDABLE** (which means roughly that it must be based on a descendant of that class).

An example of case 7 is the **Creation instruction**

```
create { WINDOW } a.set (x_corner, y_corner)
```

→ See chapter 20 about creation instructions and expressions.

which creates a direct instance of **WINDOW**, initializes it using a call to **set** with the given arguments, and attaches it to **a**. If **a** is of type **WINDOW** you may (and usually should) omit the **{ WINDOW }** part; but it is useful if **a**'s type is a proper ancestor of **WINDOW** and you expressly want to create **a** as a **WINDOW**. Another example of case 7 is the **Creation expression** in

```
screen.display (create { WINDOW }.set (x_corner, y_corner))
```

where we pass as argument to procedure **display** an object of type **WINDOW** created for the occasion. Here specifying the type is not an option but a necessity since, unlike the previous case, we don't have an entity **a** with a type declaration to serve as the default.

An example of case 8 is a multi-branch instruction

```
inspect
  last_exception.type
when { DEVELOPER_EXCEPTION } then
  fix_context ; retry
when { SIGNAL }, { NO_MORE_MEMORY } then
  cleanup
end
```

appearing in this case in a **Rescue** clause to process exceptions. This states what to do depending on the type of **last_exception**.

An example of case 9 (similar in syntax to case 5, actual generic parameters) is the tuple type

```
TUPLE [ REAL, INTEGER, RECTANGLE ]
```

which describes “tuples” — sequences of values— with at least three elements, the first of type *REAL* and so on.

An example of case [10](#) is

```
agent { RECTANGLE }.rotate (90)
```

an agent expression denoting a partially specified operation, ready to call *rotate* (assumed to be a procedure of *RECTANGLE*, with a single argument representing an angle) to rotate any rectangle by 90 degrees.

An example of case ---- ADAPT --- is an instruction

```
if {x: EXPECTED_TYPE } retrieved_from_network then
    x.f
    ...
else
    ...
end
```

which determines whether a run-time object obtained from an outside source is of a certain predicted type.

Case [11](#) covers the ability to convert the result of an arithmetic operator to the type of the second operand, as in the following in class *INTEGER*

```
plus alias "+" convert { COMPLEX } (other: REAL): REAL
... Definition of integer addition ...
```

which dispatches *your_real + your_complex* to the feature with the same name in class *COMPLEX* (rather than the one specified here).

Finally, case [12](#) allows calls of the form { *T* } *some_feature* where *some_feature* is a feature of type *T* that doesn't need a target, for example a constant attribute.

11.4 HOW TO DECLARE A TYPE

The basis of the type system is the notion of class: every type is, directly or indirectly, based on a class, which provides the principal information for determining how instances of the class will look like. But classes are only the starting point of a whole set of type mechanisms that afford you considerable flexibility:

- Certain classes, said to be **generic**, do not directly describe a type; instead, they describe a type pattern, with one or more variable parts that must be filled in, through a “generic derivation”, to yield an actual type. For example the class *LIST [G]* describes lists of elements of an arbitrary type, denoted in the class by *G*.

→ See “[BASE CLASS, BASE TYPE AND TYPE SEMANTICS](#)”, [11.7, page 332](#) below.

- Within the text of a generic class such as *LIST*, the **formal generic parameters** such as *G* themselves represent types (the possible actual generic parameters). The class may for example introduce an attribute of type *G*, or a routine with an argument or result of type *G*. Syntactically, then, a formal generic parameter is a type, although the exact nature of that type is not known in the class itself; only when a generic derivation provides the corresponding *actual generic parameter* (such as *WINDOW* above) can we know what *G* represents in that case.
- Finally, you may declare an entity *x* in a class *C* by using an **anchored** type of the form **like anchor** for some other entity *anchor*. This mechanism avoids tedious redeclarations since it ties the fate of *x*'s type to that of *anchor*: in *C*, *x* is treated as if you had declared it with the type used for the declaration of *anchor*; if a proper descendant of *C* redeclares *anchor* with a new type, *x*'s type will automatically follow.

Here is the syntactical specification covering all the possibilities.



		Types
Type	△	Class_or_tuple_type Formal_generic_name Anchored
Class_or_tuple_type	△	Class_type Tuple_type
Class_type	△	[Attachment_mark] Class_name [Actual_generics]
Attachment_mark	△	"?" "!"
Anchored	△	[Attachment_mark] like Anchor
Anchor	△	Feature_name Current

→ *Tuple_type* is defined in the chapter on tuples, page 372.

→ *Actual_generics* describes a list of types. The specification is on page 350 as part of the discussion of genericity in the next chapter.

The most common and versatile kind is *Class_type*, covering types described by a class name, followed by actual generic parameters if the class is generic. The class name gives the type's base class. If the base class is expanded, the *Class_type* itself is an expanded type; if the base class is non-expanded, the *Class_type* is a reference type.

A class is an "expanded class" if its *Class_* header begins with **expanded class**, and a non-expanded class otherwise.

Class_or_tuple_type covers tuple types as well as class types. Tuple types, studied in their own chapter, are a kind of trimmed-down class type; *TUPLE [a: X; b: Y; c: Z]* acts like a class with three features *a*, *b* and *c* of the types given. You can omit the labels: *TUPLE [X, Y, ...]* describes finite sequences of values of which the first must be of type *X*, the second of type *Y* and the third of type *Z*. Tuple types share a number of properties with class types, hence the first variant of *Type*, covering them both.

→ Chapter 13 discusses tuples.

An **Attachment_mark** ? indicates that the type is **detachable**: its values may be void — not attached to an object. The **!** mark indicates the reverse: the type is **attached**, meaning that its values will always denote an object; language rules, in particular constraints on attachment, guarantee this. No **Attachment_mark** means the same as **!**, to ensure that a type, by default, will be attached.

The second syntactical variant, **Formal_generic_name**, covers the formal generic parameters of a class. If *C* has been declared as

```
... class C [...,G,...] ...
```

then, within the text of *C*, *G* denotes a type. As noted, you cannot know the precise nature of this type just by looking at class *C*; *G* represents whatever actual generic parameter is provided in a particular generic derivation.

The next category, **Anchored** types of the form **like anchor**, accounts for anchored declarations.

Tuple_type, the last category, covers types of the form *T*

→ Chapter 13 discusses tuples.

The rest of this chapter examines these type categories, except for the generic and tuple mechanisms which have their own chapters.

11.5 INSTANCES AND VALUES

For each kind of type in the language, we must specify — along with associated syntax rules and validity constraints — the *semantics* of the type.

Defining the semantics of a type *T* involves answering two questions:

SEMANTICS

- What objects can be produced, during execution, from the description given by *T*?
- What are at run time the possible values of an entity or expression of type *T*?

The answers have precise names:

DEFINITIONS

Direct instances and values of a type

The **direct instances** of a type *T* are the run-time objects resulting from: representing a manifest constant, manifest tuple, **Manifest_type**, agent or **Address** expression of type *T*; applying a creation operation to a target of type *T*; (recursively) cloning an existing direct instance of *T*.

The **values** of a type *T* are the possible run-time values of an entity or expression of type *T*.

→ See also “Type, generating type of an object, generator”, page 506.

Specifying the direct instances might seem sufficient; the reason we also need to consider values is the difference between **expanded** and **reference** types. A type's values are objects in the first case, references to objects in the second.

Expanded types include as a special sub-category the basic types: *BOOLEAN*; *CHARACTER* and its sized variants such as *CHARACTER_8*; *INTEGER* and its sized variants such as *INTEGER_8* and *NATURAL_64*; *REAL* and its sized variants. *REAL_32* and *REAL_64*; and *POINTER*, covering addresses of features to be passed to external (non-Eiffel) routines. Clearly, an entity of integer type should give us an integer value, not a reference to a dynamically allocated cell that contains an integer.

→ "[PASSING THE ADDRESS OF AN EIFFEL FEATURE](#)".
[31.8, page 833](#).



Reference provide more flexibility thanks to dynamic object allocation, allowing the execution to create objects when and only when it needs them; reference semantics, supporting linked data structures. Expanded types, for their part, are useful not only for basic types but also for describing *sub-objects* avoiding indirections. The role of expanded types, and the criteria for choosing between expanded and reference, are further studied below.

The notion of type has, besides the expanded-reference distinction, a number of variants detailed in the following sections:

- You may define a type by **anchoring**, as *like something*, tying it to the type of an entity, so that it will follow any redefinitions in descendants. Anchoring is covered later in this chapter.
- A type may also be a **Formal_generic_name** representing a formal generic parameter of the enclosing class; it then serves as a placeholder for any type (reference or expanded) that is used in a generic derivation. The whole generic mechanism will be discussed in the next chapter.

In understanding type semantics, another useful notion is that of *instance*, complementing the notion of *direct* instance defined above:



Instance of a type

The **instances** of a type *TX* are the direct instances of any type conforming to *TX*.

Since every type conforms to itself, this is equivalent to stating that the instances of *TX* are the direct instances of *TX* and, recursively, the instances of any other type conforming to *TX*.



In the well-known example of an inheritance hierarchy with a class *FIGURE* at the top and descendants describing successively more specific geometrical figures, such as *CLOSED_FIGURE*, *POLYGON*, *RECTANGLE*, *SQUARE*, each inheriting from the preceding one, a direct instance of *SQUARE* is also an instance of all the others, including *SQUARE* itself.

This also illustrates that a deferred type such as *FIGURE*, which cannot have direct instances (since creation instructions of target *FIGURE* are invalid), may have instances if the class has effective descendants.

→ “[Creation and deferred classes](#)”, page 537.

A semantic rule connects the notion of value and instance:

Instance principle

Any value of a type *T* is:

- If *T* is reference, either a reference to an instance of *T* or (unless *T* is attached) a void reference.
- If *T* is expanded, an instance of *T*.

Thanks to this rule, it suffices, when studying type semantics, to define the *direct* instances of each possible type. The instances follow immediately and — since the type’s declaration indicates whether it is reference (and if so, attached) or expanded — so do the values.

11.6 INSTANCES OF A CLASS

Along with the instances, direct and indirect, of a *type*, it is convenient to talk about the corresponding notion for a *class*:



Instance, direct instance of a class

An instance of a class *C* is an instance of any type *T* based on *C*.
A direct instance of *C* is a direct instance of any type *T* based on *C*.

For non-generic classes the difference between *C* and *T* is irrelevant, but for a generic class you must remember that by itself the class does not fully determine the shape of its direct instances: you need a type, which requires providing a set of actual generic parameters.

11.7 BASE CLASS, BASE TYPE AND TYPE SEMANTICS

At its core, the notion of type in Eiffel proceeds from the notion of class. Indeed, we can bring down the properties of any type to those of an associated `Class_or_tuple_type` and, through it, to those of a class:



Base principle

Any type T proceeds, directly or indirectly, from a `Class_or_tuple_type` called its **base type**, and an underlying class called its **base class**.

The base class of a type is also the base class of its base type.

A `Class_type` is its own base type; an anchored type **like** *anchor* with *anchor* having base type U also has U as its base type. For a formal generic parameter G in `class C [G → T]` ... the base type is (in simple cases) the constraining type T , or *ANY* if the constraint is implicit.



The base class is the class providing the features applicable to instances of the type. If T is a `Class_type` the connection to a class is direct: T is either the name of a non-generic class, such as `PARAGRAPH`, or the name of a generic class followed by `Actual_generics`, such as `LIST [WINDOW]`. In both cases the base class of T is the class whose name is used to obtain T , with any `Actual_generics` removed: `PARAGRAPH` and `LIST` in the examples. For a `Tuple_type`, the base class is a fictitious class `TUPLE`, providing the features applicable to all tuples.

For types not immediately obtained from a class we obtain the base class by going through base type: for example T is an `Anchored` type of the form **like** *anchor*, and *anchor* is of type `LIST [WINDOW]`, then the base class of that type, `LIST`, is also the base class of T .

A general property applies to the base class and base type:



Base rule

The **base type** of any type is a `Class_or_tuple_type`, with no `Attachment_mark`.

The **base class** of any type other than a `Class_or_tuple_type` is (recursively) the base class of its base type.

The **direct instances** of a type are those of its base type.

A class text may refer to a class rather than a type in only three cases: the beginning of the class declaration, as in `class YOUR_CLASS_NAME ...`; a `Clients` part (syntax page 208); and a `Precursor` construct (syntax page 303).



Why are these notions important? Many of a type's key properties (such as the features applicable to the corresponding entities) are defined by its base class. Furthermore, class texts almost never directly refer to classes: they refer to *types* based on these classes.

For example, assuming that C is generic:

- If D is an heir of C , the **Inheritance** part of D will list as **Parent** not C , but a type of the form C [ACTUALI, ...].
- To describe objects to which C 's features are applicable, D will declare an entity e using not C but, again, a type generically derived from C .

In such situations (and all other uses of types listed earlier) the base class provides the essential information: what features are associated with C . In the first example, they give the list of features that D inherits from C ; in the second, they provide the features which D may call on e .

As for the base type, besides its role in defining the base class, it appears in many of the conformance rules, and determines what kind of object a creation operation will produce at run time.

→ Conformance: chapter 14; creation: chapter 20.

Clearly, you may only build a class type, generically derived or not, if the base class is a class of the universe:



Class Type rule

VTCT

A Class_type is valid if and only if it satisfies the following two conditions:

- 1 • Its Class_name is the name of a class in the surrounding universe.
- 2 • If it has a “?” Attachment_mark, that class is not expanded.

The class given by condition 1 will be the type's base class. Regarding condition 2, an expanded type is always attached, so an Attachment_mark would not make sense in that case.

The Base rule simplifies the presentation of type semantics. For every kind of type reviewed in this chapter and the next two we must specify the type's semantics, by stating what are the type's direct instances and its values. Thanks to the Base rule the process is straightforward:



Type Semantics rule

To define the semantics of a type T it suffices to specify:

- 1 • Whether T is expanded or reference.
- 2 • Whether T , if reference, is attached or detachable.
- 3 • What is T 's base type.
- 4 • If T is a Class_or_tuple_type, what are its base class and its type parameters if any.

→ For Formal_generic_name types the expanded/reference status depends on each generic derivation. See “SEMANTICS OF GENERIC TYPES”, 12.10, page 363.

As soon as we know T 's base type, and its actual generic parameters if any, → Chapter 19. we will know its **direct instances**: those of its base type, determined by the rules on type instances. If T is not a class type, we will know from the Base rule that its **base class** is the base class of T 's base type (itself a Class_or_tuple_type). Finally, the **values** of T will be its instances if it is an expanded type, otherwise references to such instances.

In application of the Type Semantics rule, every presentation of a new kind of type in this chapter and the next two has a SEMANTICS paragraph that simply defines the base type (item 3 above), the base class in the case of a Class_or_tuple_type (4), and whether it is expanded or reference (1).

To simplify the discussion, we allow ourselves to use “base class” and “base type” directly for expressions:



Base class and base type of an expression

Any expression e has a **base type** and a **base class**, defined as the base type and base class of the type of e .

11.8 CLASS TYPES WITHOUT GENERICITY

We start our exploration of the type categories with the simplest way of defining a type: using a class without generic parameters.

In this case there is no difference between class and type. Assume for example a class text of the form



```
class PARAGRAPH feature
  first_line_indent: INTEGER;
  other_lines_indent: INTEGER;
  set_first_line_indent (n: INTEGER)
    ... Procedure body omitted ...
  ... Other features omitted ...
end
```

Then a class of the same universe (including *PARAGRAPH* itself) may use *PARAGRAPH* as a type, for example to declare entities.

Here *PARAGRAPH* is declared as a non-expanded class, so the corresponding type is a reference type. At run-time, entities of that type represent references which, if not void, are attached to instances of *PARAGRAPH*, obtained through creation instructions.

If class *PARAGRAPH* had been declared a **expanded class** ..., then the resulting type would be expanded. In the general case:



Non-generic class type semantics

A non-generic class *C* used as a type (of the **Class_type** category) has the same expansion status as *C* (i.e. it is expanded if *C* is an expanded class, reference otherwise). It is its own base type (after removal of any **Attachment_mark**) and base class.

→ The generic version will be only slightly different: "[Generically derived class type semantics](#)", page 363.

These are not fascinating notions yet, but we must define a base class and base type for every type, and they will get less trivial as we move on.

PARAGRAPH, used as a type, is its own base type and its own base class. Values of type *PARAGRAPH* are references to instances of the class. Clients of the class may call exported features such as *first_line_indent* and others on entities of type *PARAGRAPH*.

→ See chapter 23 about calling features on entities.

Only one constraint, the Class Type rule, applies to a **Class_type** that is not generic: the **Identifier** must be the name of a class of the universe.

11.9 EXPANDED TYPES

Most of the types you define will probably be reference types similar to the last examples (*LIST*, *WINDOW*, *PARAGRAPH*...), as they offer the flexibility of creating objects on demand, and the ability to define linked structures. You can also use expanded types.

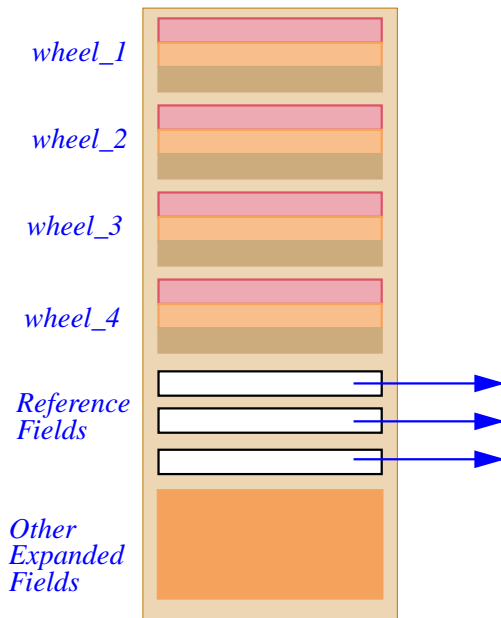
Role of expanded types

An earlier chapter previewed some of the possible reasons for using expanded types:

← "[EXPANDED CLIENTS](#)", 7.5, page 196.

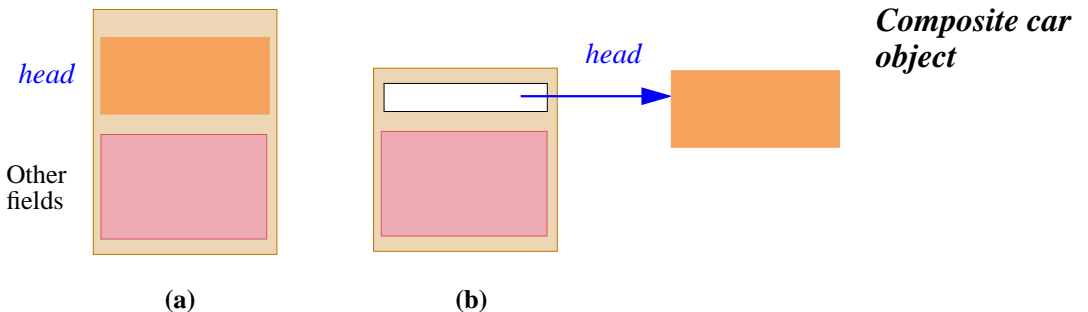
- Realism in modeling external world objects, especially when you want to describe objects that have sub-objects.
- Possible efficiency gain.
- Basic types.
- Interface with other languages.
- Machine-dependent operations.

The first case arises when we use Eiffel objects to model external world objects which are composite, rather than containing references to other objects. For example, in a Computer-Aided Design application, we may view a car as containing, among others, four "wheel" sub-objects, rather than four references to such objects. Such a decision, illustrated on the following figure, is only legitimate for objects which may never share sub-objects: in this example, a wheel may not be part of two different cars.



*Sub-object vs.
reference to
another object*

The second reason is, in some circumstances, a gain in efficiency: composite objects save space (by avoiding pointers) and time (by avoiding indirections). For example, if every instance of *PERSON* has a *head*, declaring *head* of an expanded type will give the structure illustrated by (a) on the next figure, avoiding the indirection of (b). Here again, this only applies because there is no sharing of sub-objects, at least if we exclude the case of Siamese twins.





You must realize, however, that the possible efficiency gain is not guaranteed. The last two figures, and similar illustrations of expanded attributes and composite objects, are only conceptual descriptions, not implementation diagrams. (Unlike other languages that shall remain nameless here, Eiffel is specified in terms of the abstract properties of software execution, not by prescribing a certain implementation.) The authors of an Eiffel compiler or interpreter may choose any representation they wish as long as they guarantee the *semantics* of expanded values, according to which (as explained in the discussion of reattachment in a [later chapter](#)) an assignment $x := y$ must copy the object attached to y onto the object attached to x , and an equality test $x = y$ must compare the objects field by field.

→ [Chapter 22](#).

Both the time and space gains are important in the case of basic types such as integers or characters; to manipulate the value `3`, we should not need to allocate an integer object dynamically, or to access it through a reference. For that reason, basic types are described by expanded classes of the Kernel Library, as explained in a [later section](#).

→ See [page 338](#), about basic types.

Another opportunity for expanded types may be the need to keep data structures produced and handled by software elements written in other languages. An example might be control information associated with a database management system, which Eiffel routines will not manipulate directly, but pass back and forth to foreign (non-Eiffel) routines. As you have no control over the format and size of such data structures, the best way may be simply to keep them as sub-objects within your Eiffel objects.

Defining expanded types

The class types seen so far may or may not be expanded:

- A **Class_type** whose base class is expanded is itself an expanded type; values of that type are objects (instances of the type).
- A **Class_type** whose base class is not expanded is a reference type; values are references to potential objects, created dynamically.

---- WHOLE DISCUSSION OF “EXPANDED T” REMOVED ----

Expanded types have specific properties, already previewed. First we must know precisely when a type is “expanded” and when it is “reference”:



Expanded type, reference type

A type T is **expanded** if and only if it is not a **Formal_generic_name** and the base class of its deanchored form is an expanded class.

T is a **reference type** if it is neither a **Formal_generic_name** nor expanded.



This definition characterizes every type as either reference or expanded, except for the case of a **Formal_generic_name**, which stands for any type to be used as actual generic parameter in a generic derivation: some derivations might use a reference type, others an expanded type.

→ *Genericity is discussed in the next chapter.*

Tuple types are, as a consequence of the definition, reference types.

Basic types



An important case of expanded types is a collection of **basic types** covering simple values:

→ *Detailed in chapter 30.*

- **BOOLEAN**, describing boolean values (true and false).
- **CHARACTER**, describing single characters.
- **INTEGER** and its variants supporting specific sizes: **INTEGER_8**, **INTEGER_16**, **INTEGER_64**. The sizes of values of type **INTEGER** must be settable through a compilation option (the recommended value is 64).
- **REAL**: floating-point numbers and its variants supporting specific sizes: **REAL_32**, **REAL_64**. The sizes of values of type **REAL** must be settable through a compilation option (the recommended value is 64).
- **POINTER**, serving to pass addresses of Eiffel features and expressions to non-Eiffel routines.

Three types also enjoy special properties but are not considered basic types: **ARRAY**, **STRING** and tuple types.

→ *See chapters 36 about **ARRAY** and **STRING** and 13 about tuples.*



Basic type

The basic types are **BOOLEAN**, **CHARACTER** and its sized variants, **INTEGER** and its sized variants, **REAL** and its sized variants and **POINTER**.

Like most other types, the basic types are defined by classes, found in the Kernel Library. In other words they are not predefined, “magic” types, but fit in the normal class-based type system of Eiffel.

Compilers typically know about them, so that they can generate code that performs arithmetic and relational operations as fast as in lower-level languages where basic types are built-in. This is only for efficient implementation: semantically, the basic types are just like other class types.

Properties of basic types, especially their conformance and semantics, appear in a chapter devoted to them. → *Chapter 30.*



The basic types need some special conformance properties. In general, a type U conforms to a type T only if U 's base class is a descendant of T 's base class. But then *INTEGER*, for example, is not a descendant of *REAL*. Since mathematical tradition suggests allowing the assignment $r := i$ for r of type *REAL* and i of type *INTEGER*, the definition of conformance will include a small number of special cases for basic types.

→ “EXPANDED TYPE CONFORMANCE”,
14.9, page 394.

Except for *POINTER* which has no exported feature of its own, each of the basic classes describes the operations applicable to values of the corresponding type (booleans, characters etc.). For compatibility with traditional arithmetic notation, many of the feature identifiers are **Unary** or **Binary**.

11.10 ANCHORED TYPES



The originality of an **Anchored** type, the last category in this chapter, is that it carries a provision for automatic redefinition in descendants of the class where it appears.

An **Anchored** type is of the form

like *anchor*

with the predictable definitions:



Anchor, anchored type, anchored entity

The **anchor** of an anchored type **like** *anchor* is the entity *anchor*. A declaration of an entity with such a type is an **anchored declaration**, and the entity itself is an **anchored entity**.

The anchor must be either an entity, or **Current**. If an entity, *anchor* must be the final name of a feature of the enclosing class.

Anchored types avoid “redefinition avalanche”. As long as what you only consider what happens in a class C , declaring an entity of type **like** *anchor* in C is the same as declaring it of the same type as *anchor*, say T . The difference comes from inheritance: if any descendant of C redefines the type of *anchor* to a new type (conforming to T), it will be considered to have also redefined all the entities anchored to *anchor*.

Since it is quite common to have a group of related entities that must keep the same type throughout their redefinitions, anchored declaration is essential to the smooth functioning of the type system. Without it we would constantly be writing lots of new declarations serving no other purpose than type specialization.

Anchored examples



We already encountered anchored declarations in the discussion of redeclaration; the example was that of a routine in the Data Structure Library class *LINKED_LIST*:

→ The encounter was towards the end of [10.9](#), page 269.

```
put_element (lc: like first_element; i: INTEGER)
```

whose argument *lc* represents a list cell. This declaration “anchors” *lc* to *first_element*, a feature of the class declared of type *LINKABLE [G]* (the type representing list cells). As a result, *lc* itself is considered in *LINKED_LIST* to have the same type as *first_element*, *LINKABLE [G]*. Because *lc* has been anchored to *first_element*, any descendant of *LINKED_LIST* which redefines *first_element* to a new type, taking into account more specific forms of list cells (such as cells chained both ways, or tree nodes), does not need to redefine *lc* and all similar entities of the class: their types will automatically follow the redeclared type of their anchor, *first_element*.

Anchoring is often useful for arguments of “set” procedures. If class *EMPLOYEE* has an attribute *assignment* of type *EMPLOYEE_ASSIGNMENT*, and an associated procedure



```
set_assignment (a: EMPLOYEE_ASSIGNMENT)
    -- Make a the employee's current assignment.
require
    exists: a /= Void
do
    assignment := a
ensure
    set: assignment = a
end
```

Warning: this is not the recommended style — see *text.pl*

it is usually preferable to use the type **like** *assignment* to declare the argument *a*. Within the given class, the effect is the same, since *assignment* is of type *EMPLOYEE_ASSIGNMENT*; but if a descendant redefines *assignment* to a more specific type — such as *ENGINEERING_ASSIGNMENT* — the signature of the procedure *set_assignment* will automatically follow.

Anchoring to *Current*

You may use *Current* as anchor. Declaring *x* of type **like** *Current* in a class *C* is equivalent to declaring it of type *C* in *C*, and redeclaring it of type *D* in any proper descendant *D* of *C*.

Among other advantages, this technique avoids lengthy redefinitions. *LINKABLE*, mentioned earlier, relies on it. A list cell has a reference to its right neighbor:



Linkable list cell

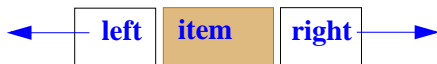
This figure and the next appeared previously on page 270.

The attribute *right* denotes that reference in class *LINKABLE*, where it is anchored to *Current*:



right: **like** *Current*

This declaration guarantees that in any more specialized version of *LINKABLE*, described by a proper descendant of class *LINKABLE*, *right* will automatically denote to objects of the descendant type. An example is class *BI_LINKABLE*, representing elements chained both ways:



Bi-linkable list cell

In this case the anchored declaration guarantees that a doubly linked list element is only used in conjunction with other elements of the same (or a more specialized) type. Another descendant of *LINKABLE* is a class describing tree nodes; here too, the anchoring guarantees that tree nodes only refer to other tree nodes, not to simple *LINKABLE* elements.

Anchoring to an expanded or generic

In **like** *x* where *x* is a query or argument, there is no particular restriction on the type *T* of *x*. In the most common case *T* will be a reference type, but it may also be anything else, such as:

- An anchored type itself — under a no-cycle requirement explained below.
- An expanded type.
- A **Formal_generic_name** representing a generic parameter of the enclosing class.
- A **Tuple_type**.

The expanded case is not very exciting because redefinition possibilities are very limited for the anchor. It enables you, however, to emphasize that a group of expanded entities must have the same type, and facilitates switching between reference and expanded status if you don't get the first time around. [NOTE: NEXT TWO SECTIONS WILL PROBABLY BE REMOVED.] → *As a consequence of “Redeclaration rule”, page 313 and “Direct conformance: expanded types”, page 396.*

The formal parameter case is more subtle. If x is of type G in a class $C[G]$, **like** x denotes the actual generic parameter corresponding to G . Declaring y : **like** x has, within the text of C , the same effect as declaring y of type G . With z of type $C[T]$ for some type T , the rules on genericity imply that $z.y$ has type T . If C has a feature $f(u: \text{like } x)$, a call $z.f(v)$ will be valid only if the type of v is exactly T — not another type conforming to T , as would be valid if u was declared just with the type G . → *“Generic Type Adaptation rule”, page 367; see also “THE TYPE OF AN EXPRESSION”, 28.11, page 782.*

The same spirit guides the interpretation of **like** t , where t is of a tuple type such as $TUPLE [A, B, C]$. If u is declared as $TUPLE [A, B, C]$, the conformance rules on tuple types let us assign to u not only a tuple such as $[a1, b1, c1]$ (with $a1$ of type A and so on) but also a longer tuple such as $[a1, b1, c1, d1, e1]$ as long as the initial items are of the requisite types (A , B and C respectively). But with u of type **like** t , only a tuple of exactly three elements will be permissible. This means that you can have your choice between a lax interpretation of tuple types (tuples of n items or more, for some n) and a restrictive one (tuples of exactly n items). The strict interpretation will be useful in particular for routine agents. → *“TUPLE TYPE CONFORMANCE”, 14.10, page 396. On the rules for agents, see*

Avoiding anchor cycles

To go from the preceding informal presentation of anchored types to their precise constraint and semantics requires that we address the issue of anchor chains and prohibit cycles.

The syntax permits x to be declared of type **like** $anchor$ if $anchor$ is itself anchored, of type **like** $other_anchor$. Although most developments do not need such anchor chains, they turn out to be occasionally useful for advanced applications. But then of course we must make sure that an anchor chain is meaningful, by excluding cycles such as a declared as **like** b , b as **like** c , and c as **like** a . The following definition helps.

DEFINITION

Anchor set; cyclic anchor

The **anchor set** of a type T is the set of entities containing, for every anchored type **like** $anchor$ involved in T :

- $anchor$.
- (Recursively) the anchor set of the type of $anchor$.

An entity a of type T is a **cyclic anchor** if the anchor set of T includes a itself.

The anchor set of $LIST$ [**like** a , $HASH_TABLE$ [**like** b , $STRING$]] is, according to this definition, the set $\{a, b\}$.

Because of genericity, the cycles that make an anchor “cyclic” might occur not directly through the anchors but through the types they involve, as with a of type $LIST$ [**like** b] where b is of type **like** a . Here we say that a type “involves” all the types appearing in its definition, as captured by the following definition.

DEFINITION

Types and classes involved in a type

The types **involved** in a type T are the following:

- T itself.
- If T is of the form $a T'$ where a is an **Attachment_mark**: (recursively) the types involved in T' .
- If T is a generically derived **Class_type** or a **Tuple_type**: all the types (recursively) involved in any of its actual parameters.

The *classes* involved in T are the base classes of the types involved in T .

$A [B, C, LIST [ARRAY [D]]]$ involves itself as well as $B, C, D, ARRAY [D]$ and $LIST [ARRAY [D]]$. The notion of *cyclic anchor* captures this notion in full generality; the basic rule, stated next, will be that if a is a cyclic anchor you may not use it as anchor: the type **like** a will be invalid.

Validity and semantics of anchored types

 The notions just introduced enable us to define the validity of anchored types. Every type has an *deanchored* version, an “unfolded form” which expands the **like**:

← “TWO-TIER DEFINITION AND UNFOLDED FORMS”, 2.11, page 100.

DEFINITION

Deanchored form of a type

The **deanchored form** of a type T in a class C is the type (Class_or_tuple_type or Formal_generic) defined as follows:

- 1 • If T is **like** **Current**: the current type of C .
- 2 • If T is **like** *anchor* where the type AT of *anchor* is not anchored: (recursively) the deanchored form of AT .
- 3 • If T is **like** *anchor* where the type AT of *anchor* is anchored but *anchor* is not a cyclic anchor: (recursively) the deanchored form of AT in C .
- 4 • If T is $a AT$, where a is an Attachment_mark: $a DT$, where DT is (recursively) the deanchored form of AT deprived of its Attachment_mark if any.
- 5 • If none of the previous cases applies: T after replacement of any actual parameter by (recursively) its deanchored form.

Although useful mostly for anchored types, the notion of “deanchored form” is, thanks to the phrasing of the definition, applicable to *any* type. Informally, the deanchored form yields, for an anchored type, what the type “really means”, in terms of its anchor’s type. It reflects the role of anchoring as what programmers might call a macro mechanism, a notational convenience to define types in terms of others.

Case 4 enables us to treat **? like anchor** as a detachable type whether the type of *anchor* is attached or detachable.



Anchored Type rule

VTAT

It is valid to use an anchored type *AT* of the form **like anchor** in a class *C* if and only if it satisfies the following conditions:

- 1 • *anchor* is either **Current** or the final name of a query of *C*.
- 2 • *anchor* is not a cyclic anchor.
- 3 • The deanchored form *UT* of *AT* is valid in *C*.

The base class and base type of *AT* are those of *UT*.

An anchored type has no properties of its own; it stands as an abbreviation for its unfolded form. You will not, for example, find special conformance rules for anchored type, but should simply apply the usual conformance rules to its deanchored form.



Other than the no-cycle requirement, the rule on anchors is liberal. In particular **an anchor’s type may be expanded**, or a **Formal_generic_name**. Anchoring is of limited benefit in these cases, since the conformance rules leave little possibility of redeclaration for an entity of expanded or formal generic types. But an anchored declaration can cause no harm, and still has the benefits of clarity and concision.

Now for the semantics. When we declare *a* as being of type **like anchor** with *anchor* of type *T* we consider *a*, for all practical purposes — such as deciding what features are applicable to *a* — to be of type *T* too. So the base type of **like anchor** will be *T*, or more generally the base type of *T* (since we allow *T* itself to be **like other_anchor** or some other non-primitive type). So in



frozen clone (other: ANY): like other is ... do ... end

we may consider, within the function’s body, that *Result* is of type *ANY*. Similarly, with

set_assignment (a: like assignment) is ... do ... end

where *assignment* is an attribute of type *EMPLOYEE_ASSIGNMENT*, we may treat *a*, within *set_assignment*, as being of that same type.



The “**current type**”, used in the **like** *Current* case, is the class name equipped with its generic parameters if applicable. So for a **like** *Current* declaration in class *PARAGRAPH* the base type is *PARAGRAPH*; in class *HASH_TABLE [G, KEY → HASHABLE]* it is *HASH_TABLE [G, KEY]*. This notion will be discussed in the next chapter.

→ “*CURRENT TYPE. FEATURES OF A TYPE*”, 12.11, page 365.

“Expansion status” means whether the type is expanded or reference. In the of anchoring to a **Formal_generic_name**, as with **like** *G* in a class *C [G]*, we shall see that the expansion status of *G* depends on every particular generic derivation: it is the same as the expansion status of the corresponding actual generic parameter. The status of **like** *G* will follow.

→ “*SEMANTICS OF GENERIC TYPES*”, 12.10, page 363.



The Anchored Type rule legitimates the use of a recursive definition of the above semantic rule. To determine the base type of **like** *anchor* we must look at the type of *anchor*, which might itself involve one or more types of the form **like** *other_anchor*, leading us to look at the type of *other_anchor* and so on. Because the Anchored Type rule requires *anchor* to be a non-cyclic anchor, this process will always terminate. This also applies to the process of determining whether the type is reference or expanded.



Anchored declaration is essentially a syntactical device: you may always replace it by explicit redefinition. But it is extremely useful in practice, avoiding much code duplication when you must deal with a set of entities (attributes, function results, routine arguments) which should all follow suit whenever a proper descendant redefines the type of one of them, to take advantage of the descendant’s more specific context.

11.11 GUARANTEEING ATTACHMENT

----ADD EXPLANATIONS

Attached, detachable

A type is **detachable** if its deanchored form is a **Class_type** declared with the ? **Attachment_mark**.

A type is **attached** if it is not detachable.

By taking the “deanchored form”, we can apply the concepts of “attached” and “detachable” to an anchored type **like** *a*, by just looking at the type of *a* and finding out whether it is attached or not.

As a consequence of this definition, an expanded type is attached.

As the following semantic definition indicates, the idea of declaring a type as attached is to guarantee that its values will never be void.

Attached type semantics

Every run-time value of an attached type is non-void (attached to an object).

In contrast, values of a detachable type may be void.

These definitions rely on the run-time notion of a *value* being attached (to an object) or void. So there is a distinction between the *static* property that an entity is attached (meaning that language rules guarantee that its run-time values will never be void) or detachable, and the *dynamic* property that, at some point during execution, its value will be attached or not. If there's any risk of confusion we may say “statically attached” for the entity, and “dynamically attached” for the run-time property of its value.

The validity and semantic rules, in particular on attachment operations, ensure that attached types indeed deserve this qualification, by initializing all the corresponding entities to attached values, and protecting them in the rest of their lives from attachment to void.

From the above semantics, the **!** mark appears useless since an absent `Attachment_mark` has the same effect. The mark exists to ensure a smooth transition: since earlier versions of Eiffel did not guarantee void-safety, types were detachable by default. To facilitate adaptation to current Eiffel and avoid breaking existing code, compilers may offer a compatibility option (departing from the Standard, of course) that treats the absence of an `Attachment_mark` as equivalent to `?`. You can then use **!** to mark the types that you have moved to the attached world and adapt your software at your own pace, class by class if you wish, to the new, void-safe convention.

11.12 STAND-ALONE TYPES

Stand-alone type

A `Type` is **stand-alone** if and only if it involves neither any `Anchored` type nor any `Formal_generic_name`.

In general, the semantics of a type may be relative to the text of class in which the type appears: if the type involves generic parameters or anchors, we can only understand it with respect to some class context. A stand-alone type always makes sense — and always makes the same sense — regardless of the context.

We restrict ourselves to stand-alone types when we want a solidly defined type that we can use anywhere. This is the case in the validity rules enabling creation of a root object for a system, and the definition of a once function.

Genericity

12.1 OVERVIEW

The types discussed so far were directly defined by classes. The *genericity* mechanism, still based on classes, gives us a new level of flexibility through **type parameterization**. You may for example define a class as *LIST* [*G*], yielding not just one type but many: *LIST* [*INTEGER*], *LIST* [*AIRPLANE*] and so on, parameterized by *G*.

Parameterized classes such as *LIST* are known as **generic classes**; the resulting types, such as *LIST* [*INTEGER*], are **generically derived**. “Genericity” is the mechanism making generic classes and generic derivations possible.

Two forms of genericity are available: with *unconstrained* genericity, *G* represents an arbitrary type; with *constrained* genericity, you can demand certain properties of the types represented by *G*, enabling you to do more with *G* in the class text.

The discussion of generically derived types will proceed as with other kinds of type in the previous chapter: to define the semantics of a type, it suffices to say whether it is reference or expanded, and to define its **base type**, always a *Class_or_tuple_type*. If the type is itself a *Class_or_tuple_type*, we must also define its **base class**, which determines its instances. For example *LIST* [*INTEGER*] has *LIST* as its base class, and is its own base type.

A subsequent chapter discusses the related *conformance* properties. → Chapter 14.

12.2 GENERIC CLASSES

To obtain generically derived types, we start from **generic classes** such as *LIST*, with one or more *formal generic parameters* such as *G*.



Generic classes describe flexible structures having variants parameterized by types. Often these are **container data structures**, used to gather objects of various possible types; examples include lists, stacks, arrays and the like, which contain objects of arbitrary type. The generic parameters of such classes specify the types of objects to be kept in the container structures, such as the elements of an array.

Container data structures were mentioned in 10.8, page 268.

The following examples from EiffelBase show beginnings (Class_header followed by Formal_generics) of classes with unconstrained generic parameters:



```
deferred class TREE [G] ...
class LINKED_LIST [G] ...
class ARRAY [G] ...
```

In each case, *G* is a **formal generic parameter** of the class, representing the types of objects to be kept in an instance of the class – a tree, a linked stack, an array. Classes may have more than one formal generic parameter; the next section will give an example with two parameters.

To derive a type from a generic class — called the **base class** of the derivation — you must provide a type, called an **actual generic parameter**, for each of the formal generic parameters of the base class. This will yield a **generically derived** type. The derived type is expanded if the base class is expanded, a reference type otherwise. Generic derivation, applied to the above base classes, will yield types such as



```
TREE [INTEGER]
TREE [PARAGRAPH]
LINKED_LIST [PARAGRAPH]]
TREE [TREE [PARAGRAPH]]
ARRAY [LINKED_LIST [TREE [LINKED_LIST [PARAGRAPH]]]]
```

Instances of the first type represent trees of integers; instances of the second one represent trees of paragraphs (that is to say, trees of instances of the reference type *PARAGRAPH*); and so on. The base classes are, respectively, *TREE*, *TREE*, *LINKED_LIST*, *TREE* and *ARRAY*.

Since all these base classes have exactly one formal generic parameter, each of the above generically derived types is obtained by providing one actual generic parameter. The actual generic parameters are *INTEGER* for the first example, *PARAGRAPH* for the second and third, *TREE [TREE [PARAGRAPH]]* for the fourth, *LINKED_LIST [TREE [LINKED_LIST [PARAGRAPH]]]* for the last.

The actual generic parameter is a type; it may itself be generically derived; the last two examples illustrate this possibility, which leads to nested genericity without any limit on the depth of nesting.

In the syntax specification, the place where all this appears is the *Class_type* construct, introduced [earlier](#) as

```
Class_type  $\triangleq$  Class_name [Actual_generics]
```

Actual_generics hasn't been defined until now. Here it is:

Actual generic parameters

```
Actual_generics  $\triangleq$  [" Type_list "]"
Type_list  $\triangleq$  {Type "," ... }+
```

← This was in the previous chapter, page [328](#).

SYNTAX

SYNTAX

12.3 GENERIC CLASSES AND GENERIC DERIVATIONS

The construct that makes a class generic is `Formal_generics`, optionally appearing after the `Class_header` of a `Class_declaration`, with this structure:

← `Class_declaration` was given in chapter 4, which on page 128 previewed the syntax shown here.



Formal generic parameters

```

Formal_generics ≜ "[" Formal_generic_list "]"
Formal_generic_list ≜ {Formal_generic ", "...}+
Formal_generic ≜ [frozen] Formal_generic_name
                  [Constraint]
Formal_generic_name ≜ [?] Identifier
  
```

and a straightforward validity constraint:



Formal Generic rule *VCFG*

A `Formal_generics` part of a `Class_declaration` is valid if and only if every `Formal_generic_name` G in its `Formal_generic_list` satisfies the following conditions:

- 1 • G is different from the name of any class in the universe.
- 2 • G is different from any other `Formal_generic_name` appearing in the same `Formal_generics` part.

→ A rule also applies to the `Constraint` part: "Generic Constraint rule", page 357.

Adding the **frozen** qualification to a formal generic, as in D [**frozen** G] rather than just C [G], means that conformance on the corresponding generically derived classes requires identical actual parameters: whereas C [U] conforms to C [T] if U conforms to T , D [U] does not conform to D [T] if U is not T .

Adding the **?** mark to a `Formal_generic_name`, as in $?G$, means that the class may declare *self-initializing* variables (variables that will be initialized automatically on first use) of type G ; this requires that any actual generic parameter that is an attached type must also be self-initializing, that is to say, make *default_create* from *ANY* available for creation.

The optional `Constraint` part of the form \rightarrow *CONSTRAINING_TYPE* puts a requirement on acceptable actual generic parameters: they must conform to the *CONSTRAINING_TYPE*. If it is not present, any type will do.

Let's make the basic terminology precise:



Generic class; constrained, unconstrained

Any class declared with a **Formal_generics** part (constrained or not) is a **generic class**.

If a formal generic parameter of a generic class is declared with a **Constraint**, the parameter is **constrained**; if not, it is **unconstrained**.

A generic class is itself **constrained** if it has at least one constrained parameter, **unconstrained** otherwise.

A generic class does not describe a type but a template for a set of possible types. To obtain an actual type, you must provide an **Actual_generics** list, whose elements are themselves types. This has a name too, per the following definition.

Generic derivation, non-generic type

The process of producing a type from a generic class by providing actual generic parameters is **generic derivation**.

A type resulting from a generic derivation is a **generically derived type**, or just **generic type**.

A type that is not generically derived is a **non-generic type**.



It is preferable to stay away from the term “generic instantiation” (sometimes used in place of “generic derivation”) as it creates a risk of confusion with the normal meaning of “instantiation” in object-oriented development: the *run-time* process of obtaining an object from a class.

Among the above examples, **PARAGRAPH** is non-generic, as a class and as a type (any non-generic class is also a type). **LINKED_LIST**, **TREE** and **ARRAY** are generic classes; to produce a generic derivation from one of them, you choose a suitable actual generic parameter, and get a generically derived type such as **LINKED_LIST [INTEGER]**.

The expansion status of a generically derived type **T** follows from its base class, independently of the actual generic parameters: **T** is expanded if its base class is an expanded class; otherwise it is a reference type.

12.4 SELF-INITIALIZING FORMAL PARAMETERS

---- EXPLAIN

Self-initializing formal parameter

A `Formal_generic_parameter` is **self-initializing** if and only if its declaration includes the optional `?` mark.

This is related to the notion of self-initializing *type*: a type which makes *default_create* from *ANY* available for creation. The rule will be that an actual generic parameter corresponding to a self-initializing formal parameter must itself, if attached, be a self-initializing type.

12.5 CONSTRAINED AND UNCONSTRAINED GENERICITY

If a formal generic parameter is constrained, appearing as $G \rightarrow T$, the constraint T determines what operations are applicable, in the class to an entity of type G : the features of T . This will be formalized very simply by defining the base type of G , in this case, as being T .

In the case of unconstrained genericity, we don't know anything about future actual generic parameters: in $C [G]$, G can represent any type. The only operations that we can apply in this case are those of *ANY*, since we know every Eiffel class conforms to *ANY*. We'll in fact allow these operations, and treat G as if it were constrained by *ANY*.

← “*Universal Conformance principle*”, page 173.

This observation allows us to simplify the validity and semantics by treating in the same way all formal generic parameters, constrained and unconstrained, thanks to the following convention, which also allows us in every case to talk about “the constraint” of a formal generic parameter:

Constraint, constraining types of a `Formal_generic`

The **constraint** of a formal generic parameter is its `Constraint` part if present, and otherwise *ANY*.

Its **constraining types** are all the types listed in its `Constraining_types` if present, and otherwise just *ANY*.

--- For the language description, it's convenient to avoid treating

A straightforward constraint applies to unconstrained generic derivations: a generically derived type of the form $C [T, \dots]$, where C does not declare any constraints for its generic parameters if any, is valid if and only if:

- C is indeed a generic class.
- The number of `Type` components T, \dots in the `Actual_generics` list is the same as the number of `Formal_generic` parameters in the `Formal_generic_list` of C 's declaration.

This property does not appear as a separate validity constraint since, thanks to the ----- REWRITE notion of unfolded form, it will follow as a special case of the validity rule for the constrained case, where we treat unconstrained genericity as constrained by *ANY*.

This is a "constraint" on "unconstrained" genericity. Sometimes language meets metalanguage.

12.6 CONSTRAINED GENERICITY

In the above unconstrained examples of genericity, any type was acceptable as actual generic parameter; this is because we do not require any special property of the objects to be entered into an array, inserted into a tree or pushed onto a stack. As long as operations applicable to all objects (such as assignment, copying or equality testing) are available, we can write the generic class, for example *TREE [T]*, without any specific knowledge about the actual types to be used for *T*.



In some cases, however, you will need a guarantee that these types possess specific properties, so that the class text may apply certain operations to the corresponding objects. A typical example is a generic class *VECTOR [T ...]* describing vectors, which must support an addition operation. To add two vectors, you need the ability to add two vector elements; in other words, you need an addition operation on *T*. Then *T* cannot be an arbitrary type.

With constrained genericity, you can guarantee that *T* supports addition, by requiring any actual generic parameter for *T* to be based on a descendant of a class that includes an addition routine. The class will appear as

```
class VECTOR [G -> NUMERIC]...
```

in reference to class *NUMERIC* of the Kernel Library, describing numerical values and having among others a feature *plus alias* "+" representing addition. Numerical classes such as *INTEGER* and *REAL* are descendants of *NUMERIC*, as will be any class that you want to declare as providing a number or number-like facility (for example class *VECTOR* itself). The *Constraint* part *-> NUMERIC* indicates that a generic derivation *VECTOR [SOME_TYPE]* will be valid if and only if *SOME_TYPE* conforms to *NUMERIC*. So you may use *VECTOR [INTEGER]*, *VECTOR [REAL]*, or even *VECTOR [VECTOR [INTEGER]]* if you have made *VECTOR* itself inherit from *NUMERIC*, but not *VECTOR [PARAGRAPH]* if class *PARAGRAPH* is not a descendant of *NUMERIC*.

Class *HASH_TABLE* of EiffelBase provides another example of constrained generic class. This class describes tables of elements, retrievable through associated keys. Its text begins with

```
class HASH_TABLE [G; KEY -> HASHABLE]...
```


The class has two generic parameters. The first one, *G*, plays the same role as those encountered in the previous section; it stands for the type of table elements, and is unconstrained. The second one, *KEY*, is constrained by the Kernel Library class *HASHABLE*.

The constraint means that the base class of any actual generic parameter used for *KEY* must be a descendant of the constraining class, *HASHABLE*. *HASHABLE* is a simple Kernel Library class introducing a function

The → symbol is reminiscent of the arrow used in inheritance diagrams.

```
hash_code: INTEGER
    --Hash_code value
deferred
end
```

In other words, keys must be "hashable" into integer values. An example of a class that inherits from *HASHABLE* is the Kernel Library class *STRING*, describing character strings, for which a standard *hash_code* function is provided. An example of a type generically derived from *HASH_TABLE* is

```
HASH_TABLE [PARAGRAPH; STRING]
```

As illustrated by these examples, the basic syntax for constrained formal generic parameters includes, after the parameter, a **Constraint** of the form

```
→ Class_or_tuple_type
```

The effect of such a **Constraint**, if present, is to restrict allowable actual generic parameters to types that conform to the given **Class_or_tuple_type**.

Recall that a type *C* conforms to a type *B* if the base class of *C* is a descendant of the base class of *B*; also, if *C* is generically derived, its actual generic parameters must (recursively) conform to those of *B*. In the *HASH_TABLE* case conformance is ensured by the property that *STRING*, as specified by the Kernel Library, inherits from *HASHABLE*.

The next chapter covers conformance.

Two supplementary facilities are available:

- You can require certain creation procedures.
- You can use multiple constraints.

→ See "[CREATING INSTANCES OF FORMAL GENERICS](#)", 20.9, page 543 for a full discussion.

The first of these enables you to write something like

```
class D [G → CONST create cp1, cp2, ... end] ...
```

where *cp1*, *cp2*, ... must be procedures of type *CONST*. The purpose — as explained in detail in the chapter on creation — is to allow creation of objects of type *G*: within the class, with *x* declared of type *G*, you can use a creation instruction of the form **create** *x.cp1* (*actuals*) and similarly for the other listed procedures, assuming valid *actuals* arguments. A generic derivation *D [T]* will then require *T* to declare its versions of *cp1*, *cp2*, ... as creation procedures. In *CONST* itself, *cp1*, *cp2*, ... must be procedures, but they do not have to be *creation* procedures, since what matters is to be able to use them to create instances of actual generic parameters such as *T*.

The second facility enables you to specify multiple constraints, as in

```
class D [G -> {CONST1; CONST2, CONST3}] ...
```

meaning that any actual parameter *T* in a generic derivation *D [T]* must conform to **all** of *CONST1*, *CONST2*, *CONST3*.

It is in fact possible to combine both of these two facilities, as in

```
class D [G -> {CONST1; CONST2 create cp1, cp2, ... end}] ...
```



More generally, as the rest of this chapter will show, multiple constraints significantly complicate the syntax and validity of generic constraints, as well as the definition of the base type for formal generic parameters. This is a typical “borderline” facility, whose presence in the language is subject to criticism.

Most developments do not need it, but there are cases, mostly involving libraries, when working without multiple constraints would make things awkward. When you have control over all classes involved, you can in principle get away with single constraints only, achieving the effect of *D [G -> {CONST1, CONST2}]* by using *D [G -> {CONST12}]*, after writing a class *CONST12* that inherits from *CONST1* and *CONST2*. But this doesn’t work with pre-existing classes, especially those from the Kernel Library and other fundamental libraries: with multiple constraints you can write *D [G -> {COMPARABLE, NUMERIC}]*, which will accept *INTEGER* or *REAL* as an actual generic parameter; but defining a class *COMPARABLE_NUMERIC* that inherits from *COMPARABLE* and *NUMERIC* won’t help you since *INTEGER*, *REAL* and the like do not know it. And you cannot define new classes — *NUMERIC_HASHABLE* and so on — for every potentially useful combination.

So even though multiple constraints are useful for only a minority of cases and users, they are *very* desirable to these users for those cases, explaining why the language supports them in spite of the added complication.

12.7 RULES ON CONSTRAINED GENERICITY

Now for the precise syntax and validity of constrained genericity. The construct that remains to be specified is **Constraint**:



Generic constraints

```

Constraint ≙ "->" Constraining_types
           [Constraint_creators]

Constraining_types ≙ Single_constraint | Multiple_constraint

Single_constraint ≙ Type [Renaming]

Renaming ≙ Rename end

Multiple_constraint ≙ "{" Constraint_list "}"

Constraint_list ≙ {Single_constraint "," ... }+

Constraint_creators ≙ create Feature_list end
  
```

There are two validity rules. One governs the **Constraint** part of the declaration of a constrained generic class; the other, complementing the Unconstrained Genericity rule, governs the validity of a type derived from In addition, of course, the base class must exist in the universe; this is a consequence of the Class Type rule. First:



Generic Constraint rule

VTGC

A **Constraint** part appearing in the **Formal_generics** part of a class **C** is valid if and only if it satisfies the following conditions for every **Single_constraint** listing a type **T** in its **Constraining_types**:

- 1 • **T** involves no anchored type.
- 2 • If a **Renaming** clause **rename rename_list end** is present, a class definition of the form **class NEW inherit BT rename rename_list end** (preceded by **deferred** if the base class of **T** is deferred), where **BT** is the base class of **T**, would be valid.

This is a validity "constraint" on the generic "constraints" of Eiffel classes. Sometimes language meets metalanguage.

There is no requirement here on the `Constraint_creators` part, although in most cases it will list names (after `Renaming`) of creation procedures of the constraining types. The precise requirement is captured by other rules.

Condition 2 implies that the features listed in the `Constraint_creators` are, after possible `Renaming`, names of features of one or more of the constraining types, and that no clash remains that would violated the rules on inheritance. In particular, you can use the `Renaming` either to merge features if they come from the same seeds, or (the other way around) separate them.

If T is based on a deferred class the fictitious class `NEW` should be declared as `deferred` too, otherwise it would be invalid if T has deferred features. On the other hand, `NEW` cannot be valid if T is based on a frozen class; in this case it is indeed desirable to disallow the use of T as a constraint, since the purpose of declaring a class `frozen` is to prevent inheritance from it

The last observation suggests a name for the resulting features:

Constraining creation features

If G is a formal generic parameter of a class, the **constraining creators of G** are the features of G 's `Constraining_types`, if any, corresponding after possible `Renaming` to the feature names listed in the `Constraining_creators` if present.



Constraining creators should be creation procedures, but not necessarily (as seen below) in the constraining types themselves; only their instantiatable descendants are subject to this rule.

The usual situation, with a declaration involving a `Constraint_creators`

```
class D [G -> CONST create cp1, cp2, ... end] ...
```

the names listed, `cp1`, `cp2`, ..., should denote procedures of `CONST`. They don't have to be *creation* procedures of `CONST` — this will be required only in the actual generic parameter, as stated by the next rule — and can in fact be deferred, but they have to be known procedures of `CONST` so that we can assess the validity of a creation call such as `create x.cp1 (actuals)`, especially the validity of the chosen *actuals*.

--- EXPLAIN CLAUSE 4 (INCLUDING CASE OF TUPLES ---

Next, the rule for generically derived types. The two properties already cited for the unconstrained case still apply: the number and types of actual parameters must match those of the formal parameters. In addition:

- Wherever a formal generic parameter is constrained, the corresponding actual parameter must conform to the constraining type or types.

- If there is a **Constraint_creators** part requiring some creation procedures, these must indeed be creation procedures in the actual generic parameter.

Here is the precise formulation:



Generic Derivation rule

VTGD

Let C be a generic class. A **Class_type** CT having C as base class is valid if and only if it satisfies the following conditions for every actual generic parameter T and every **Single_constraint** U appearing in the constraint for the corresponding formal generic parameter G :

- 1 • The number of Type components in CT 's **Actual_generics** list is the same as the number of **Formal_generic** parameters in the **Formal_generic_list** of C 's declaration.
- 2 • T conforms to the type obtained by applying to U the generic substitution of CT .
- 3 • If C is expanded, CT is generic-creation-ready.
- 4 • If G is a self-initializing formal parameter and T is attached, then T is a self-initializing type.

In the case of unconstrained generic parameters, only condition **1** applies, since the constraint in that case is *ANY*, which trivially satisfies the other two conditions.

Condition **3** follows from the semantic rule permitting “lazy” creation of entities of expanded types on first use, through *default_create*. Generic-creation-readiness (defined next) is a condition on the actual generic parameters that makes such initialization safe if it may involve creation of objects whose type is the corresponding formal parameters. → Page 360.

Condition **4** guarantees that if C relies, for some of its variables of type G , on automatic initialization on first use, T provides it, if attached (remember that this includes the case of expanded types), by making *default_create* from *ANY* available for creation. If T is detachable this is not needed, since *Void* will be a suitable initialization value.



At first the phrasing of clause **2** seems more complicated than necessary: why must the actual generic parameter conform not just to D but to “the type obtained by applying to U by ...”? This is to permit *recursive generic constraints*, as detailed next. For most practical cases, however, you can understand clause **2** as if it just read

The role of this condition is to make sure that if the operations of a class C [..., G , ...] may include creations on targets of the formal generic type G , any associated actual generic parameter T will support such creations. The basic

“*T* conforms to the constraining type”

12.8 CONSTRAINTS AND CREATION

Consider a formal generic parameter G ; under what conditions can we create an object of type G , for example through an instruction `create x .make (...)` with x of type G in one of the routines of the class? Including a `Constraint_creators` enables you to specify the applicable creation procedures for G , as in `$G \rightarrow$ CONST create make end`. The corresponding actual generic parameters will then have to provide the listed features, here `make`, as creation procedures when needed. This is not a constraint on all generic derivations, however; only on those raising the possibility of a creation on the corresponding parameter. So at this stage we don't have a constraint, just a definition:

Generic-creation-ready type

A type of base class C is **generic-creation-ready** if and only if every actual generic parameter T of its deanchored form satisfies the following conditions:

- 1 • If the specification of the corresponding formal generic parameter includes a `Constraint_creators`, the versions in T of the constraining creators for the corresponding formal parameter are creation procedures, available for creation to C , and T is (recursively) generic-creation-ready.
- 2 • If T is expanded, it is (recursively) generic-creation-ready.

Although phrased so that it is applicable to any type, the condition is only interesting for generically derived types of the form $C [\dots, T, \dots]$. Non-generically-derived types satisfy it trivially since there is no applicable T .

The role of this condition is to make sure that if class $C [\dots, G, \dots]$ may cause a creation operation on a target of type G — as permitted only if the class appears as $C [\dots, G \rightarrow \text{CONST create } cp1, \dots \text{end}, \dots]$ — then the corresponding actual parameters, such as T , will support the given features — the “constraining creators” — as creation procedures.

It might then appear that generic-creation-readiness is a validity requirement on *any* actual generic parameter. But this would be more restrictive than we need. For example T might be a deferred type; then it cannot have any creation procedures, but that’s still OK because we cannot create instances of T , only of its effective descendants. Only if it is possible to **create** an actual object of the type do we require generic-creation-readiness. Overall, we need generic-creation-readiness only in specific cases, including:

- For the creation type of a creation operation: conditions [4](#) of the [Creation Instruction rule](#) and [3](#) of the [Creation Expression rule](#). → Pages [553](#) and [562](#).
- For a [Parent](#) in an [Inheritance](#) part: condition [6](#) of the [Parent rule](#). ← Page [178](#).
- For an expanded type: condition [3](#) of the just seen [Generic Derivation rule](#). ← Page [359](#).

----- NEXT SECTIONS OBSOLETE

Clause --- covers the case of a [Constraint](#) including a [Constraint_creators](#) and complements the preceding rule (Generic Constraint). In

class $D [G \rightarrow \text{CONST create } cp1, cp2, \dots \text{end}] \dots$

the Generic Constraint rule required $cp1, cp2, \dots$ to be procedures of [CONST](#). In a generic derivation $D [T]$, T must be, as per clause [2](#), a type conforming to [CONST](#); in addition, clause --- tells us that T must make sure to specify $cp1, cp2, \dots$ as creation procedures. (As a consequence, they cannot for example be deferred.)

Both of these clauses apply to every constraining type in the case of a multiple constraint $D [G \rightarrow \{ \text{CONST1}, \text{CONST2} \}]$

No specific validity rule applies to the generic constraints themselves ([CONST](#), [CONST1](#), [CONST2](#)). A generic constraint must simply be a valid type. It might even involve a generic parameter, or even *be* a generic parameter; this is the case of “recursive generic constraints”, the topic of the next section.

12.9 RECURSIVE GENERIC CONSTRAINTS



(The case described in this section does not arise in elementary uses, and may be skipped in a first reading.)

To understand the last part of clause 2 of the Constrained Genericity rule, assume you want to define a class as

```
class C [G; H -> ARRAY [G]] ...
```



This makes perfect sense and the intent is clear: you want to allow any type of the form $C [T, U]$ where T is an arbitrary type and U is $ARRAY [T]$ or a type conforming to $ARRAY [T]$. So the following will be valid The Class Type rule appeared on page 333.

```
C [INTEGER; ARRAY [INTEGER]]
C [POLYGON; ARRAYED_LIST [POLYGON]]
-- Where ARRAYED_LIST is a descendant of ARRAY
```

But for example $C [INTEGER, REAL]$ is not valid. Similarly, you should be able to define

```
class C [G ->H; H -> G] ...
```



meaning: the first actual generic parameter must conform to the first, and conversely. Only derivations of the form $C [T, T]$, using the same type as actual generic parameter, will be valid. Unlike the first example, this scheme seems useless, but there is no reason to disallow it.

This explains the phrasing of clause 2 of the Constrained Genericity rule. The simpler phrasing

“ T conforms to the constraining type”

is appropriate in ordinary, non-recursive cases; but in our first example $ARRAY [INTEGER]$ does not conform to $ARRAY [G]$; actually this conformance question is meaningless since there usually won't even be a type G in the class that wants to use $C [INTEGER; ARRAY [INTEGER]]$. Similarly, in the $C [G ->H; H -> G]$ example, if we want to use $C [T, T]$ in a certain class other than C , the questions “does T conform to G ?” and “does T conform to H ?” are meaningless in that class.

For such conformance questions to become meaningful, we must first replace, in the constraint, any occurrence of a formal parameter by the corresponding actual parameter. Hence the rephrased clause:

“ T conforms to the type obtained from the constraining type by replacing every occurrence of a formal generic parameter of C by the corresponding actual generic parameter in CT .”

12.10 SEMANTICS OF GENERIC TYPES

We must now define the semantics of types involving genericity. This includes both generically derived types and **Formal_generic_name** (covering the formal generic parameters themselves, when used as types within the class text).

As noted in the previous chapter, defining the semantics of a type involves saying whether it is expanded or reference, and specifying its base type, as well as its base class if it is a **Class_type**.

← “*Type Semantics rule*”, page 333.

For a generically derived **Class_type** the definition is an immediate generalization of the **non-generic** case:

← “*Non-generic class type semantics*”, page 335.



Generically derived class type semantics

A generically derived **Class_type** of the form $C [\dots]$, where C is a generic class, is expanded if C is an expanded class, reference otherwise. It is its own base type, and its base class is C .

So **LINKED_LIST [POLYGON]** is its own base type, and its base class is **LINKED_LIST**.

The other case is formal generics. In a generic class $C [\dots, G, \dots]$, a formal parameter G , constrained or unconstrained (syntactically known as a **Formal_generic_name**), stands for any type to be provided as actual parameter in generic derivations of the class. Within the text of C , you may use G wherever the syntax requires a type.

For example, the EiffelBase text of

```
class HASH_TABLE [G; KEY → HASHABLE]...
```

declares a number of features using G or **KEY** as type of an argument, result or local variable. Typical is the function

```
item (access_key: KEY): G
    -- Item associated with access_key, if present;
    -- otherwise default value of type G.
do... Routine body omitted ... end
```

The type of the function result in the actual class is not exactly G but like 'last_put', where 'last_put' is an attribute of type G. See 11.10, page 339 below, on such "anchored" types.

which uses both of the formal generic parameters as **Formal_generic_name** types.

It is in fact easier to start with the constrained case. For a constrained parameter such as **KEY**, the only available information is provided by the constraining type, here **HASHABLE**; the features of that type's base class are the only operations that we know can be applied to entities of the **Formal_generic_name** type. The rule follows, applicable to the case of a single constraint (the next section will address multiple constraints):



Base type of a single-constrained formal generic

The base type of a constrained `Formal_generic_name` G having as its constraining types a `Single_constraint` listing a type T is:

- 1 • If T is a `Class_or_tuple_type`: T .
- 2 • Otherwise (T is a `Formal_generic_name`): the base type of T if it can be determined by (recursively) case 1, otherwise `ANY`.

The definition is never cyclic since the only recursive part is the use of case 1 from case 2.

Case 1 is the common one: for $C [G \rightarrow T]$ we use as base type of G , in C , the base type of T . We need case 2 to make sure that this definition is not cyclic, because we permit cases such as $C [G, H \rightarrow D [G]]$, and as a consequence cases such as $C [G \rightarrow H, H \rightarrow G]$ or even $C [G \rightarrow G]$ even though they are not useful; both of these examples yield `ANY` as base types for the parameters.

As a result of the definition of “constraining types”, the base type of an unconstrained formal generic, such as G in $C [G]$, is also `ANY`.

In $C [G \rightarrow I, I \rightarrow T, H \rightarrow E [G, H]]$ (unlikely to arise in practice):

- The constraint of I is T (case 1).
- Applying constraint of G is T , the base type of I : case 2 first gives .
- Without the substitutio

As usual for a type that is not a `Class_or_tuple_type`, the base class of a `Formal_generic_name` type is its base type’s base class. So `HASHABLE`, in the class `HASH_TABLE [G; KEY → HASHABLE]`, is both the base type and the base class of `KEY`. ← See the Base rule, page 332.

What about an unconstrained `Formal_generic_name` such as G in `HASH_TABLE`? Every object ever manipulated by a system is an instance of some class, and every developer-written class is a descendant of the universal library class `ANY`. In other words, `HASH_TABLE` could be equivalently declared as ← “ANY”, 6.5, page 172; see also chapter 35 for more details.

```
class HASH_TABLE [G → ANY; KEY → HASHABLE]...
```

This is a general rule: we consider an unconstrained generic parameter as if it were constrained by `ANY`. Hence the definition of the base type for unconstrained generics, a special case of the preceding rule:



Base type of an unconstrained formal generic

The base type of an unconstrained `Formal_generic_name` type is `ANY`.

This also enables us to consider that every formal generic parameter has a **constraining type**, taking it to be *ANY* for an unconstrained parameter.

The last definitions do not give the full semantics of a **Formal_generic_name** type, but only its base type. We also need to specify whether the type is reference or expanded. This is, however, the one case in which we can't know for sure: only the actual parameter passed in a particular generic derivation will tell.



Reference or expanded status of a formal generic

A **Formal_generic_name** represents a reference type or expanded type depending on the corresponding status of the associated actual generic parameter in a particular generic derivation.

12.11 CURRENT TYPE, FEATURES OF A TYPE

This discussion of genericity — now complete except for the special case of multiple constraints covered below — leads to a notion that will be convenient in future discussions. The presentation of classes noted that every Eiffel construct is part of a class, the *current class*. Often, what we will need is not just a class but a type. Hence the notion of *current type*. As long as classes were not generic, the current type was the same as the current class; but now the notion becomes more interesting, although straightforward.

← “*THE CURRENT CLASS*”, 4.5, page 117.

Assume that we are asked “what is the type of valid targets for *f*?”, where *f* is a feature of a generic class $C[G]$. The answer is, of course, $C[G]$ itself. Answering “the current class” would not do, since *C* by itself is not a type — only a type template, which yields a type if we provide an appropriate generic parameter.

This is one of the lessons of this chapter: the concepts of class and type — although closely related since every type is based on a class— are not identical. The difference comes not only from genericity but also from anchoring; the answer to the question “what is the base type of *like Current* in *C*?” would also be $C[G]$.

This will be called the *current type*:



Current type

Within a class text, the **current type** is the type obtained from the current class by providing as actual generic parameters, if required, the class's own formal generic parameters.

Clearly, the base class of the current type is always the current class.

In the same vein, since the type will often be our first source of information — before the underlying class — it is also useful to allow ourselves to extend the notion of “features of a class”:



Features of a type

The features of a type are the features of its base class.

These are the features applicable to the type’s instances (which are also instances of its base class).

You may note in particular that with genericity we often need to refer to the type rather than the class. If a generic class $C [G]$ has a feature

$$f(x: G)$$

then for a of type $C [T]$ (a type generically derived from C by using T as actual generic parameter) a call of the form

$$a.f(y)$$

requires an argument y of type T (not G , which is only a placeholder within the text of class C). This means that to understand f and its type properties fully we need to consider not just as a *feature of a certain class* (the class C) but as a *feature of a certain type* (the type $C [T]$).

12.12 APPLYING GENERICITY TO TYPES

Genericity means that we must be careful when using terms such as “the type of an expression” or “the type of a feature” if a generic derivation is involved. Consider a generic class

```
class C [G] feature
  some_query: G
  some_routine (arg: G) is do ... end
  ... Other features ...
end
```

and a client D with a declaration

$$x: C [T]$$

for some type T , for example *INTEGER* or *ARRAY [REAL]*. In the context of class D we may ask the questions:

- What is the type of $x.some_query$?
- What expressions y are valid in a call $x.some_routine (y)$?

Viewed from within C the type of $some_query$ is G , but this makes no sense in the context of $x.some_query$ as used in D , where we may consider that G is simply a placeholder for the actual generic parameter, T in this case -

Generic substitution

Every type T defines a mapping σ from names to types known as its **generic substitution**:

- 1 • If T is generically derived, σ associates to every **Formal_generic_name** the corresponding actual parameter.
- 2 • Otherwise, σ is the identity substitution.

Similarly, an argument y that we will pass in the call $x.some_routine(y)$ must be of type G or conforming.

These observations lead to the following substitution rule:

Generic Type Adaptation rule

The signature of an entity or feature f of a type T of base class C is the result of applying T 's generic substitution to the signature of f in C .

The signature include both the type of an entity or query, and the argument types for a routine; the rule is applicable to both parts.

In the examples cited this yields the type T , as desired, both as the result type of $x.some_query$ and as the type to which arguments to $x.some_routine(...)$ must conform.

12.13 THE CASE OF MULTIPLE CONSTRAINTS



(This last section covers an advanced technique needed only in special cases. On first reading you may skip to the next chapter.)

As noted, it is possible for a **Formal_generic_name** to have several constraints, as in

```
class D [G -> {CONST1, CONST2, CONST3}] ...
```

The role of the base class and type, as usual, is to tell us what features f we may use for a call $x.f(...)$ for x of type G . In the case of a single constraint $CONST1$, the answer was simply: those of $CONST1$.

Here, the basic idea is just as straightforward: we will accept f as long as it denotes a feature in *any* of the constraining types.

Of course, the same feature name might denote features in several of these types. That's not to scare us, since we know that any valid actual generic parameter T for G will have to inherit from all of the $CONST_i$ and hence resolve the conflicts according to the rules of the preceding chapters (renaming, sharing or select under repeated inheritance). But this may still leave some ambiguities as to what $x.f(...)$ means for x of type G in class D . To keep matters simple we take the following rule:

----- TO BE UPDATED TO ACCOUNT FOR NEW RENAMING ----

- 1 • If all the matching f in the constraining types have a common seed (meaning they all come from a single feature of a common ancestor, as would be the case if f is *equal* or *print* from class *ANY*), there won't be any problem: in any acceptable actual generic parameter T for G , either the corresponding version of f will be shared, or a **select** will designate one of the versions as the official one for T .
- Otherwise, there will be a name clash that T will have to resolve through renaming, but we don't want to go into this. We just renounce the feature for entities of type G .

• -----

•

Generically constrained feature name

Consider a generic class C , a constrained **Formal_generic_name** G of C , a type T appearing as one of the **Constraining_types** for G , and a feature f of name $fname$ in the base class of T . The **generically constrained names** of f for G in C are:

- 1 • If one or more **Single_constraint** clauses for T include a **Rename** part with a clause $fname$ **as** $ename$, where the **Feature_name** part of $ename$ (an **Extended_feature_name**) is $gname$: all such $gname$.
- 2 • Otherwise: just $fname$.

•

The following validity constraint expresses this rule:

--- EXPLAIN CLAUSE 2 ---



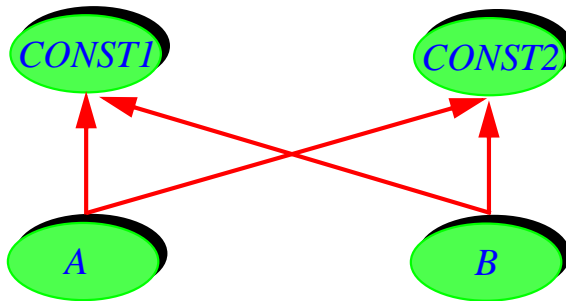
Multiple Constraints rule

VTMC

A feature of name *fname* is applicable in a class *C* to a target *x* whose type is a Formal_generic_name *G* constrained by two or more types *CONST1*, *CONST2*, ..., if and only if it satisfies the following conditions:

- 1 • At least one of the *CONST_i* has a feature available to *C* whose generically constrained name for *G* in *C* is *fname*.
- 2 • If this is the case for two or more of the *CONST_i*, all the corresponding features are the same.

Can we stop here? Not quite. We do need a precise notion of base type reflecting the Multiple Constraints rule. Although there is nothing inherently difficult in the rule, turning it into a definition of the base type for a multiply constrained Formal_generic_name requires some care. Intuitively this base type should be the “lowest common ancestor” of the constraints *CONST1*, *CONST2*, ..., but there is no such notion: the set of common ancestors — a non-empty set since it contains at least *ANY* — doesn’t necessarily include one that inherits from all the others, as in this situation if there are no other non-kernel classes involved:



*No lowest
common
ancestor*

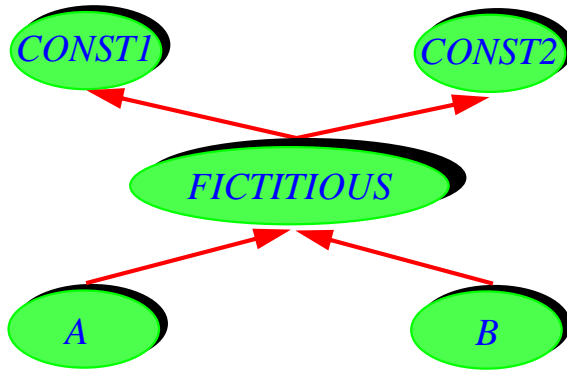
To obtain a theoretical answer (the practical answer being given by the informal rule above), we simply build a fictitious “lowest common ancestor” with all conflicts removed. Hence the definition:



Base type of a multi-constraint formal generic type

The base type of a multiply constrained Formal_generic_name type is a type generically derived, with the same actual parameters as the current class, from a fictitious class with none of the optional parts except for Formal_generics and an Inheritance clause that lists all the constraining types as parents, with the given Renaming clause if any, and resolves any conflicts between potentially ambiguous features by further renaming them to new names not available to developers.

This definition is a little as if we decided to replace the above inheritance structure by the following one, preventing *A* and *B* from resolving, each in its own desired manner, any name clashes that might arise between features of *CONST1* and *CONST2*.



*Artificial
lowest
common
ancestor*

FICTITIOUS represents the constraint we would be using if we were limited to a single constraint for *G*: features applicable to *G*, assuming the declaration `class D [G → {CONST1, CONST2}]`, are those you could use if the declaration were `class D [G → {FICTITIOUS}]`.

Even though the basic idea of this definition is simple (you may apply to a multiply constrained *Formal_generic* any of the constraining types' features that are not ambiguous), the recourse to a fictitious type is not too pleasant conceptually, and explains the doubt, expressed earlier, whether multiple constraints are really worth the trouble.



Note that the last validity rule, the Multiple Constraints rule, is conceptually redundant. The general rule that governs the applicability of a feature to a target is the Single-Level Call rule which (combined with the Export rule, both in the [chapter on calls](#)) essentially states that *f* is a valid feature for the call `x.f (...)` if it is a feature of the base type of *x* (and is exported as appropriate). This is in fact the reason why take the trouble to define the base type — always a *Class_or_tuple_type*, with clearly identifiable features — for every kind of type. All that the Multiple Constraints rule states is the application of the Single Call rule to the case of a multiply constrained *Formal_generic_name*, using the just given definition of the base type in this case.

→ “[Class-Level Call rule](#)”, page 636;
“[Export rule](#)”, page 632.

But the Single Call rule in this case is so indirect, relying through the base type on a fictitious class, that compiler writers will likely prefer, for the error message they display in case of a wrong *f*, to cite the Multiple Constraints rule.

Tuples

13.1 OVERVIEW



Based on a bare-bones form of class — with no class names — tuple types provide a concise and elegant solution to a number of issues:

- Writing functions with multiple results, ensuring complete symmetry with multiple arguments.
- Describing sequences of values of heterogeneous types, or “tuples”, such as [*some_integer*, *some_string*, *some_object*], convenient for example as arguments to printing routines.
- Achieving the effect of routines with a variable number of arguments.
- Achieving the effect of generic classes with a variable number of generic parameters.
- Using simple classes, defined by a few attributes and the corresponding assigner commands — similar to the “structures” or “records” of non-O-O languages, but in line with O-O principles — without writing explicit class declarations.
- Making possible the agent mechanism through which you can handle routines as objects and define higher-order routines.

→ Agents are the topic of chapter [27](#).

13.2 TUPLES IN A NUTSHELL

The object-oriented type system of Eiffel is based on classes, possibly equipped with generic parameters. Normally, you will give each class a name and write a class declaration `... class CLASS_NAME ... end`, as seen in previous chapters. But in some simple cases it is convenient to define a class without a name, by merely listing its properties at the place where you need it. Such *anonymous classes* can only have elementary features: a few attributes and the corresponding field-setting procedures. Being anonymous, they require no specific class declaration: you just use a *tuple type* of the form

TUPLE [*x1*: *T1*; *x2*: *T2*; ... ; *xn*: *Tn*]

where the T_i are types. This has the same effect as if you were using a type based on an explicit — non-anonymous — class with n attributes of names x_1, \dots, x_n , of the types given, and n corresponding assigner commands set_x_1, \dots, set_x_n , used to set the field values, with no preconditions. But you don't need to invent a class name or write a class declaration. The class is implicitly defined by the tuple type.

A simple notation exists for describing instances of tuple types:\

```
[v1, v2, ..., vn]
```

where v_1 is of type T_1 and so on. This is known as a **manifest tuple**. Here you don't need the labels (x_1 and so on). In fact you may omit them in the tuple type too, writing it as just `TUPLE [T1; T2; ... Tn]`; all you lose then is the notation $t.x_i$ to access fields of a tuple t (although you can still use $t.item(i)$ which returns an *ANY*), and similarly for modifying fields (use *put*).

The conformance rule for tuple types also ignores the labels: `TUPLE [T, U]` conforms to `TUPLE [T]`; `TUPLE [T, U, V]` conforms to `TUPLE [T, U]`; and so on regardless of the presence of any labels.

Short as it is, the preceding description includes the essential properties of anonymous classes and tuple types; the rest of this chapter gives the details.. On first reading you may move on to the next chapter.



13.3 USING TUPLE TYPES AND TUPLES

The syntax of tuple types is straightforward:



Tuple types

```

Tuple_type  ≙ TUPLE [Tuple_parameter_list]
Tuple_parameter_list ≙ "[" Tuple_parameters "]"
Tuple_parameters ≙ Type_list | Entity_declaration_list

```

← `Tuple_type` is one of the variants of `Type`, introduced page [328](#).

A `Type_list`, as first introduced for actual generic parameters, is a list of types separated by commas or semicolons. `Entity_declaration_list` was defined in the discussion of routines where it served to define the formal arguments of a routine; here is its syntax again:



Entity declarations

```

Entity_declaration_list ≙ {Entity_declaration_group ";" ...}
Entity_declaration_group ≙ Identifier_list Type_mark
Identifier_list ≙ Identifier "," ...}*
Type_mark ≙ ":" Type

```

← This first appeared on page [220](#).

Manifest tuples, denoting tuple values, have the following syntax:\



Manifest tuples

Manifest_tuple \triangleq "[" Expression_list "]"

Expression_list \triangleq {Expression "," ...}*

The syntax for *Tuple_type* permits examples such as:



<i>TUPLE</i>	[1]
<i>TUPLE</i> [<i>INTEGER</i>]	[1]
<i>TUPLE</i> [<i>INTEGER</i> , <i>REAL</i> , <i>POLYGON</i>]	[2]
<i>TUPLE</i> [<i>i</i> , <i>j</i> : <i>INTEGER</i> ; <i>r</i> : <i>REAL</i> ; <i>p</i> : <i>POLYGON</i>]	[3]
<i>TUPLE</i> [<i>i</i> , <i>j</i> : <i>INTEGER</i> ; <i>lr</i> : <i>LIST</i> [<i>REAL</i>]; <i>tpi</i> : <i>TUPLE</i> [<i>p</i> : <i>POLYGON</i> ; <i>i</i> : <i>INTEGER</i>]]	[4]



The type names appearing in brackets after *TUPLE*, if any, are called the **parameters** of the tuple type, by analogy with the actual generic parameters of a generically derived type *C* [*TYPE1*, *TYPE2*]. The optional names attached to individual parameters, such as *i*, *j*, *r* and *p* in example [3], are the **labels** of these parameters.

The syntactical analogy between tuple types and generically derived types is intentional, but note that the two categories are different; *TUPLE* is not a class name but a reserved word of the language.

Example [1] has no parameters. Examples [1] and [2] have parameters but no labels. In the last two examples the parameters are labeled: *i*, *r* and so on. Example [4] shows that a tuple type can have arbitrary constituent types, including another tuple type.

The syntax implies that labels must be either present for all constituent types, as in examples [3] and [4], or absent for all, as in [1] to [2].



A tuple type covers sequences of values, with types conforming to the corresponding parameters. So the above examples have **direct instances** such as, respectively:



[]	-- Empty tuple
[25]	-- Tuple with one integer element
[25, --8.75, <i>pol</i>]	-- With <i>pol</i> of type <i>POLYGON</i>
[25, 32, --8.75, <i>p1</i>]	
[25, --8.75, <i>lr</i> , [<i>p1</i> , 100]]	-- With <i>lr</i> of type <i>LIST</i> [<i>REAL</i>]



The type of example [1], *TUPLE* with no parameters, doesn't seem so useful at first since its only direct instance is the empty tuple []. But throw in conformance and the picture changes. The conformance rule for tuples, previewed below, will state that *TUPLE* [T_1, \dots, T_n] always conforms to *TUPLE* [T_1, \dots, T_m] for $n \geq m$ — labels, if present, playing no part here. So even though the *direct* instances of *TUPLE* [T_1, \dots, T_m] are sequences of exactly m values of the types given, the *instances* of this type include any sequence of m or more values, of which the first m have to conform to these types, the following ones being arbitrary.

→ “CONFORMANCE
==== TO BE REWRIT-
TEN”, 13.5, page 378 be-
low (preview), “TUPLE
TYPE CONFORM-
ANCE”, 14.10, page
396 (full rule).

So [25, —8.75, *pol*], given as a direct instance of *TUPLE* [*INTEGER*, *REAL*, *POLYGON*], is also an instance of *TUPLE* [*INTEGER*, *REAL*], of *TUPLE* [*INTEGER*], and of just *TUPLE*. More generally, every tuple type conforms to *TUPLE*, and every tuple expression, such as the examples above, is an instance of *TUPLE*, although not a direct instance.



There is no specific validity constraint on tuple types, but the Entity Declaration Rule requires all parameter labels, if present, to be different. This doesn't prevent a label from reappearing in the definition of a parameter that is itself a tuple type, the way i reappears in the last parameter of example [4].

←Page 221.

Two notions will help us define this semantics precisely. First, the sequence of types associated with a tuple type:



Type sequence of a tuple type

The **type sequence** of a tuple type is the sequence of types obtained by listing its parameters, if any, in the order in which they appear, every labeled parameter being listed as many times as it has labels.

The type sequence for *TUPLE* is empty; the type sequence for *TUPLE* [*INTEGER*; *REAL*; *POLYGON*] is *INTEGER*, *REAL*, *POLYGON*; the type sequence for *TUPLE* [i, j : *INTEGER*; r : *REAL*; p : *POLYGON*] is *INTEGER*, *INTEGER*, *REAL*, *POLYGON*, where *INTEGER* appears twice because of the two labels i, j .

This enables us to define the associated sequences of values:



Value sequences associated with a tuple type

The **value sequences** associated with a tuple type T are sequences of values, each of the type appearing at the corresponding position in T 's type sequence.

Parameter labels play no role in the semantics of tuples and their conformance properties. They never intervene in tuple expressions (such as `[25, -8.75, pol]`). Their only use is to allow name-based access to tuple fields, as `your_tuple.label`, guaranteeing statically the type of the result.

13.4 ANONYMOUS CLASSES

==== THIS SECTION WILL BE REWRITTEN SHORTLY, IGNORE IT!

To capture the precise semantics of tuple types we must realize that the run-time values they describe, tuples, are not just sequences of values but — like everything else in a system's execution — objects.

Like all other objects, these tuples are instances of classes; the only difference is that you don't have to write these classes. Instead, any tuple type implicitly defines a class, said to be *anonymous*. An anonymous class has the following features:

- *count*: *INTEGER*, the tuple size (number of values in a direct instance of the tuple type).
- *item* (*i*: *INTEGER*): *ANY*, returning the value of the *i*-th item of a tuple, for *i* between 1 and *count*.
- *put* (*x*: *ANY*; *i*: *INTEGER*), to change the value of the *i*-th item, applicable only if *x* denotes a value whose type conforms to the type of the corresponding parameter.
- For any label *l* associated with a parameter *T*, an attribute *l* of type *T*, returning the corresponding tuple item.
- For any such label, an assignment procedure **assign "l"**, to assign values to the corresponding tuple item.

So with the declarations



```
tuple4: TUPLE [i: INTEGER; Ij: NTEGER; r: REAL; p: POLYGON]
c, n: INTEGER
p1, p2, p3: POLYGON
x: SOME_TYPE
```

you can perform the following instructions:

```
tuple4 := [25, -2, r, p1]
c := tuple4.count      -- Assigns value 4 to c
if {p5: POLYGON} tuple4.item (4) then
    -- Assigns value of p1 to p5, but
    -- note need for object test
    -- since item returns an ANY
    p5.polygon_operation
end
tuple4.put (p2, 4)     -- Succeeds in replacing last item by p2
    -- since p2 is of type POLYGON
tuple4.put (x, 4)     -- Violates precondition if SOME_TYPE
    -- does not conform to POLYGON.
n := tuple4.i         -- Assigns value 25 to n
p3 := tuple4.p        -- Assigns last item's value, p2, to
    -- p3; no need for object test
tuple4.p := p4        -- Replaces last item by p4
```

Feature *item* and *put* both use an integer argument *i* indicating the position of the tuple element to be accessed or replaced. Because *i* is variable, these features can use no type more precise than *ANY* as the type of *item*'s result and *put*'s first argument:

```
item (i: INTEGER): ANY
    -- i-th item of tuple
put (v: ANY; i: INTEGER)
    -- Replace i-th element of tuple by v.
```

As a result, any practical use of *ANY* will need to perform an assignment attempt on the result, like the assignment to *poly2* in the third example instruction above.

For an unlabeled parameter, you cannot do any better. But this is where the labels, if present, come in handy: they are treated as attributes with the associated assignment procedures. The last three examples illustrate this. Our example type `TUPLE [i: INTEGER; INTEGER; REAL; p: POLYGON]` is equivalent to a class with four attributes, of which the first and last have been given names *i* and *p*, of respective types `INTEGER` and `POLYGON`. This is how we can access the corresponding tuple items `tuple4.i` and `tuple4.p`, with the exact types — not just `ANY`, and without assignment attempt. Similarly we can perform procedure assignments, such as the last example instruction `tuple4.p |= poly5`, with exact type checking.



Here is the precise definition of the anonymous classes associated with tuple types. (You may skip the rest of this section on first reading.) The only one of these classes that your software can explicitly use is `ANONYMOUS`. Eiffel compilers and tools are not required to build the other classes explicitly; but the way they handle tuple types must be the same as if the classes were actually present.



The definition proceeds by induction on the number of parameters.

Type `TUPLE`, with no parameters, is considered as an abbreviation for `ANONYMOUS`, the name of a Kernel Library class whose features have been listed above: `count`, `item` and so on. The precise specification of the class appears [in the Kernel library chapter](#). → “ANONYMOUS”. A.6.20 CLASS, page 997.

Then type `TUPLE [l1: X1; ...; ln: Xn]`, with $n \geq 1$, is defined by induction as an abbreviation for a class

```
class ANONYMOUSn inherit
    TUPLE [l1: X1; ...; ln-1: Xn-1]
feature -- Access
    ln: Xn assign
invariant
    large_enough: count >= n
end
```

without the `feature` clause if there is no label for the last parameter `Xn`. In that case, as noted, there is no way to access and modify the *n*-th component of a corresponding tuple other than through `item` and `put`, which treat the item as being of type `ANY`.

13.5 CONFORMANCE ===== TO BE REWRITTEN



$TUPLE [l_1: X_1; \dots; l_n: X_n]$, with or without labels, conforms to the following types (the precise wording of the rule will appear in the chapter on conformance): → “TUPLE TYPE CONFORMANCE”, 14.10, page 396 .

- 1 • Any other tuple type with the same parameters X_1, \dots, X_n . (In other words: the labels don't matter at all for conformance.)
- 2 • Any other tuple type $TUPLE [t_1: X_1; \dots; t_m: X_m]$, again with some or all of the labels possibly absent, for $m \leq n$. (So $TUPLE [X]$ conforms to $TUPLE$; $TUPLE [X; Y]$ conforms to $TUPLE [X]$, and so on.)
- 3 • Any array type $ARRAY [T]$ such that every one of the X_i conforms to T . (This allows us to treat tuples as arrays if we want to.)



A mathematical note will be of interest to the curious reader (non-mathematical readers should skip to the following section). The rule in case 2 seems to contradict mathematical intuition. We are used to considering that, if U conforms to T , the instances of U form a subset of the instances of T . Case 2 then doesn't look right if you think of $TUPLE [X, Y]$ as representing the mathematical set $X \times Y$ — the cartesian product of the sets represented by X and Y . The rule states that $TUPLE [X, Y]$ conforms to $TUPLE [X]$, but we do not normally think of $X \times Y$ as a subset of X (or $X \times Y \times Z$ as a subset of $X \times Y$, and so on). The trivial relation is the other way around: X is in one-to-one correspondence with a subset of $X \times Y$ (the subset made of pairs of the form $[x, y0]$ for some arbitrary element $y0$ of Y).

To remove this apparent paradox, it suffices to use another model than cartesian product. Consider $TUPLE [X_1, \dots; X_n]$ as modeling a set T_n of partial functions from \mathbf{N} (the set of natural integers) to the set X of all possible objects. T_n is the set of all such functions f whose domain includes the interval $1, 2, \dots, n$. Any tuple may be viewed as such a function; for example, the tuple $[a, b, c]$ is the function f whose domain only includes the integers $1, 2$ and 3 , such that $f(1) = a, f(2) = b$ and $f(3) = c$.

With this interpretation the subsetting relation and the subtyping (conformance) relation do coincide: $TUPLE [X_1, \dots, X_{n+1}]$ represents the set of functions whose domain includes $1, 2, \dots, n+1$, a superset of $TUPLE [X_1, \dots, X_n]$, containing the functions whose domain includes $1, 2, \dots, n$.

13.6 MULTIPLE RESULTS AND VARIABLE NUMBERS OF ARGUMENTS

Tuple types enable us to remove two limitations of routines:

- A routine has either no result (procedure) or one (function); with tuples we can provide the equivalent of a function with several results.
- A routine has a fixed number of arguments; with tuples we can provide the equivalent of a variable number of arguments.

Emulating multiple results

Tuples allows us to handle routine arguments and routine results in a consistent way. Most programming languages treat these two categories non-symmetrically: a routine has zero or more arguments, but it has either no result if it is a procedure, or, if it is a function, *exactly one* result. To achieve the equivalent of a routine with multiple results you must write a function that returns a — single — result, which happens to be a complex object. This is sometimes inconvenient.


Eiffel has a simple rule. Conceptually, as captured by the definition of “signature of a feature”. *every feature has a single tuple argument and a single tuple result.* This covers all special cases:

← “THE SIGNATURE OF A FEATURE”,
5.13, page 148 .

- Procedure: the result tuple is empty.
- Routine with no argument: the argument tuple is empty.
- Function with multiple results: the result tuple has more than one element.

Without tuples, we would need to address the last case by introducing a class to represent the result type, with attributes representing the components of the result, and procedures to set their values. This technique works and it is justified if the new class covers a valuable abstraction. But if not — if the class is just an artefact — it is needlessly heavy. Tuples give you equivalent benefits without the burden of declaring a new class.

A typical example is a division routine returning both the quotient and the remainder of a number, or other divisible object, by another. You may write its header (for a suitable *NUMBER* type) as



```
divided alias "/" (other: NUMBER):
  TUPLE [quotient: NUMBER; remainder: NUMBER]
```

and, in the body of the routine, have assignments of the form

```
Result.quotient := ...
Result.remainder := ...
```

Then you can use the function with

```
div := number1 / number2
```

and access the two components of the result through *div.quotient* and *div.remainder*.

This symmetric way of handling arguments and results permits a clearer and more concise style.

Note that a manifest tuple is an expression, not a *Variable*; so an assignment of the form $[a, b, c] := [1, 2, 3]$, or here $[a, b, c] := \text{number1} / \text{number2}$, would be syntactically invalid. The target of any assignment must be a single variable, which here will be of a tuple type; this is the case with *Result* and *div* above.

Emulating a variable number of arguments

A routine has a fixed signature, listing a set of arguments with set types. To achieve the effect of a variable number of arguments we may, as previewed in the discussion of routines, use an argument of type *TUPLE*, and call the routine with a tuple of arbitrary length, as in

```
write ([your_integer, your_string, your_real])
```

“USING A VARIABLE NUMBER OF ARGUMENTS”, 8.4, page 221.
The example used there had one more argument.

where the corresponding routine might look like this:

```
write (values: TUPLE)
    -- Print all elements of values, under given format.
local
    n: INTEGER; next: ANY
do
    from n := 1 until n > values.count loop
        next := values.item (n)
        if {i: INTEGER} next then
            write_integer (i)
        elseif {r: REAL} next then
            write_real (r)
        ... Other cases ...
    end
end
```

13.7 TUPLES AS ARRAYS ===== TO BE REWRITTEN

Another application comes from case 3 of the conformance properties of the previous section, which specifies that *TUPLE [TA, TB, ...]* conforms to *ARRAY [T]* if everyone of the types *TA*, *TB*, ... conforms to *T*. This means that you can also treat a manifest tuple as a **manifest array**, enabling you to initialize an array by giving the list of its elements, as in

```
first_primes := [1, 2, 3, 5, 7, 11, 13, 17]
```

with *first_primes* of type *ARRAY [INTEGER]*. Similarly, if *some_routine* takes a formal argument of type *ARRAY [T]*, you can call it as

```
some_routine ([t1, t2, t3])
```

where every *ti* is of a type conforming to *T*.



Conformance

14.1 OVERVIEW

Conformance is the most important characteristic of the Eiffel type system: it determines when a type may be used in lieu of another.

The most obvious use of conformance is to make assignment and argument passing type-safe: for x of type T and y of type V , the instruction $x := y$, and the call *some_routine* (y) with x as formal argument, will only be valid if V is *compatible* with T , meaning that it either *conforms* or *converts* to T . Conformance also governs the validity of many other constructs, as discussed below.

Conformance, as the rest of the type system, relies on inheritance. The basic condition for V to conform to T is straightforward:

- The base class of V must be a descendant of the base class of T .
- If V is a generically derived type, its actual generic parameters must conform to the corresponding ones in T : $B [Y]$ conforms to $A [X]$ only if B conforms to A and Y to X .
- If T is expanded, inheritance is not involved: V can only be T itself.



If this is your first reading, this simple explanation is probably sufficient to understand the references to conformance in the rest of this book, and you may want to move on right away to the next chapter.

A full understanding of conformance requires the formal rules explained below, which take into account the details of the type system: constrained and unconstrained genericity, special rules for predefined arithmetic types, tuple types, anchored types.



The following discussion introduces the various conformance rules of the language as “DEFINITIONS”. Although not validity constraints themselves, these rules play a central role in many of the constraints, so that language processing tools such as compilers may need to refer to them in their error messages. For that reason each rule has a validity code of the form **VNCx**.

These rules appear in the index with other validity codes under “validity constraints”, as well as separately under “conformance rules”.

14.2 CONVERTIBILITY AND COMPATIBILITY

To permit assignment or argument passing, conformance is only one of the two possibilities; the other is **convertibility**, allowing reattachment operations — assignment and argument passing — that convert the source to the type of the target. Convertibility is particularly useful for arithmetic types, allowing us to rely on standard mathematical conventions when assigning, for example, an integer value to a real target.

Often, as in the rules for assignment and argument passing, we must state that the type of an expression *either* conforms or converts to that of an entity. We need a term that covers both mechanisms:

DEFINITION

Compatibility between types

A type is **compatible** with another if it either conforms or converts to it.

It is also useful to extend this notion to expressions, so that we can say “*y* is compatible with *x*” rather than “the type of *y* is compatible with that of *x*”:

DEFINITION

Compatibility between expressions

An expression *b* is **compatible with** an expression *a* if and only if *b* either conforms or converts to *a*.

For conformance we may define the notion now:

Expression conformance

An expression *exp* of type *SOURCE* **conforms to** an expression *ent* of type *TARGET* if and only if they satisfy the following conditions:

- 1 • *SOURCE* conforms to *TARGET*.
- 2 • If *TARGET* is attached, so is *SOURCE*.
- 3 • If *SOURCE* is expanded, its version of the function *cloned* from *ANY* is available to the current class.

So conformance of expressions is more than conformance of their types. Both conditions [2](#) and [3](#) are essential. Condition [2](#) guarantees that execution will never attach a void value to an entity declared of an attached type — a declaration intended precisely to rule out that possibility, so that the entity can be used as target of calls. Condition [3](#) allows us, in the semantics of attachment, to use a cloning operation when attaching an object with “copy semantics”, without causing inconsistencies.

A later definition will state what it means for an expression b to *convert* to another a . As a special case these properties also apply to entities.

Conformance and convertibility are exclusive of each other, so we study the two mechanisms separately. The rest of the present discussion is devoted to conformance. → “*Conversion principle*”, page 408.

14.3 APPLICATIONS OF CONFORMANCE

Conformance governs the validity of many language constructs. For any of the following to be valid, V must conform to T , with x of type T and y of type V : → *In the first two cases (but none of the others) V may also convert to T . See chapter 15.*

- The assignment $x := y$.
- The routine call $r(\dots, y, \dots)$, where x is the formal argument declared in r at the position that y has in the call.
- The creation instruction **create** $\{V\} x \dots$, which creates an instance of V and attaches x to it.
- The redeclaration of x as being of type V in a proper descendant, where x is an attribute, a function, or a routine argument.
- Any use of $C[\dots, V, \dots]$ with V as actual generic parameter, where the corresponding formal generic parameter of C is constrained by T — in other words, the class is declared as $C[\dots, G \rightarrow T, \dots]$.

As these examples indicate, conformance is originally a relation between **types**: the language’s rules specify when a type V conforms to a type T .

The rest of this chapter starts with a generalization of the notion of conformance, originally defined for types, to signatures. The discussion then covers conformance rules for the various kinds of type studied in the last three chapters: class types, first without genericity, then with genericity added to the picture; formal generic parameters; expanded types; tuple types; anchored types (including expression conformance).

14.4 EXPRESSION AND SIGNATURE CONFORMANCE

We have already generalized the notion of conformance from types to *expressions*. Another useful generalization is to *signatures*. A signature gives the full type information for a feature: the types of its arguments, if any, and of its result, if any. Conformance of signatures is important because it governs redeclaration: whenever you redeclare a feature, the signature of the new version must conform to the signature of the original.

← The conformance constraint for signatures is clause 2 of the Redeclaration rule, page 313

The definition of conformance for signatures will follow immediately from the definition for types: a signature t conforms to a signature s if and only if every element of t (the type of an argument or result) conforms to the corresponding element of s .



More precisely, recall that a signature is a pair of sequences of the form

$$[A_1, \dots, A_n], [R]$$

← Signatures were defined in “[THE SIGNATURE OF A FEATURE](#)”, 5.13, page 148.

where all elements involved are types; the A_i are the types of the formal arguments (for a routine) and R is the result type (for a function or an attribute). Either component of the pair, or both, may be empty (the first is empty for an attribute or a routine without arguments; the second, for a procedure). The second component has at most one element, but remember that this element may be a tuple type, so for all practical purposes we can deal with multiple-result functions.

Then from a definition of type conformance, as explored in the rest of this chapter, we immediately infer a definition of signature conformance:



Signature conformance

VNCS

A signature $t = [B_1, \dots, B_n], [S]$ **conforms to** a signature $s = [A_1, \dots, A_n], [R]$ if and only if it satisfies the following conditions:

- 1 • Each of the two components of t has the same number of elements as the corresponding component of s .
- 2 • Each type in each of the two components of t conforms to the corresponding type in the corresponding component of s .
- 3 • Any B_i not identical to the corresponding A_i is detachable.

For a signature to conform: the argument types must conform (for a routine); the two signatures must both have a result type or both not have it (meaning they are both queries, or both procedures); and if there are result types, they must conform.

A “*query*” is a function or attribute, i.e. a feature returning a result.

Condition 3 adds a particular rule for “covariant redefinition” of arguments as defined next.

Covariant argument

In a redeclaration of a routine, a formal argument is **covariant** if its type differs from the type of the corresponding argument in at least one of the parents' versions.

From the preceding signature conformance rule, the type of a covariant argument will have to be declared as *detachable*: you cannot redefine $f(x: T)$ into $f(x: U)$ even if U conforms to T ; you may, however, redefine it to $f(x: ?U)$. This forces the body of the redefined version, when applying to x any feature of f , to ensure that the value is indeed attached to an instance of U by applying an `Object_test`, for example in the form

```
if {x: U} y then y.feature_of_U else ... end
```

This protects the program from *catcalls* — wrongful uses, of a redefined feature, through polymorphism and dynamic binding, to an actual argument of the original, pre-covariant type.

The rule only applies to *arguments*, not results, which do not pose a risk of catcall.

This rule is the reason why the Feature Declaration rule requires that if any routine argument is of an anchored type, that type must be detachable, since anchored declaration is a shorthand for explicit covariance.

14.5 DIRECT AND INDIRECT CONFORMANCE

Conformance is, with one restriction, a reflexive and transitive relation: any type conforms to itself, and if V conforms to U and U to T , then V conforms to T . (The restriction is that T must not be expanded; see below.)

Also, replacing an actual generic parameter by a conforming type yields a conforming type: if Y conforms to X , then $B[Y]$ conforms to $B[X]$ for a class B with one generic parameter; this generalizes to any number of parameters.

We may use these properties to simplify the study of conformance rules. By considering the relation **direct conformance**, which only covers the case of a class conforming to a different one through no intermediary, we can define general conformance by reflexive transitive closure:



General conformance

VNCC

Let T and V be two types. V **conforms to** T if and only if one of the following conditions holds:

- 1 • V and T are identical.
- 2 • V conforms directly to T .
- 3 • V is *NONE* and T is a detachable reference type.
- 4 • V is $B [Y_1, \dots Y_n]$ where B is a generic class, T is $B [X_1, \dots X_n]$, and for every X_i the corresponding Y_i is identical to X_i or, if the corresponding formal parameter does not specify **frozen**, conforms (recursively) to X_i .
- 5 • For some type U (recursively), V conforms to U and U conforms to T .
- 6 • T or V or both are anchored types appearing in the same class C , and the deanchored form of V in C (recursively) conforms to the deanchored form of T .



Cases 1 and 2 are immediate: a type conforms to itself, and direct conformance is a case of conformance.

Case 3 introduces the class *NONE* describing void values for references. ← See “*NONE*”, 6.6, page 175. You may assign such a value to a variable of a reference type not declared as attached (as the role of such declarations is precisely to exclude void values); an expanded target is also excluded since it requires an object.

Case 4 covers the replacement of one or more generic parameters by conforming ones, keeping the same base class: $B [Y]$ conforms to $B [X]$ if Y conforms to X . (This does not yet address conformance to $B [Y_1, \dots Y_n]$ of a type CT based on a class C different from B .) Also note that the **frozen** specification is precisely intended to preclude conformance other than from the given type to itself.

Case 5 is indirect conformance through an intermediate type U .

Finally, case 6 allows us to treat any anchored type, for conformance as for its other properties, as an abbreviation — a “macro” in programmer terminology — for the type of its anchor.

Thanks to this definition of conformance in terms of direct conformance, the remainder of the discussion of conformance only needs to define **direct** conformance rules for the various categories of type.

The general conformance rules follow: for any type T , direct conformance rules will yield the (possibly empty) set ST of types which conform directly to T ; then the types that conform to T are T itself, the members of ST , and, recursively, if T is a reference type, any type conforming to a member of ST .

Before we move on, let's give a name, "conformance path", to the sequence of types appearing implicitly in case [5](#) of the definition. This notion will be useful in particular in the [discussion of repeated inheritance](#):

→ "[THE REPEATED INHERITANCE CONSISTENCY CONSTRAINT](#)", [16.13](#), [page 463](#).



Conformance path

A **conformance path** from a type U to a type T is a sequence of types T_0, T_1, \dots, T_n ($n \geq 1$) such that T_0 is U , T_n is T , and every T_i (for $0 \leq i < n$) conforms to T_{i+1} . This notion also applies to **classes** by considering the associated base classes.

← "[CURRENT TYPE FEATURES OF A TYPE](#)", [12.11](#), [page 265](#)

14.6 CONFORMANCE TO A NON-GENERIC REFERENCE TYPE

Let us begin with the simple but common and important case of conformance to a reference type B obtained directly from a non-generic class. Then direct conformance is essentially inheritance: C conforms directly to B if C is an **heir** of B . (As a consequence, D conforms to B if D is a *descendant* of B .) C (and D) may be generically derived or not.

Assume for example class declarations beginning with



```
class C1 ... inherit A1 ...
class C2 [G] ... inherit A2 ...
expanded class C3 [G, H → HASHABLE] ... inherit A3 ...
```

Then, if X is any type, and Y any type conforming to $HASHABLE$:

- $C1$ conforms directly to $A1$.
- $C2 [X]$ conforms directly to $A2$.
- $C3 [X, Y]$ conforms directly to $A3$.

$C3$ is expanded, $C1$ and $C2$ are not; this has no influence on the discussion. But $A1$, $A2$ and $A3$ must not be expanded. See [14.9](#), [page 394](#), on conformance to expanded types.

These examples assume that all the types involved are **attached** (the default). Indeed if the target type is attached the source type must be attached too; otherwise — in an attachment made valid by the conformance — we could end up assigning a void value to an attached entity

Here then is the rule:



Direct conformance: reference types *VNCN*

A Class_type *CT* of base class *C* **conforms directly** to a reference type *BT* if and only if it satisfies the following conditions:

- 1 • Applying *CT*'s generic substitution to one of the conforming parents of *C* yields *BT*.
- 2 • If *BT* is attached, so is *CT*.



The restriction to a reference type in this rule applies only to the target of the conformance, *BT*. The source, *CT*, may be expanded.

As in the examples.

The basic condition, 1, is inheritance. To handle genericity it applies the “generic substitution” associated with every type: for example with a class *C* [*G*, *H*] inheriting from *D* [*G*], the type *C* [*T*, *U*] has a generic substitution associating *T* to *G* and *U* to *H*. So it conforms to the result of applying that substitution to the Parent *D* [*G*]: the type *D* [*T*].

Condition 2 guarantees that we'll never attach a value of a detachable type — possibly void — to a target declared of an attached type; the purpose of such a declaration is to avoid this very case. The other way around, an attached type may conform to a detachable one.

This rule is the foundation of the conformance mechanism, relying on the inheritance structure as the condition governing attachments and redeclarations. The other rules cover refinements (involving in particular genericity), iterations of the basic rule (as with “general conformance”) and adaptations to special cases (such as expanded types).

14.7 GENERICALLY DERIVED REFERENCE TYPES

---- SECTION TO BE REWRITTEN OR REMOVED ---

The next typing mechanism to take into account is genericity. A generic class such as

```
class B [G, H -> DT, I] ... end
```

Conformance of a generically derived type such as BT to a non-generic one raises no particular problem and is covered by the above rule on non-generic conformance.

raises two kinds of conformance issues. One is the conformance to a generically derived type *BT* based on *B*, of the form *B* [*TK*, *TL*, *TM*]. The other is conformance properties of the formal parameters *G*, *H*, *I* ... themselves, which within the class text represent types. This section deals with the first issue; the next one will cover formal parameters.

When does a type CT conform to BT of the form $B [TK, TL, TM]$? For identical B and C we already have the answer from case 4 of the General Conformance rule: it tells us that in this case CT must be of the form $B [TR, TS, TT]$ with the same number of parameters, with TR conforming to TK , TS to TL and TT to TM . ← Page 388.

Thanks to this first rule, it suffices to examine the case of different base classes, but the same actual generic parameters. Then to check conformance of, say, $C [Y]$ to $B [X]$, you will check separately the conformance of $C [Y]$ to $B [Y]$, using the rule given below, and then show that Y conforms to X , which will complete the deduction thanks to case 4 of General Conformance. See “EXPANDED TYPE CONFORMANCE”, 14.9, page 394 about the expanded case, for which conformance possibilities are very limited.

This is the basic idea for the Generic Substitution rule given below. The full rule is a little more delicate because of all the parameterization involved, but the idea is easy to understand intuitively.

The reason we must be careful in stating the rule is that the two classes involved, here C and B , may have different formal generic parameters: different in role, number or both. For example, given the above declaration for B and



```
class C [P → DT, Q] inherit
    B [TK, P, TM]
    ...
end
```

The constraining type DT plays no role in this example.

we will want the type CT defined as

```
C [TL, TN]
```

to conform to the type BT defined above as

```
B [TK, TL, TM]
```

even though the number of generic parameters is different for each class. Why CT should conform to BT is intuitively clear: if we interpret the text of C for the actual generic parameters TL , corresponding to P , and TN , corresponding to Q , the Parent $B [TK, P, TM]$ listed in its Inheritance clause really stands for $B [TK, TL, TM]$, which is precisely BT .

On first reading, if you find this example sufficient to give an intuitive understanding of conformance in such cases, you may wish to skip to the next section.



As the example shows, we will need to use substitutions (of actual to formal generic parameters) to ascertain direct conformance rigorously. If $\{x1, \dots, xn\}$ and $\{y1, \dots, yn\}$ are sets with the same number of elements, a substitution from the first set to the second is a one-to-one correspondence between them, associating a different element of the second to every element of the first. For example, a substitution σ (among six possible ones) from $\{T, U, V\}$ to $\{L, M, N\}$ is given by the table

σ maps:	to:
T	M
U	N
V	L

The number of possible substitutions between two sets of n elements is the factorial of n , here $3! = 6$.

----The notion of substitution serves to specify actual-formal correspondence rigorously.

As in the non-generic case, you can **disable** conformance of CT to BT even in the presence an inheritance link by using **non-conforming inheritance**.

← *“NON-CONFORMING INHERITANCE”*, 6.8, page 180. The observation for non-generic reference types was in 14.6, page 389.

To see that the rule is in fact easy to apply, let us use it to check that the type CT defined in the above example as $C [TL, TN]$ indeed conforms to BT , defined as $B [TK, TL, TM]$. The assumption is that B is declared as $B [G, H, I]$, with three formal parameters, and that $C [P, Q]$, with two formal parameters, lists $B [TK, P, TM]$ as **Parent**.

The application is straightforward. Here $n = 3$ and $m = 2$; the types and **Formal generic** names appearing in the definition are:

$X1 : TK$ $X2 : TL$ $X3 : TM$
 $G1 : G$ $G2 : H$ $G3 : I$
 $Y1 : TL$ $Y2 : TN$
 $H1 : P$ $H2 : Q$
 $Z1 : TK$ $Z2 : P$ $Z3 : TM$

The substitution σ associates $Y1$ to $H1$ and $Y2$ to $H2$. In other words, it defines the associations

σ maps:	to:
P	TL
Q	TN

and leaves other elements unchanged. So applying σ to the Z_j yields

σ maps:	to:	Comments
Z_1	TK	σ leaves Z_1 , i.e. TK , unchanged.

Z_2	TL	This is the result of applying σ to Z_2 , i.e. P .
Z_3	TM	σ leaves Z_3 , i.e. TM , unchanged.

The three resulting types TK , TL and TM are indeed, respectively, $X1$, $X2$ and $X3$, showing that $C [TL, TN]$ does conform directly to $B [TK, TL, TM]$.

To show that $C [TL, TN]$ conforms to $B [SK, SL, SM]$ if TK conforms to SK , TL to SL and TM to SM , we would first use the Generic Substitution rule, as was just done, to show conformance to $B [TK, TL, TM]$, and then apply case 4 of the General Conformance rule to obtain the required actual generic parameters.

14.8 FORMAL GENERIC PARAMETER CONFORMANCE

The next case is a `Formal_generic_name` type: a formal generic parameter to the enclosing class. In the text of a generic class

```
class C [G, H -> CT,...] ... end
```

you may use the formal generic parameters G and H as types. G and H illustrate the two kinds of generic parameters, placing different requirements on the types to be used as the corresponding actual generic parameters: G , unconstrained, stands for arbitrary types; H , constrained by CT , stands for types that conform to CT .



As noted in the discussion of genericity, the base type of a constrained `Formal_generic_name` such as H is the constraining type, here CT . An unconstrained generic `Formal_generic_name` such as G is considered to be constrained by the universal class ANY , which serves as its base type.

In both cases, the `Formal_generic_name` will conform directly to its constraining type (CT or ANY). In the reverse direction, however, no direct conformance is possible: if we allowed assigning to an entity of type G or H an expression of a different type, we would have no way of guaranteeing that this type is compatible for every possible actual generic parameter.

The rule for conformance to and from generic parameters follows from these observations:



Direct conformance: formal generic $VNCF$

Let G be a formal generic parameter of a class C , which in the text of C may be used as a `Formal_generic_name` type. Then:

- 1 • No type **conforms directly** to G .
- 2 • G **conforms directly** to every type listed in its constraint, and to no other type.

This assumes a single constraint. Multiple constraints are addressed below.

←12.3, page 351 presented unconstrained, and 12.6, page 354 constrained, genericity. 12.10, page 363 addressed the use of generic parameters as types, defining their base type on page 364 (page 369 for the multi-constraint case).



Remember from the general definition of conformance that every type conforms (not directly) to itself.

The last clause of the rule mentions “one or more” constraining types. This is because we allow more than one constraint, as in

```
class D [G -> {CONST1, CONST2, CONST3}] ...
```

where *G* will conform to every one of *CONST1*, *CONST2*, *CONST3*. As noted in the discussion of genericity, this occurs only rarely; most uses of constrained genericity limit themselves to one constraint as with *C* above.

← *Case 1* of *General Conformance* rule, page 388.
 ← “*THE CASE OF MULTIPLE CONSTRAINTS*”, 12.13, page 367.



In the case of recursive generic constraints, as in

```
class C [G, H -> ARRAY [G]] ...
```

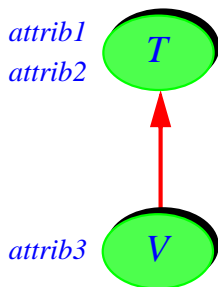
the rule is applicable without any need for a special clause: within the text of *C*, *H* represents a type that only conforms directly to *ARRAY [G]*. This corresponds to the property, ensured by the Constrained Genericity rule, that in a generic derivation *C [T, U]* the type *U* must conform to *ARRAY [T]*.

← “*RECURSIVE GENERIC CONSTRAINTS*”, 12.9, page 362.

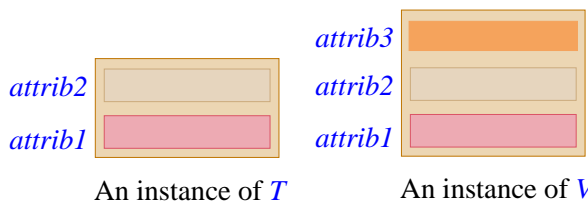
← Page 359

14.9 EXPANDED TYPE CONFORMANCE

----- TO BE REWRITTEN -----



Two classes



An instance of *T*

An instance of *V*

Typical instances

Cannot copy bigger object onto smaller one.

What about the reverse direction — conformance of an expanded type *ET* to a reference type *RT*? Here there is no implementation constraint since it is always physically possible to reattach a reference to an object of arbitrary size. But of course the attachment must be compatible with the type system: the base type of *ET* must conform to *RT*.

	(Case 1)	(Case 2)
BEFORE	y <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-left: 20px;"> (OBJ1) with ref semantics </div> (Case 1)	y <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-left: 20px;"> (OBJ1) with copy semantics </div> (Case 2)
AFTER	x y <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-left: 20px;"> (OBJ1) with ref semantics </div>	x <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-left: 20px;"> (OBJ2) with copy semantics </div> y <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-left: 20px;"> (OBJ1) with copy semantics </div>

These are the essential ideas. Now the details.



First let's review the two forms of expanded type. Examples of the first are ---- REMOVED -----

Examples of the second case are



```
A
B [X, Y, Z]
```

where *A* and *B* have been declared as **expanded class** (*A* non-generic, *B* generic). These types are their own base types; the base classes are *A* and *B*. The basic types *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL* and *POINTER* fall into this category.

---- REMOVED -----



Direct conformance: expanded types *VNCE*

No type **conforms directly** to an expanded type.

→ .

From the definition of general conformance, an expanded type *ET* still *conforms*, of course, to itself. *ET* may also conform to reference types as allowed by the corresponding rule (*VNCN*); the corresponding assignments will use copy semantics. But no other type (except, per General Conformance, for *e* of type *ET*, the type **like e**, an abbreviation for *ET*) conforms to *ET*.



This rule might seem to preclude mixed-type operations of the kind widely accepted for basic types, such as *f* (3) where the routine *f* has a formal argument of type *REAL*, or *your_integer_64 := your_integer_16* with a target of type *INTEGER_64* and a source of type *INTEGER_16*. Such attachments, however, involve **conversion** from one type to another. What makes them valid is not conformance but **convertibility**, which does support a broad range of safe mixed-type assignments.

→ Chapter 15.

14.10 TUPLE TYPE CONFORMANCE

Next consider tuple types. A tuple type is of the form

```
TUPLE [label_1: T1; ...; label_n: Tn]
```

where: the part in square brackets may be absent, giving the most general tuple type, *TUPLE*; the T_i , if present, are types, called the “**parameters**” of the tuple type; and any of the label parts *label_i*: may be absent. In fact, the labels will play no role in the conformance rules: *TUPLE* [*A*], *TUPLE* [*x*: *A*] and *TUPLE* [*y*: *A*] are all equivalent for conformance purposes. We saw that the only role of the labels is to define assignable attributes in the corresponding types.

← Chapter 13 overed tuples.

There are two sources of conformance for tuple types:

- A tuple type conforms to any other having the same initial sequence of parameters T_1, \dots, T_n , regardless of the labels present or not in either one.
- In addition, a special rule relates tuples to arrays. If we have a tuple expression, especially in the form of a manifest tuple [*x1*, ..., *xn*], it is useful to treat it as an array. This will provide the equivalent of manifest arrays — arrays defined by a list of their items — and is permitted by a rule stating that a tuple type conforms to *ARRAY* [*T*] if all of its parameters conform to *T*.

The conformance rule for tuple types follows



Direct conformance: tuple types VNCT

A *Tuple_type* *U*, of type sequence *us*, **conforms directly** to a type *T* if and only if *T* satisfies the following conditions:

- 1 • *T* is a tuple type, of type sequence *ts*.
- 2 • The length of *us* is greater than or equal to the length of *ts*.
- 3 • For every element *X* of *ts*, the corresponding element of *us* conforms to *X*.

No type conforms directly to a tuple type except as implied by these conditions.

Labels, if present, play no part in the conformance.

----FOLLOWING NOT TRUE, REPLACE BY DISCUSSION OF CONVERTIBILITY ---- allows tuples to be treated as arrays. So if every one of the types of *x1*, *x2*, ..., *xn* conforms to *T*:

- You can write an assignment $a := [x1, x2, \dots, xn]$ where *a* is of type *ARRAY* [*T*]. This provides a simple means of array initialization, as in $ia := [1, 2, 3]$ for *ia* of type *ARRAY* [*INTEGER*].
- You can write a call *some_routine* ($[x1, x2, \dots, xn]$) where the corresponding formal argument in *some_routine* is of type *ARRAY* [*T*].

14.11 ANCHORED TYPE CONFORMANCE



Anchored types greatly simplify, as you will remember, the management of groups of entities that must keep the same type in redeclarations. An anchored type is of the form

← “[ANCHORED TYPES](#)”, [11.10](#), page [339](#).

`like anchor`

where *anchor* is the name of an attribute, function or formal argument, or *Current*. Such a declaration describes a type which is the same as the type of *anchor* but will automatically follow any redefinition of the type of *anchor* in a proper descendant. Using *Current* as anchor means that the type will be the current type (class name with generic parameters if any).

← “[CURRENT TYPE, FEATURES OF A TYPE](#)”, [12.11](#), page [365](#).

There is no need for a special conformance rule, as the last clause of the General Conformance rule already told us that, for the purpose of conformance, we should simply look at the “deanchored form” of an anchored type.

Convertibility

15.1 OVERVIEW

Complementing the *conformance* mechanism of the previous discussion, convertibility lets you perform assignment and argument passing in cases where conformance does not hold but you still want the operation to succeed after adapting the source value to the target type.

15.2 WHY IMPLICIT CONVERSION?



Even if you haven't heard of convertibility before, you probably benefited from it indirectly, even in elementary applications, if you ever mixed different arithmetic types. Convertibility is indeed what allows you to write an assignment of the form `your_real := 10` where the target is of type *REAL*, or a call such as `routine_expectng_a_real (10)` where the routine has a formal argument of type *REAL*. In such cases you will expect the integer to be converted to its *REAL* equivalent.

Most programming languages handle such cases through *ad hoc* rules applying to a fixed set of arithmetic types. But there is no reason to deprive programmers from applying this mechanism to their own types. It's not just a matter of consistency and generality but, as the examples in this chapter should demonstrate, expressiveness and simplicity

Why indeed stop at *INTEGER* and *REAL*? Thanks to convertibility you can define a class *COMPLEX* that makes the assignment `c := 10` valid for `c` of type *COMPLEX*, with the effect of calling a conversion query to assign to `c` the complex number `[10.0, 0.0]`. Similarly, you can write graphics classes that permit assignments and argument passing between objects representing images in PNG, JPEG, GIF and other graphics formats.



You can always in principle do without the conversion mechanism, by explicitly calling routines to perform the needed transformations. The semantics clearly defines conversion, in all cases, as an **abbreviation** for such an explicit form: `c := 10` is a shorthand for **create** `c.from_integer (10)` where class `COMPLEX` has a creation procedure `from_integer` that has been marked as a *conversion procedure*. In many cases, you may prefer to stay away from the implicit mechanism, forcing the explicit form because you feel it reflects more directly what is actually happening. Yet in some cases the same criteria of expressiveness, clarity and simplicity favor implicit conversions. This applies in particular when you frequently go back and forth between two types that map one-to-one into each other's values; an example, detailed below, is the use of Eiffel on the Microsoft .NET environment, which has its own form of string. To pass strings back and forth between Eiffel and non-Eiffel routines would require, without the conversion mechanism, polluting the code with countless calls to `from_dotnet` and `to_dotnet` routines. Thanks to the conversion specifications in class `STRING`, you can instead use strings the way you would in pure Eiffel, and let conversions happen automatically as needed.

In other cases, you will have to decide which is preferable: the explicitness of routine calls, with the attendant verbosity; or the terseness of implicit conversion. When you choose the latter, the validity rules ensure that this cannot cause any ambiguity or surprise.

15.3 CONVERSION BASICS AND EXAMPLES

Here is the basic way to make conversions work. In the class representing conversion targets, you may add after the **Creators** part, which lists creation procedures, a **Converters** part specifying that some of them are **conversion procedures**, each with the types from which it will convert:



```
expanded class REAL_64 inherit ... create
    from_integer, from_real_32, ... Other creation procedures...
convert
    from_integer ({INTEGER}),
    from_real_32 ({REAL_32})
feature -- Initialization
    from_integer (n: INTEGER)
        -- Initialize by converting from n.
    do
        ... Conversion algorithm ...
    end
    ... Similarly for from_real_32 ...
... Rest of class omitted ...
end
```

This suggests some basic terminology:



Conversion procedure, conversion type

A procedure whose name appears in a **Converters** clause is a **conversion procedure**.

A type listed in a **Converters** clause is a **conversion type**.

We'll say that the conversion types, here *INTEGER* and *REAL_32*, **convert** to the current type, here *REAL_64*. (Fuller definition [below](#).) The validity rules imply that every conversion procedure is also a creation procedure. → [Page 415](#).

By listing conversion procedures you permit assignments and argument passing from the given conversion types to the current type, even though there is no conformance relation between them. With *REAL_64* as shown the following assignments will be valid:

```
your_real_64 := your_integer
your_real_64 := your_real_32
```

As before this assumes your_integer of type INTEGER and so on.

with exactly the same effect, respectively, as

```
create your_real_64.from_integer (your_integer)
create your_real_64.from_real_32 (your_real_32)
```

Here you may omit the **create** in both cases since *REAL_64* is expanded, but you would need it for a reference target type.

Similarly, you may use calls of the form

```
routine_expectng_a_real_64 (your_integer)
routine_expectng_a_real_64 (your_real_32)
```

with the guarantee that the arguments will be converted to *REAL_64*, using *from_integer* in the first case and *from_real_32* in the second case.

This is the property that justifies mixed-type arithmetic expressions of the form

```
your_real + your_integer
```

since that notation is really, [as you know](#), a shortcut for a function call

```
your_real.plus (your_integer)
```

*Similarly: your_real + your_real_32.
← "OPERATOR FEATURES", 5.15, page 154; also "THE EQUIVALENT DOT FORM", 28.8, page 780.*

where the function *plus alias* "+" from *REAL_64* has a formal argument of type *REAL_64*; when we pass an *INTEGER* as in the above assignments, a conversion will occur. So everything — well, almost everything — falls into place: you can continue using arithmetic expressions as you know them, mixing types if you wish, while respecting the type rules.

Why “almost” everything? Because the rule does not deal with such variants as *your_integer + your_real*: interpreted as a shortcut for the function call *your_integer.plus (your_real)* it doesn't work, since *plus* from *INTEGER* expects an integer argument and cannot take a *REAL*. In such cases we need the companion technique of **target conversion** presented below, allowing us to start by converting the target, *your_integer*, to *REAL*, and then use the "+" function from *REAL*.

→ “*MIXED-TYPE EXPRESSIONS: TARGET CONVERSION*”, 15.12, page 428..

As a non-arithmetic example of conversion, a class *DATE* could start



```
class DATE create
  from_tuple, ...
convert
  from_tuple ({TUPLE [NATURAL, STRING, NATURAL]})
...
```

allowing you to initialize a date entity from a [*Day, Month, Year*] tuple:

```
your_date := [1, "January", 2000]
```

As before, this is really just an abbreviation for *create your_date .from_tuple ([1, "January", 2000])*, but may be convenient, especially if you move from assignment to argument passing as in

```
compute_revenue ([1, "January", 2006], [1, "January", 2007])
```

where the routine expects two *DATE* arguments. We can express this without conversions, through creation expressions, but the result is noticeably heavier:

```
compute_revenue (create {DATE} .from_tuple ([1, "January", 2006]),
  create {DATE} .from_tuple ([1, "January", 2007]))
```

Some people may still prefer this second form since it makes it clear that two new objects are being created. Others — including me, at least for such examples — enjoy the simplicity and concision afforded by conversion. The design of the mechanism, as detailed below, makes sure that no surprise or ambiguity can arise.

No ambiguity can arise because *INTEGER* and *REAL_32* **do not conform** to *REAL_64*, and *TUPLE [INTEGER, INTEGER, INTEGER]* does not conform to *DATE*. So when you see

```
your_real_64 := your_integer
```


or the assignment of a tuple to *your_date*, you know immediately they can only work through conversion. This is not just a feature of these examples but a key principle of convertibility, ensuring clarity and safety.

→ See below "[CONVERSION PRINCIPLES](#)", 15.6, page 408.

15.4 CONVERSION QUERIES

The preceding examples illustrate the basic mechanism to specify that a type *U* converts to a type *T*: in the base class of *T*, include a conversion procedure allowing conversion **from** *U*. In principle, this covers all cases. But a practical consideration suggests that we also need a conceptually equivalent facility that works in the reverse direction, specifying conversions **to** *T*. Instead of a conversion procedure this variant will use a conversion query. The reason we need it is that sometimes *U* is one of your types but *T* is someone else's, which you can't modify, even just to add a conversion procedure. All you can work on is *U*.

When you have access to both *T* and *U*, the two mechanisms — a conversion procedure from *U* in *T*, or a conversion query to *T* in *U* — you may use either mechanism for equivalent effect, but in keeping with general goals of clarity and simplicity the validity rules will ensure that you use only one.

As a typical example of the need for conversion queries consider type *OTHER_STRING* denoting strings from some other language or environment, which can be mapped to Eiffel strings but have their own internal representation. (This is indeed the case with *DOTNET_STRING* in the version of EiffelStudio for Microsoft .NET, representing native .NET strings.) In the Eiffel Kernel Library class *STRING* we can provide a conversion procedure as well as a function going the other way:

```
from_other (s: OTHER_STRING)  
    -- Initialize from s  
  
to_other: OTHER_STRING  
    -- Representation in "other" form of current string
```

Assuming that *eiffel_routine* expects an Eiffel *STRING* and *external_routine* expects an *OTHER_STRING* we could — without taking advantage of the conversion mechanism yet — satisfy their requirements through explicit transformations:

```
eiffel_string: STRING
external_string: OTHER_STRING
...
eiffel_routine (create s.from_other (external_string) ) [1]
external_routine (s.to_other) [2]
```

Now assume that we prefer the conversions to remain implicit (as is indeed desirable in the .NET case, since a typical system using Eiffel for .NET might have thousands of such back-and-forth transformations between the Eiffel and .NET formats, and the repeated application of the above style would clutter the code, damaging its readability by detracting from its more important aspects). To allow replacing the first call [1] by just

```
eiffel_routine (external_string) [3]
```

it suffices, as illustrated in the previous section, to mark *from_other* as a conversion procedure. But to allow replacing [2] by

```
external_routine (s) [4]
```

you would need, if restricted to this technique, to add a conversion procedure to *OTHER_STRING*. This won't work if (as in the .NET case) it's an external class over which you have no control.

You may in such cases work from the other side — conversion source, rather than target — by marking *to_other* as a conversion query. The syntax is predictable:



```
class STRING create
  from_other, ...
convert
  from_other ({OTHER_STRING})
  to_other: {OTHER_STRING}
...
```

This allows [1]; in fact the effect is exactly the same as with the addition of a creation *procedure* to the other class:

```
class OTHER_STRING create
  from_string, ...
convert
  from_string ({STRING})
...
```

but it works even if you can't touch that other class.



The different syntax for listing the types:

- `from_string` (`{STRING}`) for a creation procedure;
- `to_other` : `{STRING}` for a creation function

reflects the role of the corresponding values: as procedure argument in the first case, as function result in the second. This is the reason in the procedure case for using both parentheses and braces.

With conversion queries we must extend the preceding terminology:



Conversion query, conversion feature

A query whose name appears in a `Converters` clause is a **conversion query**.

A feature that is either a conversion procedure or a conversion query is a **conversion feature**.



The definition of *conversion type*, introduced in the context of conversion procedures, also covers result types of conversion queries. ← Page 401..

Another useful piece of terminology, for two types T and U (non-generically-derived) of base classes CT and CU : we say that U **converts to** T , and also that T **converts from** U , if either:

- CT has a conversion procedure listing U as conversion type.
- CU has a conversion query listing T as conversion type.

→ For the full definition, integrating genericity, see below "[Converting to and from a type](#)", page 415..

These two cases are exclusive: were they to hold for the same T and U , conversions would be ambiguous, as they could apply either the procedure or the function. As we'll see shortly, the validity rules prohibit this, removing any such confusion.

15.5 USING CONVERSIONS PROPERLY



As noted at the beginning of this chapter you never *have* to include a conversion facility in a class: whenever you feel you might be creating confusion in the minds of the clients' authors minds, you may provide instead a creation procedure without making it a conversion procedure, or a function without making it a conversion query. Indeed, in any of the preceding examples, instead of

```
target := source
```

you may use one of the explicit forms

```
create target.conversion_procedure (source)  
target := source.conversion_query
```

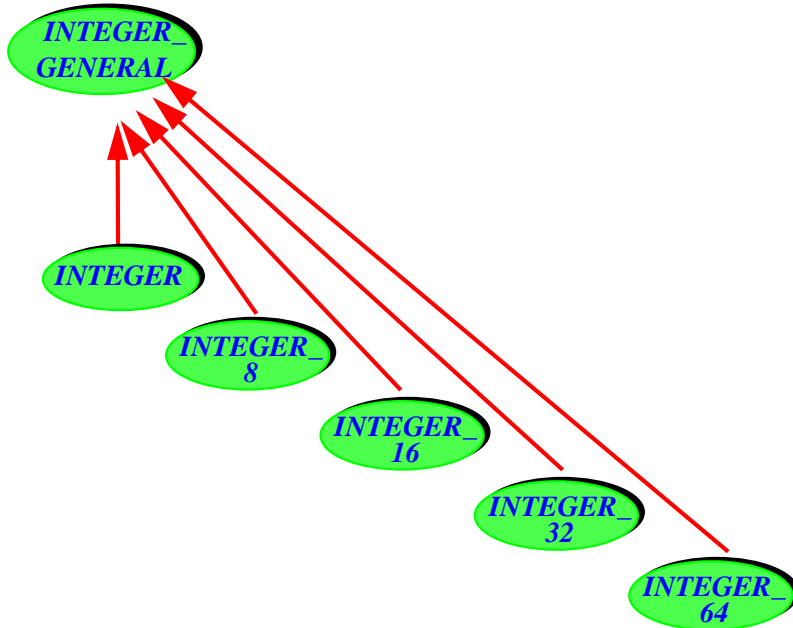
You may also choose a hybrid policy: make the creation procedure a conversion procedure but for some types only, so that the first form is valid only if the type of *source* is one of those; the second form, with an explicit **Creation_instruction**, remains available for other types, including types which may conform or convert to the conversion types (an example involving integer types appears next).

The general advice — a methodological principle, not a language rule — is that a creation procedure should provide a conversion mechanism only if the associated operation does not entail any loss of information. For example it is OK to convert an integer to a 64-bit real silently through an assignment *your_real_64 := your_integer*, as this can be implemented without any loss of information. In the other direction, however, you must choose between various forms of truncation and rounding. For that reason **INTEGER** is listed as a convertible type in class **REAL** but not conversely. To convert a real source to an integer target you have to use a creation procedure as above, or something like *your_integer := your_real_64.truncated*.

Converting **INTEGER** to **REAL** may cause loss of information, but tradition, perhaps misguided, suggests permitting assignment in this case.



Sized *INTEGER* variants provides an interesting example. The relevant part of the inheritance hierarchy is



Integer type inheritance hierarchy

→ Extracted for the figure of page 818, also covering *REAL* and *CHARACTER* variants.

INTEGER_GENERAL is the general notion; the other provide variants for specific sizes: 8-bit integers and so on (*INTEGER* is 32 bits).

In class *INTEGER_GENERAL* you will find the claused

```

create
  from_integer, ... Other creation procedures ...
convert
  from_integer ({INTEGER, INTEGER_8, INTEGER_16,
                INTEGER_32, INTEGER_64})
feature -- Initialization
  from_integer (n: INTEGER_GENERAL)
    -- Set from n
    do ... Conversion algorithm ... end
  ... Rest of class omitted ...
  
```

From class *INTEGER_GENERAL*

Since *INTEGER_GENERAL* describes integers with an arbitrary precision, it is safe to allow conversion from any of the sized integer types. But when a sized variant is used as target, reattachments from *bigger* integers would cause a loss of information (typically they may keep the least significant digits). So we will only allow conversion from smaller integers; for example class *INTEGER* has the clauses

```

create
    from_integer, ... Other creation procedures ...
convert
    from_integer ({INTEGER_8, INTEGER_16})
  
```

From class *INTEGER*
(32-bit).

As always with creation procedures, the creation status of a procedure in a class is independent from the status it had in proper ancestors. So *INTEGER* is free to specify the creation status of *from_integer* differently from *INTEGER_GENERAL*.

→ “*CREATORS AND INHERITANCE*”,
20.5, page 534.

So a reattachment such as *your_integer := your_integer_8* is valid but not *your_integer := your_integer_64*. Note, however, that *from_integer* remains a creation procedure in *INTEGER*, and can take any argument of type *INTEGER_GENERAL*, including an *INTEGER_64* which it will in this case truncate. (The feature’s header comment mentions this, as header comments always should in such cases.) So you can still write

```

your_integer.from_integer (your_integer_64)
  
```

(with a **create** if you wish). This is a good policy in such cases: making an conversion operation into a creation procedure, applicable to a wide range of types, including some for which it may lose information; but specify as conversion types only those for which there is no information loss.

15.6 CONVERSION PRINCIPLES



PURPOSE

To ensure the clarity and safety of the mechanism, it is essential to rule out any potential ambiguity. Three fundamental properties ensure this. They are validity properties, but without codes of their own since they simply follow from the validity rules introduced in the next section.



VALIDITY

Conversion principle

No type may both conform and convert to another.

Conversion Asymmetry principle

No type *T* may convert to another through both a conversion procedure and a conversion query.

Conversion Non-Transitivity principle

That V converts to U and U to T does not imply that V converts to T .

These principles avoids any surprise for software authors and readers.

As a result of the first principle, when you read

```
x := y
```

or a corresponding argument passing, for x and y of types TX and TY with base classes CX and CY , you see immediately, which of conformance and convertibility applies:

- If y conforms to x — because TX and TY are the same, or there's direct inheritance between their base classes — this is a plain attachment, with no conversion involved.
- If the class texts specify that TY converts to TX , the attachment will involve is a conversion

If neither of these cases holds, the attachment is invalid. If either holds, the other doesn't, as a result of the Conversion principle. So if an attachment will cause a conversion, it's **always** because the types don't conform; since reattachment otherwise requires conformance, you cannot — when reading the text — miss the anomaly, immediately alerting you to the presence of a conversion.

Eiffel tools can provide visual feedback to highlight that conversion; EiffelStudio uses tooltips for this purpose:

*EiffelStudio:
tooltip
highlighting a
conversion*

Whereas the first principle removes conflicts between conformance and convertibility, the second one makes sure that a conversion through a procedure can't compete with a conversion through a query.

The third principle follows from the observation that convertibility of a type to another is always the result of an explicit **convert** clause as illustrated by the preceding examples. Conformance, for its part, is usually transitive: from conformance of V to U and U to T it generally follows that V conforms to T , without any special mention in their base classes. That is not the case for convertibility: all mentions must be explicit. The main reason is that conversion involves a transformation of values, which should always be explicit. With transitivity, additional transformations would occur behind the scene.

← As per "[General conformance](#)", page 388.

When you do want multiple conversions, a specific expression syntax is available: with tI , uI , vI of respective types T , U , V you may not, if V doesn't conform or convert to T , use $tI := vI$, but as detailed later in this chapter you may (assuming again that V converts to U and U to T) write

← "[CONVERTING AN EXPRESSION EXPLICITLY](#)", 15.9, page 416.

$tI := \{U\} vI$

where the expression, using a **manifest type** $\{U\}$, denotes vI converted to U . This makes the assignment valid since U convert to T . For even more explicitness you may write $\{T\} \{U\} vI$ (making the second conversion explicit too); but this is not necessary since an attachment may always use *one* implicit conversion.

15.7 CONVERSION SYNTAX AND VALIDITY

The syntax construct covering the mechanism as seen is **Converters**, one of the clauses of a **Class_declaration**:

← Page 119.



Converter clauses
Converters \triangleq convert Converter_list
Converter_list \triangleq {Converter ", "...} ⁺
Converter \triangleq Conversion_procedure Conversion_query
Conversion_procedure \triangleq Feature_name "(" "{" Type_list "}" "
Conversion_query \triangleq Feature_name ":" "{" Type_list "}" "

A validity rule covers **Conversion_procedure** and another, very similar, covers **Conversion_query**. Here is the first:



Conversion Procedure rule VYCP

A Conversion_procedure listing a Feature_name *fn* and appearing in a class *C* with current type *CT* is valid if and only if it satisfies the following conditions, applicable to every type *SOURCE* listed in its Type_list:

- 1 • *fn* is the name of a creation procedure *cp* of *C*.
- 2 • If *C* is not generic, *SOURCE* does not conform to *CT*.
- 3 • If *C* is generic, *SOURCE* does not conform to the type obtained from *CT* by replacing every formal generic parameter by its constraint.
- 4 • *SOURCE*'s base class is different from the base class of any other conversion type listed for a Conversion_procedure in the Converters clause of *C*.
- 5 • The specification of the base class of *SOURCE* does not list a conversion query specifying a type of base class *C*.
- 6 • *cp* has exactly one formal argument, of a type *ARG*.
- 7 • *SOURCE* conforms to *ARG*.
- 8 • *SOURCE* involves no anchored type.

Conditions 2 and 3 (the second one covering generic classes) express the crucial requirement, ensuring the Conversion principle: no type that conforms to the current type may convert to it.



In many practical uses of conversion the target class *CX* is expanded; this is the case with *REAL_64*, and with *REAL_32*, to which *INTEGER* also converts. Such cases satisfy condition 2 almost automatically since essentially no other type conforms to an expanded type. But the validity of a conversion specification does not require the enclosing class to be expanded; all that condition 2 states is that the conversion types must not conform to it (more precisely, to the current type).

← “EXPANDED TYPE CONFORMANCE”, 14.9, page 394.

Even if *CX* is not expanded you can still let *CY* be an heir of *CX* and satisfy the rule: simply use **non-conforming inheritance**. Under this form of inheritance *CY* has access to the features of *CY*, and retains its invariant

← This technique was described in “NON-CONFORMING INHERITANCE”, 6.8, page 180.

Clause 3 is the same as clause 2 but applied to generic classes. In this case the type to which *SOURCE* must not conform is, rather than the current type, a type obtained from it by replacing every generic parameter by its constraining type (with the usual convention that an unconstrained parameter is understood as constrained by *ANY*). Why this clause? In *C [G → CONSTRAINT]* we want to preclude specifying conversion from a type *C [U]* if *U* conforms to *CONSTRAINT*. This would create an

← “SEMANTICS OF GENERIC TYPES”, 12.10, page 363.

ambiguity for $x := y$ with x of type $C [T]$ and y of type $C [U]$ where U conforms to T , and hence to $CONSTRAINT$: we would have both convertibility and conformance, violating the Conversion principle. It is not enough here, as in clause 2, to prohibit conversion types from conforming to the current type, here $C [G]$: they must avoid conformance to $C [CONSTRAINT]$.

Clause 4 further removes possible ambiguity by making sure that if a class has two or more conversion procedures, as in

```
class C create
    cp1, cp2, cp3, , ...
convert
    cp1 ({T1, U1}), cp2 ({T2}), cp3 ({T3, U3})
...
```

all the conversion types involved — $T1, U1, T2, T3, U3$ — are different. So when you see $x := y$ where x is of type C you know which one of the procedures will be called: the type TY of y must be one of the conversion types, and the assignment will use the corresponding creation procedure ($cp1$ if T is $T1$ or $U1$ and so on).

To get rid of the last possible source of potential ambiguity and ensure the Conversion Asymmetry principle, clause 5 guarantees that convertibility “from” a type U , as specified in C , doesn’t conflict with convertibility “to” C specified in T . More precisely we prohibit U from including a conversion query whose result type is *based on* C . Again this is because of the generic case: with $C [G]$, constrained or not, specifying a conversion from U , we prohibit U from specifying a conversion to $C [T]$ for any T . A less restrictive policy is possible, but it would have to affect the rule on attachment, and we don’t want to break its simplicity — y can be attached to x as long as it conforms or converts to it — for the sake of an unlikely case. (Remember that in principle it would suffice to limit the language to *either* conversion procedures or conversion queries; the only compelling reason for having both is to manage conversions to or from classes we don’t control, as in the .NET case. But then we don’t expect, in those existing classes, to find a conversion to or from *our* own new types!)



We only require the conversion types to be different. It’s OK if some of them conform or convert to others: we look at the exact type of y and apply the associated procedure. The Conversion Non-Transitivity principle tells us that convertibility — unlike conformance — is never indirect. If the type TY of y is not one of the conversion types, but conforms to one of them, say TZ , then convertibility of TZ to TX does **not** make $x := y$ valid. Conversion attachment requires an exact match to the conversion type.

Clause [6](#) requires every conversion procedure to have exactly one argument, so that we can interpret an assignment $x := y$ as a creation call **create** $x.cp(y)$.

Clause [7](#) make such an equivalence valid by ensuring that an argument such as y — of type TY if the conversion procedure was listed as $cp(\{TY\})$ — will be a valid argument to cp . The exact wording is that TY must be **compatible** with the formal argument type ARG of cp , meaning that it either conforms or converts to ARG ; if it converts to ARG , passing y to cp will imply another conversion. But that's all: executing $x := y$ (or, once again, the corresponding argument passing) may cause at most two conversions. one to convert y to ARG if needed, another to convert the result to TX , the type of x . There is no danger of multiple behind-the-scenes conversions, or circularity.

← “Compatibility between types”, page 384.

The rule just discussed covers conversion procedures. Here is its companion for conversion queries:



Conversion Query rule

VYCQ

A Conversion_query listing a Feature_name fn and appearing in a class C with current type CT is valid if and only if it satisfies the following conditions, applicable to every type $TARGET$ listed in its Type_list:

- 1 • fn is the name of a query f of C .
- 2 • If C is not generic, CT does not conform to $TARGET$.
- 3 • If C is generic, the type obtained from CT by replacing every formal generic parameter by its constraint does not conform to $TARGET$.
- 4 • $TARGET$'s base class is different from the base class of any other conversion type listed for a Conversion_query in the Converters clause of C .
- 5 • The specification of the base class of $TARGET$ does not list a conversion procedure specifying a type of base class C .
- 6 • f has no formal argument.
- 7 • The result type of f conforms to $TARGET$.
- 8 • $TARGET$ involves no anchored type.

Condition [5](#) is redundant with condition [5](#) of the Conversion Procedure rule but is included anyway for symmetry. In case of violation, a compiler may refer to either rule.



Neither rule addresses the issue of *preconditions* in conversion features. In general, we don't want any: an assignment $x := y$, or an argument passing, should not be subject to a precondition, even if it involves a conversion. This requirement is addressed a few sections down through the notion of "converting an expression".

These rules also enable us to give a precise definition of the principal notion of this chapter, already used informally: that a type may "convert to" another. First we define what it mean for a type to convert to a *class*:



Converting to a class

A type T of base class CT **converts to** a class C if either:

- The deanchored form of T appears as conversion type for a procedure in the **Converters** clause of C .
- A type based on C appears as conversion type for a query in the **Converters** clause of CT .

The preceding validity rules imply that the two cases are exclusive. In



```
class D [G, H] create
    cv1, cv2, cv3...
convert
    cv1 ({E1}),
    cv2 ({E2 [G, H]},
    cv3 {TUPLE [H, INTEGER]
...

```

the types that convert to class C are $E1$, $E2 [G, H]$ and $TUPLE [H, INTEGER]$.

We must generalize this to the notion that a type converts to another *type*. For a non-generic class such as *REAL_64* it's the same, but with genericity and tuples we need a small refinement involving the notion of substitution, as with the conformance rule for generically derived types in the discussion of conformance. With D as shown above, the types converting to $D [X, Y]$ should be $E1$, $E2 [X, Y]$, $E3 [X]$ and $TUPLE [U, INTEGER]$. Hence the definition:

← "GENERALLY DERIVED REFERENCE TYPES", 14.7, page 390..



Converting to and from a type

A type U of base class D **converts to** a Class_type T of base class C if and only if either:

- 1 • The deanchored form of U is the result of applying the generic substitution of the deanchored form of T to a conversion type for a procedure cp appearing in the Converters clause of C .
- 2 • The deanchored form of T is the result of applying the generic substitution of the deanchored form of U to a conversion type for a query cq appearing in the Converters clause of D .

A Class_type T **converts from** a type U if and only if U converts to T .



T must be a class type: the target type of a conversion may not be anchored, or a formal generic parameter. It must be of the form C for a non-generic class C , or $D [T, U]$ for a generic class D .

Since here are two ways — procedure and function — of specifying convertibility, it's convenient to give them names:



Converting “through”

A type U that converts to a type T :

- 1 • Converts to T **through a procedure** cp if case 1 of the definition of “converting to a type” applies.
- 2 • Converts to T **through a query** cq if case 2 of the definition applies.

These terms also apply to “**converting from**” specifications.

From the definitions and validity rules, it's clear that if U converts to T then it's either — but not both — “through a procedure” or “through a query”, and that exactly one routine, cp or f , meets the criteria in each case.

15.8 SEMANTICS OF CONVERSION

The specification of *when* to perform a conversion belongs to the semantics of reattachment operations (assignment and argument passing) and expressions. Here we should see *what* a conversion will do. We define the “conversion attachment” operation — not a construct of the language but a purely semantic notion, useful in defining the semantics of reattachment:



Conversion semantics

Given an expression e of type U and a variable x of type T , where U converts to T , the effect of a **conversion attachment** of source e and target x is the same as the effect of either:

- 1 • If U converts to T through a procedure cp : the creation instruction **create** $x.cp(e)$.
- 2 • If U converts to T through a query cq : the assignment $x := e.cq$.

This is an “unfolded form” specification expressing the semantics of an operation (conversion attachment) in terms of another: either a creation or a query call. Both of these operations involve an attachment (argument passing or assignment) and so may trigger one other conversion.

← “TWO-TIER DEFINITION AND UNFOLDED FORMS”, 2.11, page 100

15.9 CONVERTING AN EXPRESSION EXPLICITLY

When a type U converts to a type T , conversions will usually happen implicitly as a result of a reattachment, as with $var := exp$ with var of type T and exp of type U . In addition, you can in some cases take advantage of a form of expression that explicitly implies a conversion. There is no need for a language mechanism to achieve this: a simple library feature does the job.

Warning: the word “type”, in diverse shapes and colors and occasionally several times in a row, occurs a lot in this section!

$TYPE$ is a generic class of the Kernel Library, such that an instance of $TYPE [T]$, for any type T , is a “type object” describing the properties of some type conforming to it. (This is a “reflection” mechanism, giving programs access, at run time, to some of their own properties.)

You can write manifest constants representing types: the notation

```
{T}
```

is a Manifest type, of type $TYPE [T]$; it denotes a type object representing T . → =====

In the declaration of $TYPE [G]$ we find the following innocent-looking function:

```
adapted alias "[ ]" (x: G) : G
```

Owhich for any value of type G returns a result of the same type. Not only of the same type, actually, but representing the same value. It can indeed be implemented just like this:

```

adapted alias "[]" (x:  $G$ ) :  $G$ 
    -- Value of x, adapted if necessary to type  $G$ 
    -- through conformance or conversion
do Result := x end

```

What then is the purpose? Think conversions (and conformance) and you'll see: in a call

```

your_type.adapted (y) [5]

```

Note the recommended form: see next.

where *your_type* is of type $TYPE [T]$, representing some type conforming to T , *y* must consequently be of type T — or of a type U compatible with T . The feature will then return the value of *x*, but adapted to type T through the process of attaching the actual argument *x* of type U to the formal *x* of type T . If U conforms to T the result is its original value; but in the case of convertibility the routine will perform a conversion.

As an example using a *Manifest_type*, you may write

```

(|{ $T$ }|).adapted (y) [6]

```

Note the recommended form: see next.

where the target, a non-basic expression, must be parenthesized by $(|\dots|)$. But there's a more convenient form: the function *adapted* is declared with a "bracket alias": *adapted* **alias** "[]"; this means that — like for the *item* feature of arrays, allowing *your_array* [*x*] as a synonym for *your_array.item* (*x*) — it can be used in bracket notation, yielding just

```

{ $T$ } [y] [7]

```

a simple and clear notation meaning: "the T version of *y*". Thanks to the signature of the function *adapted*, this is valid if and only if the type of *y* is compatible with T .

Let us give this a name:

Explicit conversion

The Kernel Library class *TYPE* [*G*] provides a function

adapted alias "[*I*]" (*x*: *G*): *G*

which can be used for any type *T* and any expression *exp* of a type *U* compatible with *T* to produce a *T* version of *exp*, written

{*T*} [*exp*]

If *U* converts to *T*, this expression denotes the result of converting *exp* to *T*, and is called an **explicit conversion**.

Explicit conversion involves no new language mechanism, simply a feature of a Kernel Library class and the notion of bracket alias.

For example, assuming a tuple type that converts to *DATE*, you may use

```
{DATE} [[20, "April", 2005]]
```

The *Basic_expression* [20, "April", 2005] is a *Manifest_tuple*. Giving it — through the outermost brackets — as argument to the *adapted* function of {*DATE*} turns it into an expression of type *DATE*. This is permitted for example if class *DATE* specifies a conversion procedure from *TUPLE* [*INTEGER*, *STRING*, *INTEGER*].

← As in the example of page 402.

In most usual cases of conversion you do not need to specify such a *Manifest_type*; the earlier example showed a routine call of the form

```
compute_revenue ([1, "January", 2006], [1, "January", 2007])
```

where *compute_revenue* expects two *DATE* arguments; this is valid with the proper conversion procedure from *TUPLE* [*NATURAL*, *STRING*, *NATURAL*] to *DATE*. There is no need in this case to specify a *Manifest_type*, although it wouldn't hurt. The need arises because conversion is **not transitive**. Assume a routine

```
process_date_string (s: STRING)
```

← "*Conversion Non-Transitivity principle*", page 409.

which expects a string representing a date in short string format, such as "01.01.2008"; also assume that *DATE* converts to *STRING*, with a conversion feature that produces such a string from a *DATE* object. With *deadline*: *DATE* you could use

```
deadline := [1, "January", 2008]
process_date_string (deadline)
```


where the call converts *deadline* to a *STRING* as desired. But you can't directly use

```
process_string_date ([1, "January", 2008])
```

Warning: not valid under given assumptions.

because this would require *two* conversions: tuple to *DATE* and *DATE* to *STRING*. An actual argument to a routine call must conform or convert to the corresponding formal argument. Conformance is transitive but convertibility is not because, as noted earlier, multiple implicit conversions would cause confusion for the program reader. You may still, in such a case, avoid the use of an intermediate variable — *deadline* in the example — by specifying a *Manifest_type* explicitly:

```
process_date_string ( {DATE} [[1, "January", 2008]])
```

Now valid.

The argument now results from a conversion and is of type *DATE* rather than *TUPLE [INTEGER, STRING, INTEGER]*; then we can pass it to the routine *process_string_date* which expects a *STRING*; this triggers a second conversion. At most one conversion in such a scheme is implicit.

You may specify several conversions in this style, as in

```
{T} [{U} of_type_V] -- Type as implied by the name
```

which yields an expression of type *T*, assuming *V* converts to *U*. Although such an accumulation of conversions is seldom needed, you can use it to make the presence of conversions more explicit, rewriting for example the previous example as

```
process_string_date ( {STRING} {DATE} [1, "January", 2008])
```

Now valid.

Using one or more such manifest types is useful not only when *V* that **converts** to *U*, as in the preceding examples, but also sometimes when it **conforms** to *U*. In an assignment

```
of_type_U := of_type_V
```

(with types as implied by the names), or the corresponding argument passing, you need not prefix *of_type_V* by *{U}*, although again it wouldn't hurt; but if *U* converts to another type *T*, while it is OK to write

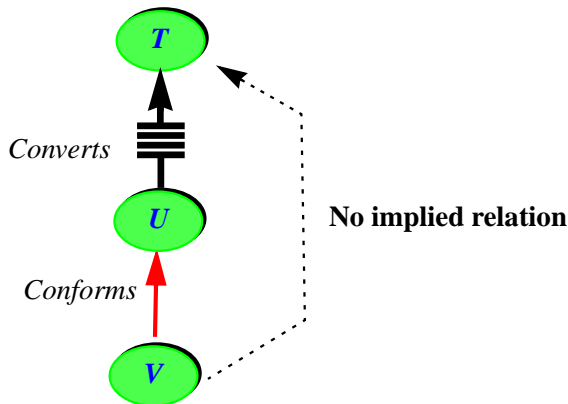
```
of_type_T := of_type_U
```

(causing a conversion), you may not directly use

```
of_type_T := of_type_V
```

Warning: not valid under given assumptions.

if U does not itself convert (or conform) to X . The following figure describes the situation:



No indirect compatibility

Here too you could use an intermediate variable, but it's simpler just to write

```
of_type_T := {U} [of_type_V]
```

Now valid.

Because V conforms to U , the right-hand-side expression actually has the same value as just of_type_V — no conversion is involved — but its type, U rather than V , makes the assignment and associated conversion valid. To be fully explicit you might again include two manifest types:

```
of_type_T := {T} [{U} [of_type_V]]
```



This mechanism resembles in some respects the *casts* available in C and other languages. A major difference is that a C cast forces a type on an expression, compatible or not; if not, it can result in serious run-time errors. Here the validity rules imply that the type **always** make sense for the expression, since the types are compatible. This is a type-safe mechanism, then, which can cause no such errors.

15.10 EXPRESSION CONVERTIBILITY: THE ROLE OF PRECONDITIONS

We've seen the basic notion of convertibility for types: a type U converts, or not, to a type T . Since we will use conversions for attachments, it's convenient to extend it to the notion of an *expression* converting to an *entity*, as we did with conformance. The basic idea is the same: an expression of type U converts to an entity of type T if and only if U converts to T ; but there is a little twist, arising from the possible presence of *preconditions* in conversion procedures and functions.

← "*Compatibility between expressions*", page 384.

In general, it's a bad idea to have a precondition in a conversion feature. When you see an assignment or argument passing

```

x := y
p (y)                                -- With x as the formal argument

```

[8]

with *x* of type *TX* and *y* of type *TY*, you expect that at run time it will always, without conditions, attach *y* to *x*. This should still be the case if the attachment causes a conversion. After all, if you do expect restrictions, you can explicitly use a function or procedure, as in

```

create x.from_TY (y)
x := y.to_TX

```

[9]

where the procedure *from_TY* and the function *to_TX* may have a precondition, which the caller should then check, rewriting for example as

```

if y.is_acceptable then
    create x.from_TY (y)
end

```

[10]

In this case *from_TY* and *to_TX* should not be conversion features; then *TY* does not convert to *TX*, and the implicit form [8] is not valid.

These observations suggest that for clarity and reliability we should prohibit a conversion feature from having a precondition. This is the basic rule, but we need to be a bit more tolerant for conversions whose conditions can be checked *statically* — by the compiler — so that no unpleasant effect can result at run time. Two important examples illustrate the necessity of this extra flexibility.

The first example concerns constants of numeric types, in particular integer types (*NATURAL*, *INTEGER*, *NATURAL_X*, *INTEGER_X* for *X* = 8, 16, 32, 64). With entity types as suggested by the names, you may use the following assignments:

```

of_type_NATURAL_32 := of_type_NATURAL_8
of_type_INTEGER_32 := of_type_INTEGER_8
-- etc.

```

This is all valid thanks to the presence of appropriate conversion features: *NATURAL_8* conforms to *NATURAL_32*, *INTEGER_8* to *INTEGER_32* and so on. More generally, you may expect any conversion between basic types to be available when it does not involve any loss of information (as well as in conversions from *NATURAL_* and *INTEGER_* types to *REAL_* types, for which the loss of information is traditionally considered acceptable). The other way around you'll get a validity error, as in

```
of_type_NATURAL_8 := of_type_NATURAL_32      [11]
of_type_INTEGER_8 := of_type_INTEGER_32
```

WARNING: not valid.

This is as it should be. We could enforce this behavior by providing conversion features from “lighter” numeric types to “heavier” ones (considering *NATURAL_m* lighter than *NATURAL_n* for $m < n$ and so on), but not the other way around.

The only problem with this policy is that it would also prevent us from performing attachments where the source is a numeric **constant**, as in

```
of_type_NATURAL_8 := 1                        [12]
```

`1` is very reasonable as a constant of type *NATURAL_8* — but it's not of that type! Every value in Eiffel must have a well-defined type, and the type of the constant `1` is *INTEGER*. Sure, we could in the end get what we want:

- We may use a constant with an explicit type: `{NATURAL_8} 1` (not to be confused with a conversion). → [“FORCING A TYPE ON A CONSTANT”](#), 29.3, page 789
- We may use some transformation function, as in `to_natural_8(1)`.

The first alternative is more attractive than the second one, but if you have to do extensive manipulations of special-size values, for example many *NATURAL_8* in some system-level application, you'll quickly get tired of having to write the type explicitly. Sometimes a cigar is just a cigar, and sometimes one — including *NATURAL_8* one — is just one.

Another alternative, in the end equivalent to the first but with a more compact syntax, would be for the language to provide specific syntax for constants of sized types, such as *NATURAL_8*; something like `8n1`, `16n1` etc. But with all the variants available this would cause an explosion of special notations.

The conversion mechanism looks the savior here: if we could convert from *NATURAL* to *NATURAL_8*, then we could interpret the assignment [12] as a conversion, making it valid. Formally using a conversion need not affect run-time performance: with a reasonably smart compiler, the instruction can directly assign the 8-bit representation of `1` to the target `of_type_NATURAL_8`.

Such a conversion-based solution assumes a conversion procedure or function from *NATURAL* to *NATURAL_8*. Not all integer values can be converted, however — only those between 0 and 255. The conversion feature should express this through a precondition, reading (assuming a procedure, of argument *x*: *INTEGER*):

```
require
  not_too_small: x >= 0
  not_too_large: x <= 255
```

This is where we run into conflict with the prior observation that conversion features should not have preconditions, and indeed we still do **not** want to permit [11]

```
of_type_NATURAL_8 := of_type_NATURAL_32
```

WARNING: not valid.

where the right-hand-side is variable. But if it is a constant, as in [12]

```
of_type_NATURAL_8 := 1
```

we can require that the precondition be checked **statically** — by the compiler, which should reject the attachment if it can't prove the precondition will always hold at run time.

This explains why we need a more flexible version of the rule as sketched. The first impulse was to prohibit preconditions altogether in conversion features. The more tolerant attitude will accept a precondition *only if it can be enforced at compile time*.

The rule follows directly from this discussion, but let's look first at the other major example (besides constants of sized numeric types). Assuming that *t1*, *t2* and *t3* are all of type *T*, the following **Manifest_tuple**

```
[t1, t2, t3] [13]
```

is of type *TUPLE* [*T*, *T*, *T*]. Now assume that we want a **manifest array** of type *ARRAY* [*T*], given by its items, for example these three. How do we denote it? Rather than introducing a special notation for manifest arrays, it's better to reuse [13], converted from the tuple type to *ARRAY* [*T*]; but such a conversion can only be defined for tuples in general, and is only possible if the types of all the elements in a tuple conform to the element type for the desired array, here *T*. We can express this through a precondition in the conversion procedure. Once again, this would make it inappropriate to apply the conversion to a variable source, as in

```
your_tuple: TUPLE ; your_array: ARRAY [T]
...
your_array := your_tuple
```

since *your_tuple* might or might not satisfy the precondition at run time. But with a manifest tuple, as in

```
your_array := [t1, t2, t3]
```

it is clear at compile time whether the assignment will always work (if and only if all the types conform to *T*). This is another example where we need to accept a conversion feature with a precondition, as long as it is possible to ascertain the precondition statically.

The constraint applies not directly to conversions, but to attachments. The relevant rules are the Attachment rule and, for argument passing, the Argument rule: both state that we may attach *y* to *x*, through assignment or argument passing, if and only if *y* conforms or converts to *x*. In the conversion case, this would normally mean that the type of *y* converts to the type of *x*, but we make the rule a bit more sophisticated to integrate the present discussion:

→ “Assignment rule”, page 590; “Argument rule”, page 634.



Expression convertibility VYEC

An expression *exp* of type *U* **converts to** an entity *ent* of type *T* if and only if *U* converts to *T* through a conversion feature *conv* satisfying either of the following two conditions:

- 1 • *conv* is precondition-free.
- 2 • *exp* statically satisfies the precondition.

← For the counterpart of this rule for conformance, see “Compatibility between expressions”, page 384

This definition has a validity code even though it is a definition rather than a validity rule. This allows compilers, if they find an attachment or argument passing that would use a conversion, but the source doesn't convert to the target, to produce a precise error message referring to this definition, rather than a general one indicating violation of the Assignment or Argument rule.

In case 1, the routine has no precondition, or a precondition guaranteed always to be true; “precondition-free” is defined precisely below. Case 2 provides the extra flexibility resulting from the preceding discussion. It relies on the following definition:

Statically satisfied precondition

A feature precondition is **statically satisfied** if it satisfies any of the following conditions:

- 1 • It applies to a boolean, character, integer or real expression involving only constants, states that the expression equals a specific constant value or (in the last three cases) belongs to a specified interval, and holds for that value or interval.
- 2 • It applies to the type of an expression, states that it must be one of a specified set of types, and holds for that type.

The two cases cover the two examples discussed.



The “constants” of the expression can be manifest constants, or they can be constant actual arguments to a routine — possibly the unfolded form of an assignment, as in `of_type_NATURAL_8 := 1`, whose semantics is that of `create of_type_natural.from_INTEGER (1)`. Without the notion of “statically satisfied precondition” such instructions would be invalid because `from_INTEGER` in class `NATURAL_8` has a precondition (not every integer is representable as a `NATURAL_8`), and arbitrary preconditions are not permitted for conversion features. This would condemn us to the tedium of writing `{NATURAL_8} 1` and the like for every such case, and would be regrettable since `1` is as a matter of fact acceptable as a `NATURAL_8`. So the definition of expression convertibility permits a “statically satisfied” precondition, making such cases valid.

It would be possible to generalize the definition by making permissible any precondition that can be assessed statically. But this would leave too much initiative to individual compilers: a “smarter” compiler might accept a precondition that another rejects, leading to incompatibilities. It was judged preferable to limit the rule to the two cases known to be important in practice; if others appear in the future, the rule will be extended.

The definition of expression convertibility also refers to “precondition-free” routines. The precise definition of this term accounts for the possibility of overriding the original precondition in the case of an inherited routine:



Precondition-free routine VYPF

A feature r of a class C is **precondition-free** if it is either:

- 1 • **Immediate** in C , with either no **Precondition** clause or one consisting of a single **Assertion_clause** (introduced by **require**) whose **Boolean_expression** is the constant **True**.
- 2 • **Inherited**, and such that every **precursor** of r is (recursively) precondition-free, or r is **redeclared** in C with a **Precondition** consisting of a single **Assertion_clause** (introduced by **require else**) whose **Boolean_expression** is the constant **True**.

A definition, but with a validity code for the same reasons as VYEC above.

A feature is “**immediate**” if it is declared in the class itself. In the other case, “**inherited**” feature, it’s OK if the feature had a precondition in the parent, but then the class must redeclare it with a clause **require else True**. A simple **require** without the **else** is **not permitted** in this case.

← “*Inherited, immediate; origin: redeclaration; introduce*”, page 133.

← Clause 3 of “*Redeclaration rule*”, page 313.

A “**precursor**” of an inherited routine is its version in a parent; there may be more than one as a result of feature merging and repeated inheritance.

→ “*Precursor*”, page 473.

15.11 MULTIPLE CONVERSION TYPES

As noted it is possible to associate two or more conversion types with a given conversion procedure, as with *cp1* ($\{T1, U1\}$) in the last example sketch. Every one of them must conform or convert to the type of the procedure’s argument per clause 7 of the Conversion Procedure rule.

As an example, assume that class *REAL* looks like this:



```
expanded class REAL inherit ... create
create
```

```
    from_integer, ...
```

```
convert
```

```
    from_integer ({INTEGER}), ...
```

```
feature -- Initialization
```

```
    from_integer (n: INTEGER)
```

```
        -- Initialize by converting from n.
```

```
        do
```

```
            ... Conversion algorithm ...
```

```
        end
```

```
    ... Rest of class omitted ...
```

```
end
```


This means a call *routine_expectng_a_real (10)* is valid and will convert its argument. Class *REAL_64* as given earlier was of the form

```

note
    version: 1
expanded class REAL_64 inherit ... create
    from_integer, from_real_32, ...

convert
    from_integer convert ({INTEGER}),
    from_real_32 convert ({REAL_32}), ...

feature -- Initialization
    from_integer (n: INTEGER)
        -- Initialize by converting from n.
        do ... Conversion algorithm ... end
    from_real_32 (r: REAL_32)
        -- Initialize by converting from r.
        do ... Conversion algorithm ... end
    ... Rest of class omitted ...

end

```

but because *REAL_32* itself has the ability to convert an integer we may omit *from_integer* and limit ourselves to a single conversion procedure applicable to two conversion types:

```

note
    version: 2
expanded class REAL_64 inherit ... create
    from_real_32, ...

convert
    from_real_32 ({REAL_32, INTEGER}), ...

feature -- Initialization
    from_real_32 (r: REAL_32) is ... As before ...
    ... Rest of class omitted ...

end

```

It is indeed valid to list *INTEGER* as one of the *Conversion_types* associated with *from_real_32*, since this procedure takes a *REAL_32* argument and *INTEGER* converts to *REAL_32* (with the version of *REAL_32* given above, through its conversion procedure *from_integer*).

So even though version 2 of *REAL_64* has shed the conversion procedure *from_integer* present in version 1, the following assignment remains valid since *INTEGER*, the type of 10, converts to *REAL_64*:

```
your_real_64 := 10
```

There is a difference of semantics, however: in version 1, this assignment was considered a shortcut for

```
create your_real_64.from_integer (10)
```

whereas with version 2 it means

```
create your_real_64.from_real_32 (10)
```

still valid because *INTEGER* converts to *REAL_64*, but implying one more conversion, since it really stands for

```
create your_real_64.from_real_32
      (create {REAL_64}.from_integer (10))
```

As noted earlier, the game stops here: an assignment may involve no conversion (if the target conforms to the source), one (as in the case of *your_real_64 := your_real_32*) or two (as in the last example), but no more.

In this example the choice between versions 1 and 2 of *REAL_64* is not critical since you may expect a good compiler to know about basic types and implement the conversions *directly*, without routine calls. In your own classes, however, the extra conversion may be worth avoiding.

Because of accuracy issues, the numerical effect of such an operation may be different under versions 1 and 2.

→ Based on the specifications of the Kernel Library in appendix A.

15.12 MIXED-TYPE EXPRESSIONS: TARGET CONVERSION

Conversions participate in one more mechanism, *target conversion*, designed to reconcile the object-oriented type system of Eiffel with the syntactical conventions of arithmetic expressions. It is only of direct interest to advanced users and library designers, but if you are a novice you should still read the beginning of this section to be reassured that when you write mixed-type arithmetic expressions such as *your_integer + your_real*, with the first operand of type *INTEGER* and the second of type *REAL*, you are actually working within the rules of the type system.

Like the *Manifest_type* facility of the last section, target conversion applies to the *Expression* construct, but its semantics is closely related to the other topics of this chapter.

Using target conversion

We know already, from the discussion in this chapter, is to treat an expression such as *your_real + your_integer*: it is a shortcut for a call

```
your_real.plus (your_integer)
```

where *plus* alias "+" is a function of class *REAL*. As long as we know how to make this function accept arguments of type *INTEGER* and convert them to type *REAL* this works fine.

The more delicate case is *your_integer + your_real*, because here we don't want to use *plus* of class *INTEGER*, which expects an *INTEGER* argument and can in no way handle a *REAL*. What we really want — more precisely, what mathematical tradition tells us we should expect — is for this expression to mean something like

```
(my_integer.converted_to_real) + my_real
```

in other words: convert the first operand (the target of the call) to *REAL* so that can use *plus* alias "+" from class *REAL*, which expects a *REAL* argument and can hence take *my_real*.



The target conversion mechanism makes this possible. This is really all you need to know on first reading: that a conversion mechanism exists, permitting you to write mixed-type arithmetic expressions respecting the long-accepted traditions of mathematics, without departing from the O-O type rules of Eiffel. The Kernel Library classes defining basic arithmetic classes — *INTEGER*, *REAL* and others — use this mechanism as needed to support the common arithmetic cases; authors of new library classes can rely on it as well. Reassured that such common practices have received official O-O blessing and that the Kernel Library classes support them, you may on first reading skip the rest of this chapter.

The target conversion mechanism is supported syntactically by the optional **convert** mark of an [Alias](#) for an infix feature:

← Page [151](#).

Operator aliases

```
Alias  $\triangleq$  'alias "' Alias_name '"' [convert]
```

An example is the following function declaration in class *INTEGER*:

```
plus alias "+" convert (other: INTEGER): INTEGER
    -- Sum of current integer and other
do
    ... Implementation of integer addition ...
end
```

The highlighted **convert** mark means: if the type of the argument you get is not the normally expected argument type, here *INTEGER*, but one to which the current type converts, such as *REAL* and its variants:



- 1 • Convert the target of the call (the first operand of the operation, here *your_integer*) to the applicable conversion type, here *REAL*; the validity rule below ensures that a single, unambiguous conversion procedure is always available.
- 2 • **Ignore the implementation given here** — the part that reads ... *Implementation of integer addition* ... above — as it is irrelevant.
- 3 • Instead, call the function with the same name from the applicable conversion type, here *plus alias "+"* from *REAL*.

So with *your_integer + your_real* the result will indeed fulfill the intent expressed above as *(your_integer.converted_to_real) + your_real*.

Validity of target conversion



No specific validity rule limits our ability to include a **convert** mark in an *Alias* as long as it applies to a feature with one argument and an *Operator* alias. Of course in an example such as *your_integer + your_real* we expect the argument type *AT*, here *REAL*, to include a feature with the given operator, here *+*, and the target type *CT*, here *INTEGER*, to convert to *AT*. But we don't require this of *all* types to which *CT* converts, because:

- This would have to be checked for every new type (since *CT* may convert to *AT* not only through its own “from” specification but also through a “to” specification in *AT*).
- In any case, it would be too restrictive: *INTEGER* may well convert to a certain type *AT* for which we don't use target conversion.

Instead, the validity constraints will simply rule out individual *calls* that would require target conversion if the proper conditions are not met. For example if *REAL* did not have a function specifying *alias "+"* and accepting an integer argument, or if *INTEGER* did not convert to *REAL*, the expression would be invalid.

Remarkably, there is no need for any special validity rule to enforce these properties. All we'll need is the definition of **target-converted form of a binary expression** in the discussion of expressions. The target-converted form of $x + y$ (or a similar expression for any other binary operator) is $x + y$ itself unless *both* of the following properties hold:

- The declaration of “+” for the type of x specifies **convert**.

→ Page 771.

- The type of y does **not** conform or convert to the type of the argument of the associated function, here *plus*, so that the usual interpretation of the expression as shorthand for $x.plus(y)$ cannot possibly be valid. This is critical since we don't want any ambiguity: either the usual interpretation or the targeted conversion should be valid, but not both.

Under these conditions the targeted-converted form is $(\{TY\} [x]) + y$, using as first operand the result of converting x to the type TY of y . Then:

- The target-converted form is only valid if TY has a feature with the “+” alias, and y is acceptable as an argument of this call. The beauty of this is that we don't need any new validity rule: if any of this condition is not met, the normal validity rules on expressions (involving, through the notion of Equivalent Dot Form, the rules on calls) will make it illegal. → Page 780.
- We don't need any specific semantic rule either: the normal semantic rules, applied to the target-converted form, yield exactly what we need.

Target conversion: a discussion



The Target Conversion mechanism deserves some justification. If at first it sounds “ad hoc”, that's because it is. Its purpose is to support traditional conventions of mathematical notation, in particular mixed-type arithmetic expressions. People manipulating integers, reals and the like are used to mixing them freely in expressions, with the understanding that any “lighter” operand will automatically be converted to the “heavier” operand.

← With a notion of “weight” as sketched in 15.10, e.g. INTEGER_8 lighter than INTEGER_16 etc.

Given that this is the usual expectation for numerical computation, four approaches are possible in an object-oriented language:

- 1 • **Ignore the issue:** Treat arithmetic types as completely different from O-O types defined through classes. Being outside the normal type system, arithmetic types don't need to observe O-O rules and can retain whatever properties they had in languages such as Fortran, Pascal or C.
- 2 • **Insist on purity:** Tell programmers that mixed-type expressions are “wrong” and that they should use explicit conversions, perhaps in the style illustrated above as *(my_integer.converted_to_real) + my_real*.
- 3 • **Provide special rules for basic types:** Keep integers, reals and such within the O-O type system, but introduce special cases in the conformance and reattachment rules to support the traditional properties of arithmetic expressions, including automatic conversion. → Chapter 15; “CONVERSIONS”, 22.6. page 591.
- 4 • **Provide a general conversion mechanism:** As described in this section and complementary ones in other chapters.

Solution [1](#) is used in such languages as Java and C++. Its drawbacks are obvious: treating arithmetic values as “magic” and somehow different from everything else yields an inconsistent language and makes it hard to set up a good genericity mechanism (one that enables you to use [ARRAY \[INTEGER\]](#) as well as an [ARRAY \[PERSON\]](#) and so on).

Solution [2](#) might satisfy a theoretician, but experience shows that people doing numerical computation do want their mixed-type expressions and automatic conversions. The language design should try to accommodate this request — if that’s possible within the confines of statically typed object-oriented software engineering.

Solution [3](#) was used in Eiffel 3, which defined special conformance conventions for the basic types — [INTEGER](#) conforms to [REAL](#) and [DOUBLE](#), [REAL](#) conforms to [DOUBLE](#) — with associated implicit conversions. These rules suffice for cases such as *your_real + your_integer*, but not for *your_integer + your_real* since the first operand (the target of the call) is lighter than the second (the argument); a special rule, the *arithmetic expression balancing rule*, took care of this case. This achieves the basic aim of finding a place for traditional conventions in the type system of object technology, but only works for the predefined types. The programmer who wants to define a [COMPLEX](#) class and apply the same standard rules to the corresponding expressions can legitimately be jealous of the language designer and accuse him of callous selfishness.

“Programmer Admits Guilt in Shooting, Pleads Crime of Passion: ‘Selfish Language Designer was Driving Me Mad with Jealousy!’”. National Enquirer, 3 February 2003, p. 14.

Eiffel 3 had only one integer type, [INTEGER](#). With all the sized variants, the conformance and balancing rules would become quite clumsy.

[INTEGER](#), [INTEGER_8](#), [INTEGER_16](#), [INTEGER_32](#), [INTEGER_64](#), [NATURAL](#), [NATURAL_8](#), [NATURAL_16](#), [NATURAL_32](#), [NATURAL_64](#)

Solution [4](#), chosen here, generalizes solution [3](#): it retains the conformance, conversion and expression balancing rules expected for integer and real types, but not any more as “magic” properties of these basic types. Because they follow from a general language construct, [Conversion_types](#), and the associated validity and semantic rules, these properties are now a result of specifications included explicitly in Kernel Library classes [INTEGER](#), [REAL](#) and others; such specifications, no longer magic, may appear in any class that you want to endow with similar benefits.

As explained [elsewhere](#), Eiffel stays away from any form of in-class “**overloading**”: within a class a feature name such as *plus* may never denote **more than one feature**. The mechanism introduced here doesn’t affect this rule: what we are doing is catching some calls with an integer target and processing them through another feature with the same name in class [REAL](#) — similarly non-overloaded in that class.

← [“NO IN-CLASS OVERLOADING”](#), [5.22, page 167](#).

Repeated inheritance

16.1 OVERVIEW

Inheritance may be **multiple**: a class may have any number of parents. A more restrictive solution would limit the benefits of inheritance, so central to object-oriented software engineering.

Because of multiple inheritance, it is possible for a class to be a descendant of another in more than one way. This case is known as **repeated** inheritance; it raises interesting issues and yields useful techniques, which the following discussion reviews in detail.

The figure on the next page shows examples of repeated inheritance.

The present chapter is the last of three devoted to inheritance. It doesn't introduce any new language construct but explains the validity rules and semantics of repeated inheritance. As a consequence, it will complete our understanding of two important inheritance concepts: **inherited feature** and **name clash**.

← The other two were [6](#) and [10](#).

Our view of inheritance will only be final when we have grasped the semantics of reattachment and feature call, involving the powerful techniques of polymorphism and dynamic binding.

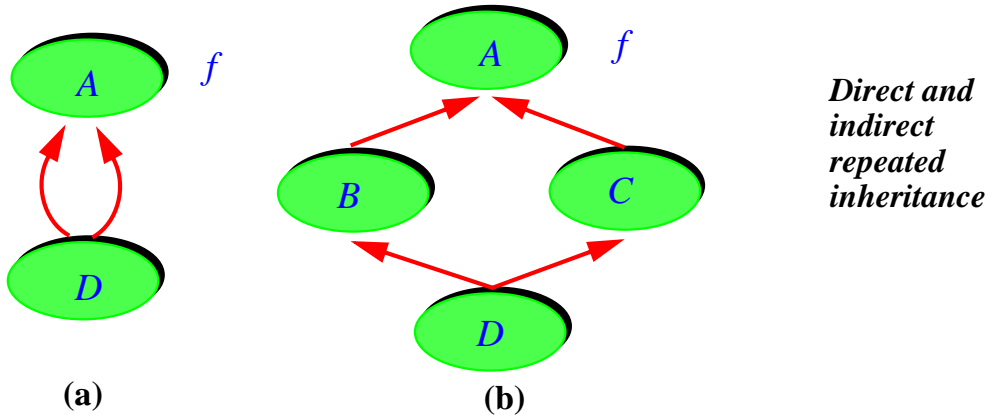
→ “[POLYMORPHISM](#)”, 22.11, page [606](#); “[DYNAMIC BINDING](#)”, 23.12, page [638](#).

This chapter is organized in four parts:

- We look into the circumstances of repeated inheritance.
- We identify the **two questions** that repeated inheritance implies for an object-oriented language — Are features shared or replicated? If replicated, what does this mean for dynamic binding? — and answer them through simple language rules.
- We explore applications of the resulting techniques.
- We finish off the formal rules.

16.2 CASES OF REPEATED INHERITANCE

The Parent rule indicates that the inheritance graph of a set of classes may not contain any cycles. It is perfectly possible, however, for two classes to be connected through more than one path. The figure on the next page provides two examples. ← “Parent rule”, page 178.



Here is the definition:



Repeated inheritance, ancestor, descendant

Repeated inheritance occurs whenever (as a result of multiple inheritance) two or more of the ancestors of a class *D* have a common parent *A*.

D is then called a **repeated descendant** of *A*, and *A* a **repeated ancestor** of *D*.

Why does the first sentence of the definition use the word “ancestor” rather than “proper ancestor”?

As shown by the two examples in the figure, *D* can repeatedly inherit from *A* directly (a) as well as indirectly (b).

The simplest case, called **direct repeated inheritance** and making *D* a **repeated heir** of *A*, occurs when *D* lists *A* in two or more **Parent** clauses:

```
class D inherit
    A rename ... redefine ... end
    A rename ... redefine ... end
    ... Rest of class omitted ...
```


The second case, **indirect repeated inheritance**, arises when at least one parent of D is a proper descendant of A , and at least one other is a descendant of A .



The discussion so far has neglected the generic parameters, if any, of the repeated ancestor A . In reality, a **Parent** is not just a class but a **Class_type** — a class name possibly followed by actual generic parameters. Uses of A as repeated ancestor with different actual generic parameters still cause repeated inheritance (D 's ancestors have a common parent class even though the corresponding **Parent** types are different); this case will show up in the consistency constraints and semantic rules.

→ “[THE CASE OF CONFLICTING GENERIC DERIVATIONS](#)”, 16.7, page 450.

16.3 THE TWO QUESTIONS OF REPEATED INHERITANCE

Repeated inheritance, although not a tool for beginners, is in fact a simple mechanism if approached properly. Only two issues arise, the answers to which make up this section and the next (and the principal new concepts of this chapter): does a feature inherited twice yield one feature, or two? If it yields two, which one should dynamic binding trigger?

First, the matter of repeatedly inherited features:

The first question of repeated inheritance: Fate of a repeatedly inherited feature

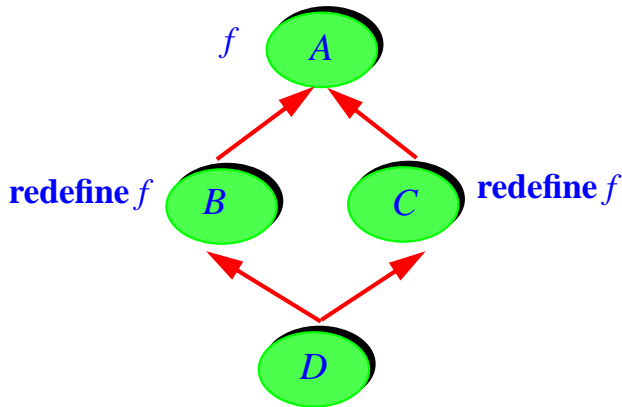
Given a feature from a repeated ancestor, what feature or features does it yield in a repeated descendant?

In the absence of repeated inheritance, the situation was simple: if Y is a descendant of X , every feature of X yields at most one feature of Y . But now things are not so clear any more. In either of the preceding pictures, what should D get out of a feature f of A : one feature, or two?

Usually one, but the Join mechanism (10.21 and 10.22) may merge several inherited features.

The second question arises from the combination of repeated inheritance and dynamic binding. Assume that in a case of indirect repeated inheritance, b on the last figure, one or both of the branches provides a new version for f :

The problem arises as soon as one branch redefines f ; for symmetry we assume both do.



*Conflicting
redefinitions*

Eiffel's dynamic binding policy (which suffers no exception) tells us that the call will use the version of f applicable to D (regardless of the declaration of a). But now we have two such versions. Hence:

**The second question of repeated inheritance:
Ambiguities under dynamic binding**

Given a feature repeatedly inherited under two different redeclarations, which one should a call execute if its target is statically of the repeated ancestor type and dynamically of the repeated descendant type?

Developing answers to these two questions is our principal task for this chapter. Both answers will turn out to be remarkably simple, but we must study the issues carefully before we can deduce the answers.

Also remarkable is that we can for a large part tackle the two questions separately, as they have little bearing on each other.

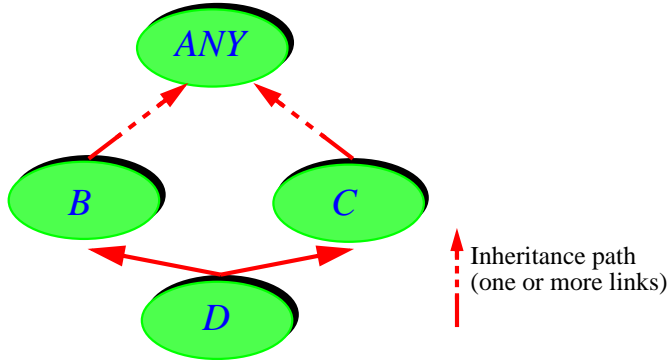
16.4 SHARING AND REPLICATION

Consider first the question of the fate of a repeatedly inherited feature. In the common descendant, does it yield one feature, or two?

We cannot settle for a single, universal answer. Depending on the context, either solution may be the right one, and you will need some leeway for choosing between them in any particular case:

- 1 • In some circumstances you may use repeated inheritance precisely because you like a feature of an ancestor so much that you want two of it.
- 2 • Often, however, one copy is enough. For example, the scheme illustrated on the figure above may arise when you write both B and C (the intermediate ancestors) as heirs of A because each needs A 's features, such as f ; D needs the new features introduced by B and C , but only one copy of A 's features.

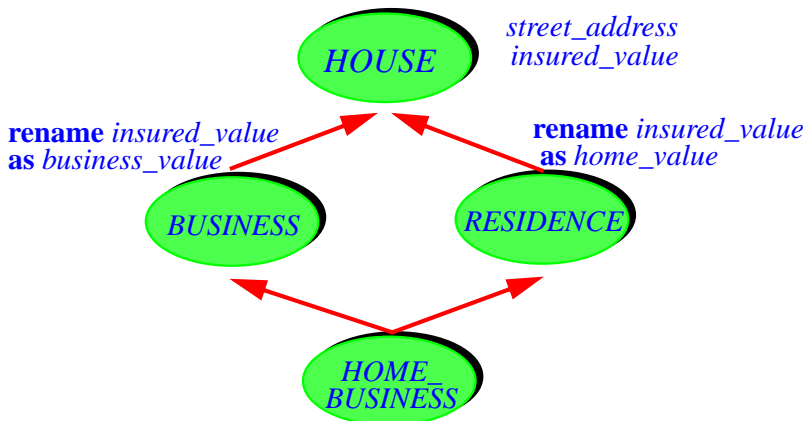
An extreme example of case 2 is the universal class *ANY* of the Kernel Library, an obligatory ancestor of all Eiffel classes. The presence of *ANY* means that any use of *multiple* inheritance is automatically a case of *repeated* inheritance, since even if the two parents, *B* and *C* on the figure below, do not explicitly list a common ancestor, they are automatically descendants of *ANY*, making *D* a repeated descendant of *ANY*. ← “*ANY*”, 6.5, page 172; see also chapter 35 for more details.



Any multiple inheritance causes repeated inheritance from ANY

For any non-trivial Eiffel system, the repeated inheritance structure induced by *ANY*, if we ever tried to draw it, would be rather luxuriant. In most cases, useful as the features of *ANY* are, you would not want your classes to inherit multiple copies of all of them.

The language could of course force you to choose one of the solutions, 1 or 2, globally for all the features from a given repeated ancestor. (This is roughly the C++ approach, through the notion of “virtual base class”.) But such a solution would be too restrictive: you may need replication for some features and sharing for some others. The Eiffel policy uses the expected default, sharing, but lets you choose the other possibility, replication, for any specific feature. The criterion is straightforward: is the feature inherited under a single name, or different names?



Homes, businesses, and home businesses

The **rename** subclauses shown will produce the desired effect: sharing for *street_address*, replication for *insured_value*. See next.

To see why we need such flexibility, consider the simple example, illustrated by the figure. In a system used by an insurance company, a class *HOUSE* has heirs *RESIDENCE* and *BUSINESS*. A special class *HOME_BUSINESS* handles the case of people who run a business from their house; it is legitimate to write this class as an heir to both of the previous two. The features of *HOUSE* include attributes *street_address* and *insured_value*. For the street address, an instance of *HOME_BUSINESS* should inherit a single attribute; but for *insured_value* it needs two, since the insured value may be different for the two viewpoints.

The same reasoning will apply to routines, such as `update_street_address` and `change_insured_value`.

The repeated inheritance mechanism gives you the desired flexibility: when writing a repeated descendant such as *HOME_BUSINESS* you can decide which repeatedly inherited features will yield single features (“sharing”) and which duplicate features (“replication”).

The policy is the simplest possible, and follows once again from the **no overloading** principle: within a class, make sure every name denotes a feature and only one. The principle implies that if the inherited features have the same final name, they *must* denote the same feature, and so will cause sharing; if they have different final names, they must yield different features, and will cause replication.

← “**NAMECLASHES**”.
[10.23, page 297.](#)

This is the answer to the first question of repeated inheritance, enabling us to introduce the principal rule of this chapter:



Repeated Inheritance rule

Let D be a class and B_1, \dots, B_n ($n \geq 2$) be parents of D based on classes having a common ancestor A . Let f_1, \dots, f_n be features of these respective parents, all having as one of their seeds the same feature f of A . Then:

- 1 • Any subset of these features inherited by D under the same final name in D yields a single feature of D .
- 2 • Any two of these features inherited under a different name yield two features of D .

Since A may be any ancestor, not just a proper one, the rule applies to direct repeated inheritance, where B_1, \dots, B_n are all the same as A , as well as to the indirect case.

This is the basic rule allowing us to make sense and take advantage of inheritance, based on the programmer-controlled naming policy: inheriting two features under the same name yields a single feature, inheriting them under two different names yield two features.

Sharing, replication

A repeatedly inherited feature is **shared** if case 1 of the Repeated Inheritance rule applies, and **replicated** if case 2 applies.

---- REMOVE A fine point about the rule's phrasing: it refers to "parents *based on classes* having a common ancestor" rather than "parents having a common ancestor" because a **Parent** is syntactically not a class but a type. With **class *D* inherit *P***... we are looking at ancestors not of *P* but of *P*'s base class.

← **Parent.syntax: page 171.**

Also, like all semantic rules, this one assumes that class *D* is valid. Otherwise, of course, we would get no feature at all in either case.

The Repeated Inheritance rule applies to attributes as well as to routines. It provides the designer of a repeatedly inheriting class with all the needed flexibility through proper choice of names:

- If two or more of the parents of *D* happen to have a common ancestor *A*, and you do not take any particular renaming action, each feature of *A* will yield just one feature of *D*. This will usually be what you want in simple cases, such as repeated inheritance from **ANY** as mentioned above. The rule also renders harmless a common oversight: making *A* a parent of *D* because *D* needs the features of *A*, forgetting that among the other parents of *D* one is already a descendant of *A*.
- If, however, you want two or more versions of a repeatedly inherited feature, just make sure that it is inherited under different names. This is the modern version of the loaves and fishes miracle: if you have one of a good thing, you may turn it into as many as you like, just by asking.

"They took up twelve baskets full of the loaves, and of the fishes", Mark 6:43. Scholars believe Loaf and Fish to be ancient Aramean for attribute and routine. See Proc. ALOOF 3 (Archaeo-Linguistic Object-Oriented Forum), Acapulco, 1998, pp. 798-923.

To determine which of the two cases applies, the only criterion that matters is the **final name** of the feature in *D*. It will be affected by any renaming performed in *D* itself as well as in intermediate ancestors between *A* and *D*. This means that, as the author of *D*, you are the master when it comes to setting the fate of a feature *f* coming from an indirect repeated ancestor through parents *B* and *C*:

- If *f* has the same name in *B* and *C*, *f* will normally be shared, but you may force replication by renaming one of the inherited versions, or renaming both forms with different names.
- If there has been some renaming between *A* and *D*'s parents, *f* will normally be replicated, but you may force sharing by renaming both inherited versions to the same name.



The sharing case of the Repeated Inheritance rule enables us to understand fully the notion of name clash and the prohibition of name clashes. The guideline (made formal by the Join rule) stated that a name clash is permissible only in three cases:

← After the definition of “Name clash” on p. 297.

- 1 • At most one of the clashing features is effective.
- 2 • The class redefines all the clashing features into a common version.
- 3 • The clashing features are the same feature, inherited without redeclaration from a common ancestor.

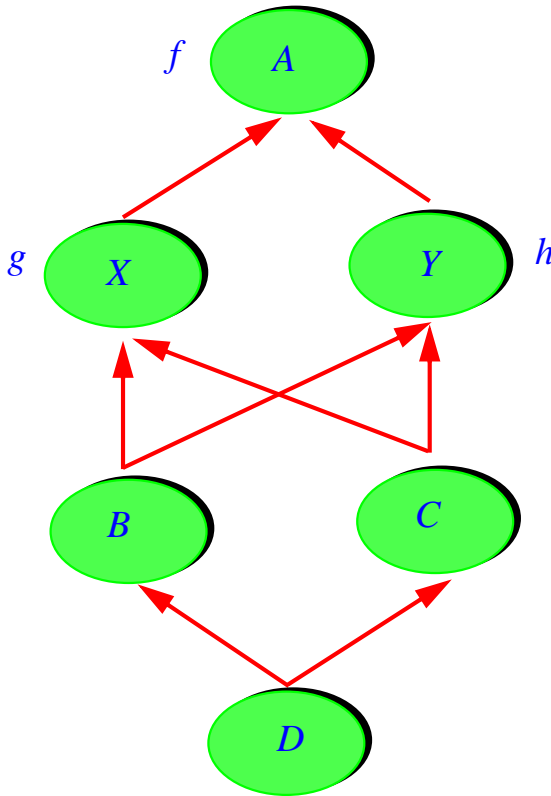
It’s the Repeated Inheritance rule that gives its meaning to the last case: even though there is an appearance of name clash because two parents *B* and *C* of a class *D* have a feature with the same name, in reality they are the **same feature**, inherited from a common ancestor *A*. If *D* inherits it in both cases under the same name, there is no real name clash; the sharing part of the Repeated Inheritance rule implies, naturally enough, that *D* will get the feature from *A*, exactly as if it had been declared as a direct heir of *A* without any intermediate classes.

This assumes of course that the feature is not redefined anywhere, otherwise it wouldn’t be the “same” feature. The next section will study the case of conflicting redeclarations.



One more general observation is in order on the scope of the Repeated Inheritance rule. As you will have noted from the definition, the rule only applies if f is the common seed of the features under consideration or, equivalently, if A is their origin. **Remember** that the *seed* of a feature is its original version in the most remote ancestor (the feature's *origin*) where it appears, regardless of any redeclaration or renaming that it may have endured between that ancestor and the current class.

← For the precise definitions see "[Origin, seed](#)", page 311. Joining a set of features gives all of them a new seed and origin.



Only the seed and origin matter

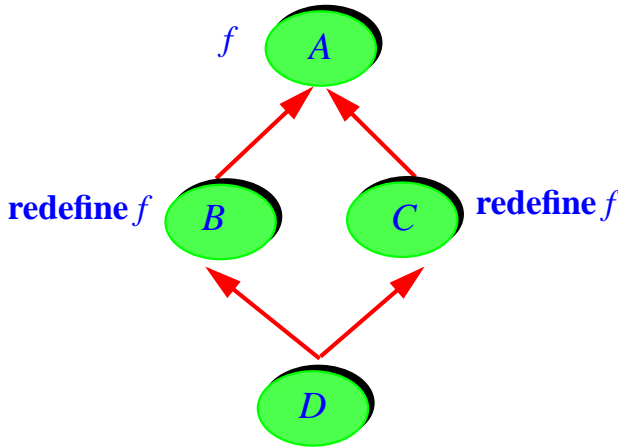


This requirement that A be the origin of f is important. Without it, as illustrated by the preceding figure, the Repeated Inheritance rule would be ambiguous. In the figure, f is a feature of A , but it is also a feature (an inherited one) of X and Y . All three classes are repeated ancestors of D . To infer sharing or replication from the rule, we need to know what repeated ancestor to consider. The rule's phrasing answers this question precisely: for f , the only relevant ancestor is class A , the origin of that feature. Similarly, to determine the fate of g and h , you must apply the rule (respectively) to X and Y , assumed to be the origins of these features.

16.5 THE CASE OF REDECLARED FEATURES

The Repeated Inheritance rule would define all we need to know about repeated inheritance were it not for the second question raised at the beginning of this chapter: ambiguities under dynamic binding.

Here is the picture again. We assume that both *B* and *C* redefine *f*:



*Conflicting
redefinitions*

If *D* inherits the two versions under the same name, it gets a single feature (sharing); otherwise, two different features (replication). But then what happens in a call of the form *a.f*, where *a* is declared of type *A* but is attached, at run time, to an instance of *D*?

The **sharing** case is easy because even in the absence of dynamic binding we have a problem: *D* gets two features with the same name. We know this case! It's a name clash. That the two features originally come from a common seed, the *A* version, doesn't matter here: at the level of *D* they are now *different features*.

← "NAMECLASHES".
10.23, page 297.

Studying the join rule has taught us that in such a conflict:

← "Join rule", page 319.

- If all of the variants, or all but one, are deferred and still have a single signature, there is no particular problem. They will all be joined, and live happily ever after as a single feature.

If some intermediate redefinition has led to different signatures, you may still use a join, but it will require a redefinition (or effecting) to a feature whose signature matches all the inherited ones.

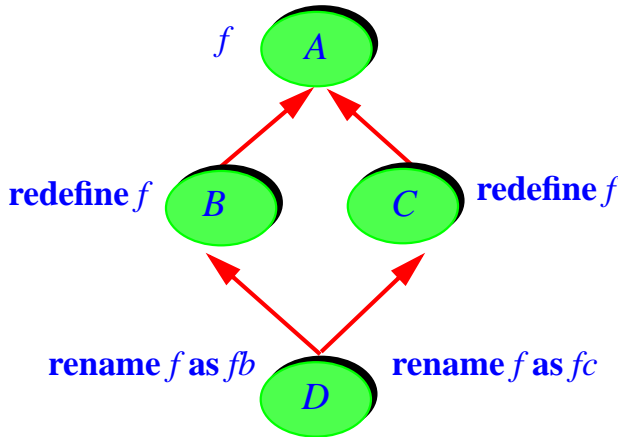
- If two or more are effective, the name clash would make the class invalid. In the general case we could resolve it by renaming, but here this would mean feature *replication* (the case discussed next), whereas we are explicitly assuming *sharing*, meaning all variants have the same final name. To remove the name clash we have to force a join by undefining all the effective features except at most one.

So here if both redefined versions are effective you must write D as either

```
class D inherit
  B
  undefine f end
  C
  ... Rest of class omitted ...
end
```

or the form that undefines the C version instead. You may also redefine both. If you do not include such an undefinition or redefinition, the class is invalid. We don't need any new validity constraint to express this requirement: the rules of the Feature Adaptation chapter took care of it.

This addresses the sharing case. But what if (as in the following figure) one or both features are renamed, causing replication?



**Redeclaration
and replication**

Because D renames the two inherited versions of f , we have a case of replication: f yields two features in D , called fb and fc . These features are truly different, since both B and C redefine their inherited versions of f . Note for generality that:

- The example assumes redefinition, but it would arise in any case of **redeclaration**, including conflicting effectings of an inherited feature. ←Redeclaration covers redefinition and effecting. See “Redeclare, redeclaration”, page 263.
- For symmetry, the example assumes that both B and C redefine f , but the problem would arise in the same way if one of these classes redefined the feature and the other kept the original.
- The renaming takes place at the level of D , but it could occur anywhere above, or for only one of the features, as long as the final names in D are different, causing replication.
- The problem will also arise, even without redefinition, in the case of **attributes**, as will be seen next.

Only dynamic binding with a target of static type based on A and dynamic type based on D causes a problem. There is nothing ambiguous about calls with a target entity dl of type D :

```

dl:  $D$ 
...
-- Attach dl to an object of type  $D$ :
create dl

dl.fb; dl.fc
-- A call of the form dl.f would be invalid,
-- since  $D$  has no feature of name f.

```

The first call will trigger execution of the version of f redefined in B , and the second will use the C version. Nothing new or surprising.

No difficulty arises either with polymorphism and dynamic binding applied to entities of types B or C :

```

bl:  $B$ ; cl:  $C$ 
...
create dl
-- Attach each entity to an object of type  $D$ :
bl := dl; cl := dl
-- The calls of interest:
bl.f; cl.f

```

To keep things simple, this example assumes that f is a procedure without arguments, that the classes involved are all non-generic — so that they are also types — and that D has no creation procedure. Also, the classes involved are all reference (non-expanded); if B or C were expanded, D would not conform to them, making the assignments invalid.

The two assignments are **polymorphic**, allowing bl and cl , although declared of types B and C , to become attached to an object of type D . The type rules permit this since D conforms to both B and C . Complementing polymorphism, **dynamic binding** commands that the version executed in each case is the one redefined by the ancestor closest to D . This means that (on the last line) the first call will trigger the B version and the second will trigger the C version. Still no particular problem.

→ [“POLYMORPHISM”, 22.11, page 606](#); [“DYNAMIC BINDING”, 23.12, page 638](#).

Where the situation becomes potentially ambiguous is if you use polymorphism and dynamic binding to call f on an entity al of type A , the repeated ancestor, as in



```

al: A; dl: D
...
create dl
    -- Attach the entity to an object of type D:
al := dl
    -- The call of interest:
al.f

```

Dynamic binding rules indicate that the call should trigger the version of f applicable to the actual object, which here is an instance of D . But there are two such versions of f resulting from the B and C redefinitions, and none of them is a priori better than the other.

Here is for example how B , C and D (deprived of any properties not relevant to this discussion) might appear:



```

class B inherit
    A redefine f end
feature
    f is do print ("Yes!") end
end

```

```

class C inherit
    A redefine f end
feature
    f is do print ("No!") end
end

```

```

class D inherit
    B
        rename f as fb end
    C
        rename f as fc end
end

```

WARNING: D as given is invalid. As explained next, one of the branches must use non-conforming inheritance.

Will the call $al.f$ print “Yes!”, obeying B , or will it obey C and print “No!”?



One may imagine various language solutions:

- We could rely on the order of the **Parent** clauses for *B* and *C* in *D*. But this is not acceptable: by reversing the order of parents, an innocuous editing change, you would change the semantics of the class. Besides, such a convention only makes sense for simple cases such as the above; with more levels of repeated inheritance, the “order” of ancestors becomes murky. In the earlier example, if *B* lists its parents in the order *X*, *Y*, but *C* lists its parents in the reverse order, what is the order of *X* and *Y* as ancestors of *D*? ← *Figure page 441.*
- We could require the class author to “select” one of the variants for use in dynamic binding, through a special language construct, every time such a conflict arises. This solution works and was indeed used in Eiffel 3. But further reflection has shown that a simpler approach was possible.

What makes that approach simpler is that it is more radical: *disallow polymorphism* whenever it could cause dynamic binding trouble. We suddenly remember that we have a straightforward way to disallow polymorphism when we don’t want it: instead of plain polymorphic inheritance, use **non-conforming inheritance**, also known as *expanded inheritance* because it builds on Eiffel’s notion of expanded class and indeed uses the keyword **expanded**. ← “*NON-CONFORMING INHERITANCE*”, 6.8, page 180.

A simple way to guarantee that an inheritance branch will not induce conformance is indeed to add that keyword to the corresponding **Parent** clause: if you declare a class as

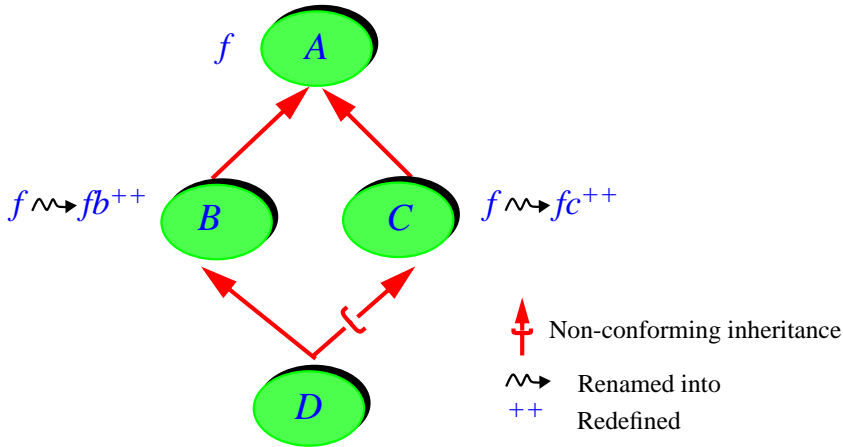
```
class D inherit
  expanded C
... No other parents ...
```

then attachments such as *c1 := dl*, with *c1* of type *C* and *dl* of type *D*, are not permitted. Without the **expanded** qualification, they would be valid.

This discussion still assumes that the classes involved are not themselves expanded classes.

To avoid the ambiguity in the previous example it suffices to guarantee that only one of the two branches is polymorphic, by declaring *D* as

```
class D inherit
  B
  rename f as fb end
  expanded C
  rename f as fc end
end
```



Removing dynamic binding ambiguity through non-conforming inheritance

This means that we have chosen only one of the two branches as permitting polymorphic attachment. So in the kind of situations seen above as causing trouble with polymorphism and dynamic binding:



```
al: A; dl: D
...
al := dl; al.f
```

There is no ambiguity any more: *dl* conforms to *al* in only one way, through *B*, so the feature *f* to be applied is the *B* version, *fb*.

The approach just studied implies resolving all potential dynamic binding ambiguities in favor of the same parent, *B* in the example. In rare cases you might want *al.f* to call the *B* version for some features *f*, but *al.g* to use the *C* version for a particular *g*. We will see [later in this chapter](#) how to adapt the technique to this case.

→ “[RETAINING VICTORS FROM ALTER-NATIVE BRANCHES](#)”, 16.11, page 460.

The scheme works just as well for direct repeated inheritance:

```
class D inherit
  expanded A
    rename f as f1 end
  A
    rename
      f as f2
    redefine
      f2
    end
  ... Rest of class omitted ...
end
```

With this form the version for dynamic binding is the redefined one, *f2*. Moving **expanded** to the first branch would select instead the original version, under the name *f1*.

You **must** mark one of the two **Parent** clauses involving *A* as cases of non-conforming inheritance — for example by using **expanded A** as here — to make valid such a case involving replication and redeclaration of one or more features.

This is the basic mechanism for resolving conflicts in such cases. Note that using an **expanded** qualification for one of the parent branches is the means, not the end. What the rule will state is that **conformance** may hold along at most one branch. If an inheritance branch is non-conforming for some other reason, then it does not create any conflict and there is no need for the explicit **expanded** qualification. In particular, if *C* is an expanded class — so far this section has assumed that none of the classes involved were expanded — the applicable conformance rules imply that *D* will not conform to *C* in spite of inheriting from it, so you may dispense with any special qualification, writing simply

← *The assumption was made on page 444.*

```

note
    note: "This version of the example assumes an expanded class C."
class D inherit
    B
        rename as fb end
    C
        rename f as fc end
end

```

The rule introduced by this discussion is the **Repeated Inheritance Consistency constraint**. The rule will be formulated precisely at the end of this chapter, but it's basically what we have just seen.

→ *"Repeated Inheritance Consistency constraint", page 466*

To gain a full understanding, we must now check what happens in two specific cases: attributes and conflicting generic derivations.

16.6 THE CASE OF ATTRIBUTES

The last example involved a feature *f* which was a routine. For attributes, a similar problem arises even in the absence of redefinition.

You may redefine an attribute, but this is only useful for type redefinition, since the redefined version must still be an attribute. See condition 6 of the Redeclaration rule.

← *"Redeclaration rule", page 313.*

The cause of ambiguity here is that a replicated attribute will yield two fields rather than one in the repeated descendant. Then, with dynamic binding, a reference to such a replicated attribute may become ambiguous in the same way as a reference to a multiply redeclared routine.

This may occur even with direct repeated inheritance of a class *D* from a class *A*, with a scheme such as this:



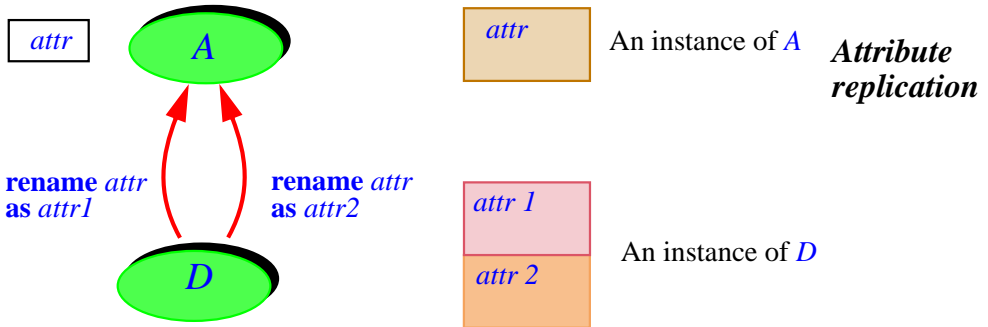
```

class A feature
  attr: SOME_TYPE
  some_procedure is do print (attr) end
end

class D inherit
  A
  rename attr as attr1 end
  A
  rename attr as attr2 end
end
    
```

WARNING: D as shown is invalid. Using non-conforming inheritance will make it valid; see the next version.

A direct instance of *A* has only one field, corresponding to *attr*. In an instance of *D*, however, *attr* yields two fields, for *attr1* and *attr2*:



As in the case of conflicting redeclarations, it is not clear which one of the fields the following should print:

```

a1: A; d1: D
...
create d1; a1 := d1; a1.some_proc
    
```

Because any new attribute implies a new field in every instance of the applicable class, we may view replication, for attributes, as implying a kind of implicit redefinition, similar in its effects to the explicit redefinition of routines.

Similar problem, same solution: whenever the Repeated Inheritance rule implies replication of an attribute, the Repeated Inheritance Consistency constraint will require that one of the inheritance paths involve non-conforming inheritance, as in → *“Repeated Inheritance Consistency constraint”, page 466*

```

class D inherit
  A
  rename attr as attr1 end
  expanded A
  rename attr as attr2 end
end
    
```

16.7 THE CASE OF CONFLICTING GENERIC DERIVATIONS



(This section, addresses the semantics of a rare case and may be skipped on first reading.)

Like attribute replication, different generic derivations from a common generic ancestor cause a form of implicit redefinition.

It is not hard to devise a simple example. Assume that A is generic, with one formal generic parameter G , and has a feature f whose signature involves G :

```
class A [G] feature
  f(x: G)is ... Routine body omitted ... end
end
```

```
class B inherit
  A [INTEGER]
end
```

```
class C inherit
  A [REAL]
end
```

What the body of f does is irrelevant; so is the exact nature of f — procedure as above, attribute or function — as long as f 's signature depends on G . The texts of classes A , B and C as shown only include the properties relevant to this discussion.

The different generic derivations of A used in the **Parent** parts of B and C cause f to have different signatures in these classes:

```
in B: [], [INTEGER]
in C: [], [REAL]
```

This means that the name f , in these two classes, denotes **different features**: a feature is defined not only by its specification (assertions) and its implementation, but also by its signature.

What then if you want to write a class D as heir to both B and C ? This creates a conflict, as in the two previously studied cases (routine redefinitions and attributes). Because the features are different, sharing is impossible in this case, but the same replication-based solutions are available as in the previous two:

- 1 • Using replication and making sure that at most one of the inheritance paths uses conforming inheritance.
- 2 • Letting one of the versions override the other through undefinition.

→ Chapter 12 studies generic classes and generic derivations.

← The signature of a feature is the specification of its argument and result types. See “Signature, argument signature of a feature”, page 149.

← The reasons that preclude sharing were analyzed at the beginning of 16.5, page 442.

The second solution requires special care here because the signatures are different. The problem is that if a version overrides the other it must have a conforming signature; but this may not be true because of conflicting generic derivations. In the above example, indeed, the signatures of the *B* and *C* versions are incompatible since neither of the types *INTEGER* and *REAL* conforms to the other. The only solution is to undefine both features and provide a fresh redeclaration in *D*. Here, in the absence of a useful common descendant to *INTEGER* and *REAL*, that fresh feature may only be of the form

```
f(x: NONE) is do ... Some routine body ... end
```

and hence cannot do anything useful with its argument *x*. (Recall that *NONE* is a common descendant of all classes, but has no exported feature.) ← “*NONE*”, 6.6, page 175.

In more favorable cases, one of the actual generic parameters used for generic derivations of *A* in *B* or *C* will conform to the other; then you may use its version of *f* to overtake the other’s. Redefinition into a version whose signature conforms to both (if possible not just through *NONE*) will also work.

16.8 KEEPING THE ORIGINAL VERSION OF A REDEFINED FEATURE

The most novel aspect of the Repeated Inheritance rule is the replication case: here for the first time there is a way for one feature of a parent to yield two or more features in an heir.

Among other applications, this mechanism enables us to “redefine our feature and eat it”: provide a new version of an inherited routine, but retain the original as well.

In the majority of cases, you do not need repeated inheritance to achieve this goal, because the most common use of the original version is to help write the redefined version. We have seen the simple language mechanism that directly addresses this need: *Precursor*. You will simply write the redefinition of a routine as

```
your_routine (args: ...)
  do
    “Something else”
    Precursor (...)
    “Yet something else”
  end
```

With this technique — applicable only to routines, not attributes — the inherited version does *not* remain a feature of the new class: all you have is its implementation, usable only in the corresponding redefinition.

It's a very simple setup. You can use it whenever **Precursor** doesn't suffice because you want to keep the original as a feature of the new class with all the associated privileges. For example:



```
class MONEY_MARKET_ACCOUNT inherit
    SAVINGS_ACCOUNT
    redefine compute_interest end

    expanded SAVINGS_ACCOUNT
    rename
        compute_interest as
            compute_interest_as_for_plain_savings
    export
        {NONE} compute_interest_as_for_plain_savings
    end

... Rest of class text omitted ...
end
```

← Compare with the examples in the discussion of **Precursor** in 10.24, page 299.

This class illustrates what to do if you want to keep the original version, here under the name `compute_interest_as_for_plain_savings`, for internal purposes only: hide it from clients at the point of inheritance through a New exports clause that stipulates access to no useful clients. This is required in particular if the original version does not preserve the invariant of the new class.

← “Adapting the export status of inherited features”, . . . page 204.

16.9 USING REPLICATION: COUNTERS AND ITERATION

The technique studied in the previous section relies on the Repeated Inheritance rule's automatic mechanism for duplicating routines and attributes. Let's see a couple more applications of this possibility.

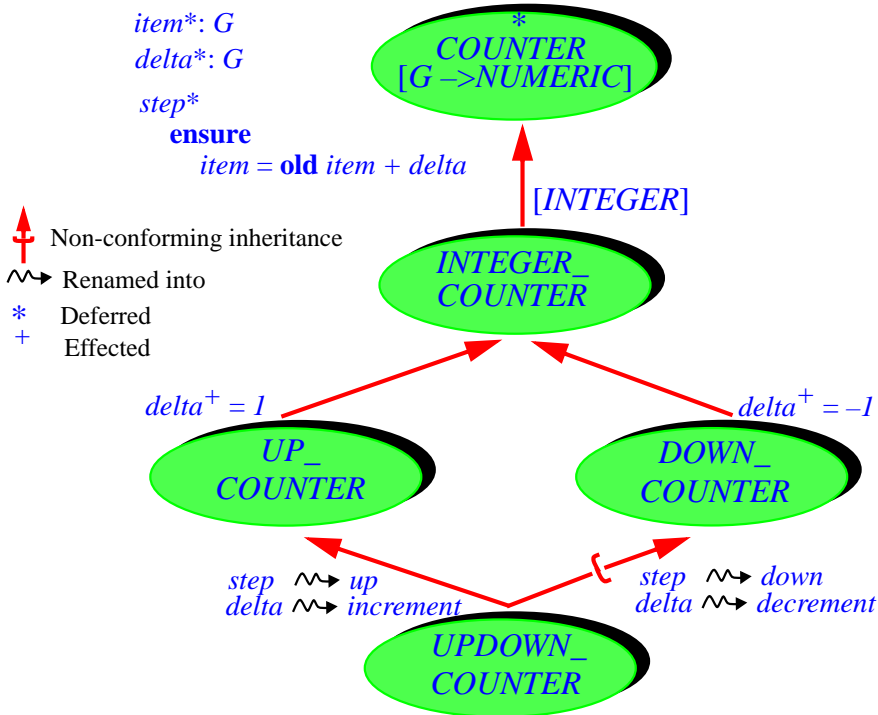
The first example is a pedagogical exercise (due to Christine Mingins). The inheritance hierarchy is shown on the following figure. We have a general notion of `INTEGER_COUNTER` with

- A query `item` giving the current value associated with the counter.
- A procedure `step` with no argument, to advance the counter by one step.
- A query `delta` giving the amount by which a `step` will change the value.

For more generality we can make `INTEGER_COUNTER` inherit from `COUNTER [INTEGER]` and introduce these three features at the level of the generic class `COUNTER`

Right from the start (in `COUNTER`), procedure `step` should have a postcondition stating `item = old item + delta`.

*Counters, up,
down and both*



The figure is explicit enough that we don't need to write the actual class texts. We have two variants of *INTEGER_COUNTER*, representing counters that increment their value by +1 and -1. It suffices in *UP_COUNTER* to effect *delta* as returning +1, and -1 in *DOWN_COUNTER*. Procedure *step* should be effected to execute $item := item + delta$; this may be done in *INTEGER_COUNTER* or even *COUNTER*.

Then we want a notion of counter that can count both up and down, with two procedures *up* and *down*. It suffices that *UPDOWN_COUNTER* inherit from both *UP_COUNTER* and *DOWN_COUNTER*, renaming *step* to *up* and *down*, and *delta* to *increment* and *decrement* (these two words being used as nouns, as in “an increment”, not as verbs as in “increment this”). Both cases are valid uses of inheritance: an updown counter is definitely an up counter, and a down counter as well. For dynamically bound uses of *step* and *delta* on updown counters known statically as just counters, we choose the “up” interpretation, so the inheritance from *UPDOWN_COUNTER* to *DOWN_COUNTER* is non-conforming. The machinery of repeated inheritance gives us exactly what we need thanks to replication.

If the postcondition of *step* is to make sense in both versions *up* and *down* of this feature, it is critical that the redeclarations of *step* go hand in hand with those of *delta*: the postcondition must mean $item = \text{old } item + increment$ in *UP_COUNTER* and $item = \text{old } item + decrement$ in *DOWN_COUNTER*. This will require a semantic clarification in the next section.

→ “*Replication Semantics rule*”, page 459.

The second example deals with multiple iterations. The agent mechanism actually provides a more dynamic way to address this issue, but the technique described here can still be interesting in some cases. → *Chapter 27 covers agents and include several iteration examples.*

Consider an **iterator** class providing a way to perform certain operations on every element of a certain structure. These operations are denoted in the iterator class by deferred routines; descendants will effect them to represent the actual operations needed in a particular iteration case. For example a class *LINEAR_ITERATION* (such as provided by the iteration cluster of EiffelBase) may include a procedure *do_until* with this general form: ← *Compare to until_do in “PARTIALLY DEFERRED CLASSES AND PROGRAMMED ITERATION”, 10.15, page 277.*

```
do_until (s: TRAVERSABLE [T])
    -- Iterate on s, up to and including
    -- the first item satisfying test.
do
    from
        start (s); prepare (s)
    until off (s) or else test (s) loop
        action (s); forth (s)
    end
    if not off (s) then action (s) end; wrapup (s)
end
```

Any effective descendant of *LINEAR_ITERATION*, describing an iteration scheme over a specific kind of data structure — for example a list implemented by an array with a current position *position*—, will effect *start*, *forth* and *off* to provide, for the corresponding iterative structure:

- An implementation of *start*, bringing the cursor iteration to the first position; in the array case, it will be the assignment *position := 1*.
- An implementation of *forth*, to advance the cursor by one position: for arrays, *position := position + 1*.
- An implementation of *off*, to query whether we have exhausted the list of meaningful cursor positions: for arrays, the test *position > count*, where *count* is the number of occupied positions.

The class providing these effective declarations may be a class *LIST_ITERATION*. All that remains to do for a descendant needing actual iterations is to effect the routines describing the actions and tests to be performed on every list element: *prepare*, *test*, *action* and *wrapup*.

But what if a class needs **two** variants of the iteration mechanism? It is possible to use repeated inheritance from *LIST_ITERATION*, with sharing for the traversal routines (*start*, *forth*, *off*) and replication for the operation routines *prepare*, *test*, *action* and *wrapup*, which need separate versions.

An example is an application that handles lists of atomic particles, as described by the class

```

class PARTICLE feature
  mass: REAL; speed: VECTOR
  positively_charged: BOOLEAN
  ... Other attributes and routines ...
end

```

where the lists are sorted by increasing mass. The application needs both to

- 1 • Print the mass of all particles in a list, up to and including the first positively charged one.
- 2 • Compute the total vector speed of the first fifty particles in the list and store it into an attribute *total_speed*. (To add speeds, we assume a procedure *add* in class *VECTOR*.)

Using repeated inheritance:



```

class PARTICLE_LIST_PROPERTIES inherit
  LIST_ITERATION [PARTICLE]
  rename
    do_until as print_masses, prepare as do_nothing,
    test as positive_test, action as print_one_mass,
    wrapup as do_nothing
  end
  expanded LIST_ITERATION [PARTICLE]
  rename
    do_until as add_speeds, prepare as set_speed,
    test as at_threshold, action as add_one_speed,
    wrapup as do_nothing
  end
feature
  positive_test (s: FIXED_LIST [PARTICLE]): BOOLEAN
    -- Is particle at current cursor position in s positive?
  do
    Result := s.item.positively_charged
  end

  print_one_mass (s: FIXED_LIST [PARTICLE])
    -- Print the mass of particle at cursor position in s.
  do
    print (s.item.mass)
  end
end
... Rest of class omitted ...

```

16.10 THE SEMANTICS OF REPLICATION

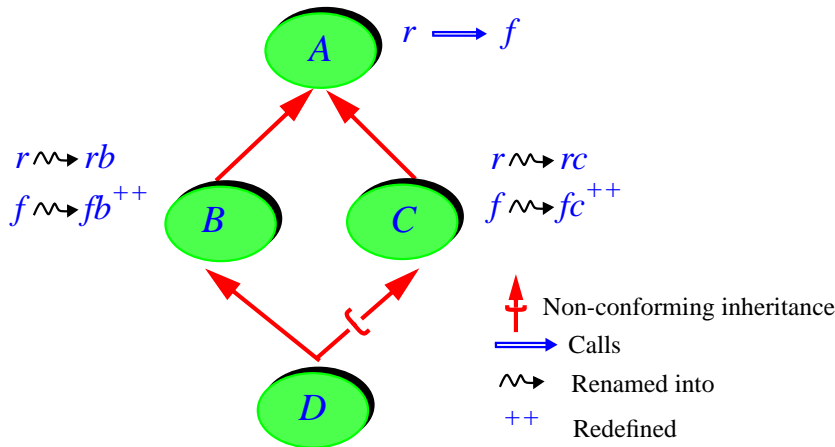
The Repeated Inheritance rule specifies that a feature inherited repeatedly under two different names yields two features in the repeated descendant. We must clarify what replication entails, especially for routines. We need the corresponding semantic rule to ensure the correct functioning of both examples reviewed in the last section.

For attributes, we saw that replication is to be taken literally: instances of the common descendants will have two separate fields.

← Figure “Attribute replication”, page 449.

For routines, we normally do not need to replicate any code. But a special case arises when *two* or more routines, calling each other, get replicated along the same branch.

Consider our usual diamond-shaped repeated inheritance structure, with two features *r* and *f* where *r* is an effective routine; *f* may be an attribute or a routine. We assume that *r* calls *f*:



Multiple routine replications

Both *r* and *f* get renamed differently along the two branches, so the Repeated Inheritance rule implies replication for both. In addition *f* gets redefined, so that the Repeated Inheritance Consistency constraint applies. The constraint states that at most one of the inheritance paths may support conformance; this is achieved here by using non-conforming inheritance from *D* to *C*. Viewed from *A*, then, the dynamic binding version of *f* in *D* is the *B* version, *fb*, in the sense that it’s the feature called by *al.f*, for *al*: *A* dynamically attached to an object of type *D*.

→ “Dynamic binding version”, page 468.

All this, as we have seen, also applies whenever *f* is an attribute, even if neither *B* nor *C* redefines it.

← “THE CASE OF ATTRIBUTES”, 16.6, page 448.

Such situations raise a new problem: since *r* calls *f*, and *D* now has two versions of the original *f*, which one of these should *rb* and *rc* call?

Since the example include no redefinition for the features of seed r (r , ra , rb), the features ra and rb are just duplicates of the original r . If they are identical, they will call the same version of f in D ; if so, that version should presumably, in keeping with the spirit of the Repeated Inheritance Consistency constraint, be fb , as fc comes from the non-conforming branch.

← “Seed” was defined on page 311. A revised definition appears on page below.

But is this right? Conceptually, D has two versions of r and two versions of f . The original property of r was that it called the corresponding version of f . There doesn't seem to be any good reason for a replicated version of r to call a version of f that results from a mutation of the original along a *different* inheritance branch.

A rare but illuminating case is for f to be the same routine as r :

```

r (args: ...)
  -- A routine that may call itself recursively
  do
    ...
    r (other_args)
  end

```

Assume B redefines r but (to keep things simple) C retains this original A version shown above. It seems reasonable to expect that the highlighted call to r should still be a recursive call, both in C and in D . Why should we call the B version? This seems a betrayal of the originally intended semantics, since the routine would now cease being recursive.

These reflections suggest that we should take the notion of replication seriously. Compiler writers, of course, will avoid physically duplicating the code of a routine whenever they can. But an Eiffel programmer should be able to believe the replication case of the Repeated Inheritance rule literally, as if it caused code duplication for a routine in the same way it causes field duplication for an attribute.

----- EXPLAIN



Call Sharing rule

VMCS

It is valid for a feature f repeatedly inherited by a class D from an ancestor A , such that f is shared under repeated inheritance and not redeclared, to involve a feature g of A other than as the feature of a qualified call if and only if g is, along the corresponding inheritance paths, also shared.

If g were duplicated, there would be no way to know which version f should call, or evaluate for the assignment. The “selected” version, discussed below, is not necessarily the appropriate one.

The following rule expresses this property:

Replication Semantics rule

Let f and g be two features both repeatedly inherited by a class A and both replicated under the Repeated Inheritance rule, with two respective sets of different names: $f1$ and $f2$, $g1$ and $g2$.

If the version of f in D is the original version from A and either contains an unqualified call to g or (if f is an attribute) is the target of an assignment whose source involves g , the $f1$ version will use $g1$ for that call or assignment, and the $f2$ version will use $g2$.

This rule (which, unlike other semantic rules, clarifies a special case rather than giving the general semantics of a construct) tells us how to interpret calls and assignments if two separate replications have proceeded along distinct inheritance paths.

Another way to state this is that replication may cause a form of **implicit redefinition**: if the replicated routine r calls a feature f that has been redefined, or is an attribute (in either case causing physical replication), then even if r has not been redefined anywhere in the process we must pretend that it has — to versions that call the corresponding versions of f .

If you review the examples of the preceding section, you will notice that they can only work under this rule:

- In the multiple counter example, the postcondition of up , inherited by $UP_COUNTER$ from $COUNTER$ as $item = \mathbf{old\ }item + \mathbf{delta}$, must use the version of $delta$ applicable to $COUNTER$: $increment$, with value +1; for $DOWN_COUNTER$, the corresponding postcondition for $down$ must use $decrement$, with value -1.

- In the multiple iteration example, *print_masses* and *add_speed*, both of them mere renamings of the general iteration procedure *do_until*, must use the versions of the list item operations *prepare*, *test*, *action* and *wrapup* applicable to its branch.

In both cases this means that even though the calling routine — *step*, the seed of both *up* and *down*, and *do_until*, the seed of both *print_masses* and *add_speeds* — is never explicitly redefined, it must take into account the separate redeclarations of features that it calls.

16.11 RETAINING VICTORS FROM ALTERNATIVE BRANCHES

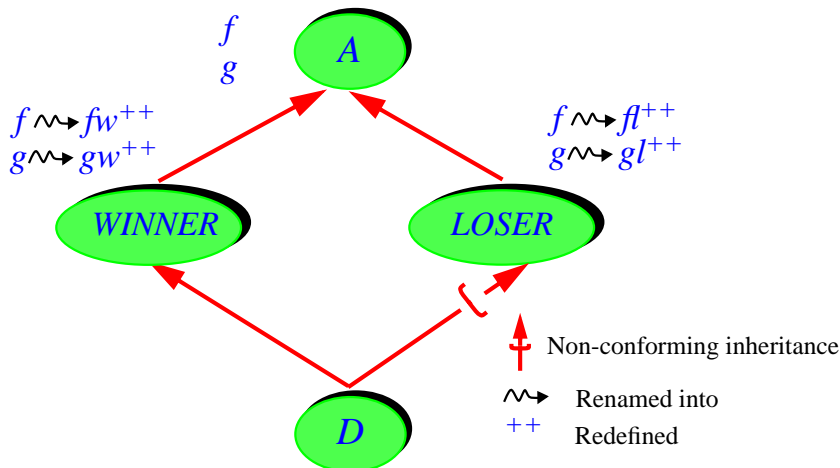
This is a time for celebration: by now you know all the important concepts of inheritance and feature adaptation.



There remains to see a technique addressing a fine point of the combination between dynamic binding and replication (this section) and the precise rules for the concepts that we have studied but not yet formalized (next two sections). All this is material that you can safely skip on first reading.

In studying the rules for redeclaration under repeated inheritance we have seen how to avoid ambiguities by forcing all branches but one to involve non-conforming inheritance. What if we want some of the versions for dynamic binding to come from another branch?

Let's consider again our basic figure for such cases:



The winner and the loser

*This is the figure of page 447, with a new feature *f* and different names for the intermediate classes.*

We have learned how to resolve the potential ambiguity of calls such as *al.f* for *al:A* dynamically attached to an object of type *D*: make sure that one of the inheritance paths involves non-conforming inheritance. Then the call will use the version from the other branch.

Once we have settled on where to use non-conforming inheritance, this policy will be the same for all features such as *f*. To emphasize this property, the intermediate classes (*B* and *C* in the original examples) have been renamed *WINNER* and *LOSER* on the last figure. The choice between them is indeed absolute: like the America's cup, this is a race with no second place.

But what if we want to use the *WINNER* version for feature *f*, and for another feature subject to the same problem — *g* on the figure — we want to retain the version redeclared in the other class, *LOSER*?

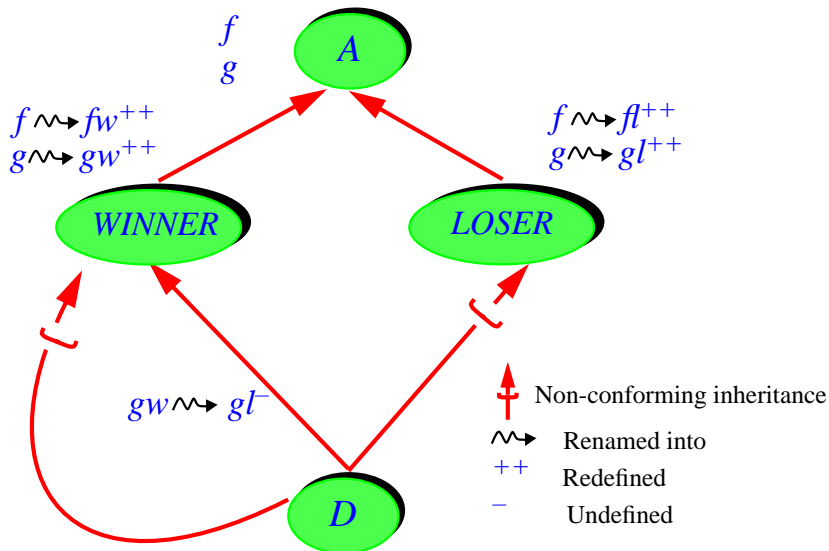
The reason this hasn't been a major concern until this stage of the discussion is that the case is not common. Most of the time, in repeated inheritance situations of the above type with conflicting redeclarations, one of the parents is indeed the victor, providing all the variants for dynamic binding. (Sometimes it's because its form of inheritance was more for subtyping, and the loser's was more implementation inheritance.)

On various forms of inheritance see the inheritance methodology chapter in [Object-Oriented Software Construction, 2nd edition](#)".

But there will be exceptions to this observation, and we need a way to address them. The idea is simply to rely on the Join mechanism.

First assume that although you want two versions of the original *f* you need only one of *g*, the *LOSER* version. Then a simple join will solve the problem: it suffices to inherit both versions under the same name, and to undefine the one from *WINNER*; the other will take over.

If you want to keep both versions of *g*, but make *gl* the selection for dynamic binding from higher-ups, you will use essentially the same technique but in this case you need to inherit *once more* from *WINNER* (as if mere repeated inheritance from *A* were not already enough), this time in non-conforming form:



*We like it so
much we want
not just two but
three of it!*

This gives *D* another version of *gw*, leaving you free to do whatever you like with the first — the one used for dynamic binding — so that you can let it be overridden by *gl*'s implementation through renaming, undefinition and join (the loser's revenge):

```
class D inherit
  WINNER
    rename
      gw as gl
    undefine
      gl
    end
    -- One more time, with feeling:
    expanded WINNER
    -- Not such a total defeat after all:
    expanded LOSER
  ... Rest of class text omitted ...
end
```

You will obtain a similar effect by redefining the *gl* from *LOSER* and the *gl* renamed from *gw* (in the conforming *WINNER* branch) into a common feature. For attributes — which you can't undefine — this is the only possible technique.

16.12 THE NEED FOR SELECT

--- EXPLAIN !!!



Select clauses

`Select \triangleq select Feature_list`

The **Select** subclass serves to resolve any ambiguities that could arise, in dynamic binding on polymorphic targets declared statically of a repeated ancestor's type, when a feature from that type has two different versions in the repeated descendant.

--- EXPLAIN



Select Subclause rule *VMSS*

A **Select** subclause appearing in the parent part for a class *B* in a class *D* is valid if and only if, for every **Feature_name** *fname* in its **Feature_list**, *fname* is the final name in *D* of a feature that has two or more potential versions in *D*, and *fname* appears only once in the **Feature_list**.

This rule restricts the use of **Select** to cases in which it is meaningful: two or more “potential versions”, a term which also has its own precise definition. We will encounter next, in the Repeated Inheritance Consistency constraint, the converse requirement that if there is such a conflict a **Select** *must* be provided.

16.13 THE REPEATED INHERITANCE CONSISTENCY CONSTRAINT

Although we have seen all the concepts, it remains to formalize some of the definitions and rules:

- The **versions of a feature** and its **dynamic binding version** in a descendant of its class of origin.
- The **Repeated Inheritance Consistency constraint** — the major constraint on the use of repeated inheritance.
- The precise definition of **inherited features of a class** — needed for the more general notion of “features of a class”

- As a consequence, the precise definition of the **final name set** of a class and the **Feature Name rule**, governing the choice of feature names and avoiding unwanted name clashes.

As noted, this material and the remainder of this chapter are not required on first reading.



The purpose of the Repeated Inheritance Consistency constraint is to make sure (by permitting at most one conforming inheritance path) that for any feature of a class there is at most one *dynamic binding version* in any proper descendant. Before defining “dynamic binding version” we need to know what a “version” is, but here we’ve essentially done the job already by introducing the notion of “seed”:



Version

A feature g from a class D is a **version** of a feature f from an ancestor of D if f and g have a seed in common.

The seed of a feature was defined as the original form of the feature in the class where it was first introduced, prior to any redeclarations, renamings or other transformations in proper descendants. A version of f is a reincarnation of f in a descendant.

← “*Origin, seed*”,
page 311

The definition of “seed” implies that if f is immediate (introduced by its class as a new feature) then the common seed of f and g mentioned in the above definition of “version” is f itself.

When may a feature have more than one version in a proper descendant of its class of origin? The answer was given by the semantic rules of this chapter: Repeated Inheritance and Replication Semantics rules. The following rule brings nothing new, but summarizes the consequences of these previous results.

← “*Repeated Inheritance rule*”, page 438;
“*Replication Semantics rule*”, page 459.



Multiple versions

A class D has n **versions** ($n \geq 2$) of a feature f of an ancestor A if and only if n of its features, all with different final names in D , are all versions of f .

-- REMOVED CLAUSES:

, and any two among them satisfy any of the following properties:

- 1 • A redeclaration applied to one has not been applied to the other.
- 2 • Any of them is an attribute.
- 3 • They have different signatures.
- 4 • Any of them calls a feature of A having (recursively) two or more versions in D .

----- END REMOVED CLAUSES -- DISCUSSION BELOW IS OBSOLETE

Although this rule doesn't mention repeated inheritance, it can only be understood as a consequence of the rules introduced in this chapter: the only way in which D may, as required by the definition, have two or more versions of f — meaning, from the definition of “version”, two or more features with the same seed — is through the replication mechanism of repeated inheritance.

Case 1 is the most common source of multiple versions: the features have been redeclared in different ways along different inheritance paths, or one has been redeclared and the others haven't.

To cover both of these cases, the rule uses careful phrasing: at least one redeclaration has occurred (along one of the inheritance branches) that applies to one of the features but not to the other. This may mean, for the other, no redeclaration at all, or a different redeclaration.

Case 2 follows from the discussion of what replication means in the special case of attributes. Note that it suffices that one of the features be an attribute; it may have as its seed a function that, along the other branch, was either not redeclared or redeclared as a function. ← “[THE CASE OF ATTRIBUTES](#)”, 16.6, page 448.

Case 3, as stated, sounds very general, but if you reflect about it you will realize that it is only relevant in the other special case of replication: conflicting generic derivations. True, another source of differing signatures would be redefinition; but then the more general case 1 will also apply. ← “[THE CASE OF CONFLICTING GENERIC DERIVATIONS](#)”, 16.7, page 450.

Case 4 follows from the discussion of replication semantics: even if a routine has not been explicitly redeclared, it may have an implicit redefinition as a result of replication under repeated inheritance, if it calls a feature that has been redeclared. This case only applies to routines, since only a routine may call another feature (routine or attribute). Note that the call may be in the Routine_body but it might also be, as in the COUNTER example, in a Precondition or Postcondition, as well as in a Rescue clause. ← “[THE SEMANTICS OF REPLICATION](#)”, 16.10, page 457.

For the reader interested in theoretical consistency: clause 4 may appear to risk infinite recursion, since it is possible for a routine r to call a routine s which also calls r . This was the case with the example of a recursive routine interpreting the definition constructively — as a definition by induction, or a fixpoint — avoids this problem: to determine the set of features with more than one version in D we first apply cases 1, 2 and 3, the non-recursive cases, to all relevant features; then we repeatedly apply clause 4 to include any features that call a feature already in our set, stopping at the first iteration that yields nothing new. The process is guaranteed to terminate, since the set of features of D (and hence too the transitive closure of the call graph) is finite.

For an introduction to fixpoints and the theory of recursive definitions see [Introduction to the Theory of Programming Languages](#)”.

Throughout this chapter we have used the Repeated Inheritance Consistency constraint, which removed ambiguities for dynamic binding in the presence of conflicting redeclarations. For all practical purposes the earlier informal statements of the constraint were sufficient, but now we can express it in a completely precise form:



Repeated Inheritance Consistency constraint *VMRC*

It is valid for a class D to have two or more versions of a feature f of a proper ancestor A if and only if it satisfies one of the following conditions:

- 1 • There is at most one conformance path from D to A .
- 2 • There are two or more conformance paths, and the **Parent** clause for exactly one of them in D has a **Select** clause listing the name of the version of f from the corresponding parent.

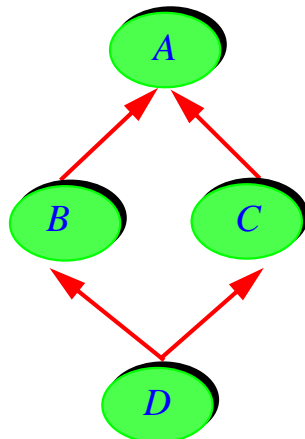


A “conformance path” is a sequence of classes from D to A such that each of the associated current types conforms to the next. Thanks to the non-conforming inheritance it is possible for D to have some inheritance paths to A that are not conformance paths.

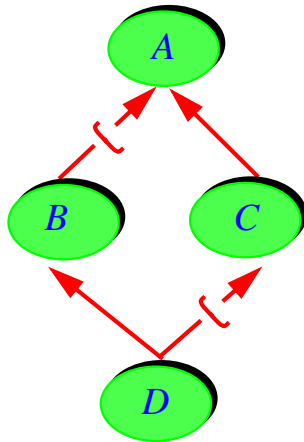
← “Conformance path”, page 389; “Current type”, page 365.



According to this constraint **it is not invalid** for a class to have more than one conformance path to a proper ancestor if no replication causes any ambiguity for dynamic binding. As soon as such a potential ambiguity arises, however, you need to make sure that all inheritance paths, except possibly one, involve at least one non-conforming link.



Conversely, nothing forces you, in a repeated inheritance situation with or without replication, or in any inheritance situation, to have a conforming path. A class may inherit from another, singly or multiply, without conformance of the associated current types. This is the case of facility or implementation-only inheritance, which does not permit subtyping. It is not the most common use of inheritance, but it is possible:



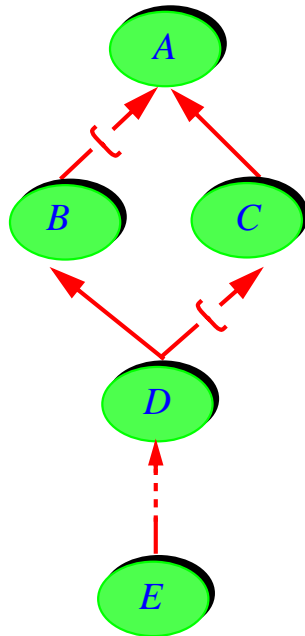
No path conforms

Suffering from an proper ancestor's repeated inheritance?

In this case there is no polymorphism: with $al: A$ and $dl: D$, attachments such as $al := dl$ are invalid. (Similarly, with the assumptions of the figure, $al := bl$ and $cl := dl$ with $bl: B$ and $cl: C$.)

A final comment on the Repeated Inheritance Consistency constraint — important in particular for compiler writers — is that the rule as stated might seem to require, for any feature f of a class A , verification in **every** proper descendant E of A , at least every E such that repeated inheritance with replication occurs somewhere between A and E , even if the culprit is not E but an intermediate descendant D :

You don't have to worry about what happens in E , however: thanks to the definition of “version”, if possible dynamic binding ambiguities arises for E , that can only be (if the only cases of repeated inheritance are those appearing on the figure) because they arise for D ; once you resolve them for D in accordance with the Repeated Inheritance Consistency constraint, that will take care of E as well.



Thanks to the constraint we can now define *the* dynamic binding *version* (note the singular) of a feature in any descendant of its class of origin:

DEFINITION

Dynamic binding version

For any feature f of a type T and any type U conforming to T , the **dynamic binding version** of f in U is the feature g of U defined as follows:

- 1 • If f has only one version in U , then g is that feature.
- 2 • If f has two or more versions in U , then the Repeated Inheritance Consistency constraint ensures that either exactly one conformance path exists from U to T , in which case g is the version of f in U obtained along that path, or that a Select subclass name a version of f , in which case g is that version.

As you will have noted:

- The definition has moved on from classes to types, since this is what matters for feature calls and dynamic binding. All the concepts transpose immediately; in particular, “features of a type” was defined precisely in an earlier chapter. ← “CURRENT TYPE, FEATURES OF A TYPE”, 12.11, page 365.
- If T and U are the same type, case 1 applies; so the definition indicates — as it should — that f is its own dynamic version.

The definition enables us to obtain a **single** dynamic binding version for every inherited feature. This is of course the very purpose of the entire present discussion, and the reason for the Repeated Inheritance Consistency constraint.

The result is at the very heart of the object-oriented machinery of Eiffel: when discussing the fundamental computational mechanism, feature call, we will specify that a call $a.f(\dots)$ triggers the **dynamic binding version of f** in the type of the object dynamically attached to a . Thanks to the preceding rules and definitions, we now have the guarantee that this notion will always be unambiguously defined, even under the most sophisticated forms of multiple and repeated inheritance.

16.14 THE INHERITED FEATURES OF A CLASS



(Like the previous one, you may skip this last section on first reading.)

The final prize we earn from all the work done in this chapter is the ability to provide a precise, conclusive definition of a key notion: the features of a class — in particular its inherited features.

As specified in the original discussion of features, the “features of a class” include its immediate features (those introduced in the class itself), and its inherited features, which were defined informally as the features “obtained from” the parents’ features. ← Chapter 5; see [“IMMEDIATE AND INHERITED FEATURES”, 5.4, page 133.](#)

The reason for being informal at that earlier stage is now clear: two mechanisms, repeated inheritance and join, affect how a class may “obtain” features from its parents. Without these mechanisms, every feature from a parent (every **precursor**) would yield one feature in the heir. But:

- The **join** mechanism merges two or more features from parents into a single one in their common heir.
- With **sharing** under repeated inheritance, two or more precursors, inherited from different parents but coming from the same features of a common ancestor, yield a single feature of D .
- Conversely, with **replication** under direct repeated inheritance (D has two or more **Parent** clauses listing the same parent), a single precursor may yield two or more features of D .

Only with the benefit of these observations can we now obtain a precise definition of the “inherited features of a class”, and hence (since immediate features — the new, non-inherited ones — raise no particular problem) of the **features of a class**. Here is the full definition:



Inherited features

Let D be a class. Let *precursors* be the list obtained by concatenating the lists of features of every parent of D ; this list may contain duplicates in the case of repeated inheritance. The list *inherited* of **inherited features** of D is obtained from *precursors* as follows:

- 1 • In the list *precursors*, for any set of two or more elements representing features that are repeatedly inherited in D under the same name, so that the Repeated Inheritance rule yields sharing, keep only one of these elements. The Repeated Inheritance Consistency constraint (sharing case) indicates that these elements must all represent the same feature, so that it does not matter which one is kept.
- 2 • For every feature f in the resulting list, if D undefines f , replace f by a deferred feature with the same signature, specification and header comment.
- 3 • In the resulting list, for any set of deferred features with the same final name in D , keep only one of these features, with assertions and header comment joined as per the Join Semantics rule. (Keep the signature, which the Join rule requires to be the same for all the features involved after possible redeclaration.)
- 4 • In the resulting list, remove any deferred feature such that the list contains an effective feature with the same final name. (This is the case in which a feature f , inherited as effective, effects one or more deferred features: of the whole group, only f remains.)
- 5 • All the features of the resulting list have different names; they are the inherited features of D in their parent forms. From this list, produce a new one by replacing any feature that D redeclares (through redefinition or effecting) with the result of the redeclaration, and retaining any other feature as it is.
- 6 • The result is the list *inherited* of inherited features of D .

← “Join Semantics rule”, page 320.

This definition looks a little like an algorithm, but it's not; you may view it as a plain mathematical specification. There is no requirement that compilers implement the corresponding mechanisms by mimicking the rule's successive steps, as long as the result is compatible.

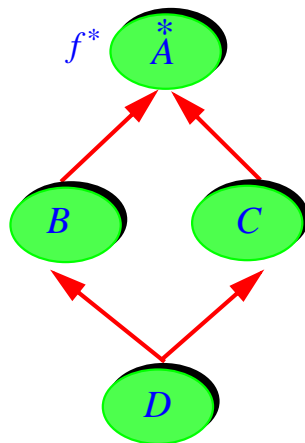
The order of the clauses is significant. Note in particular that the very first step, clause 1, takes care once and for all of repeated inheritance. This removes a small potential ambiguity, which we may remove through a semantic rule (not a new property, just a consequence of the preceding definition):



Join-Sharing Reconciliation rule

If a class inherits two or more features satisfying both the conditions of sharing under the Repeated Inheritance rule and those of the Join rule, the applicable semantics is the Repeated Inheritance rule.

The situation is illustrated by the figure below: f is deferred at the level of A , and nothing else — renaming, effecting ... — happens to it down to the level of D . It's a case of sharing under repeated inheritance, but we might also apply the Join semantics, as always when a class inherits under a single name a set of features, all deferred (or, although this doesn't apply here, all deferred except one). You may have wondered about this case: which of the two semantic rules should we apply? You may also have brushed off the question: does it matter at all?



* Deferred

Join, or sharing?

It matters not much, but it matters just a little and we must leave no semantic stone unturned. The only difference has to do with assertions. Assume that *f*, deferred as it may be, has a postcondition

```
ensure
  your_condition
```

Then the Join Semantics rule prescribes combining the header comments of the joined features, and also their assertions: through an **or** for the preconditions, and an **and** for postcondition. Because *a and a* has the same value as *a*, no really bad semantic consequence will follow, but for example a class documentation tool, such as a flat-short form displayer, might mistakenly display the postcondition of *f* in *D* as something like:

← “Join Semantics rule”, page 320.

← “Contract view, flat-short form”, page 216

```
ensure
  -- From A:
  your_condition
  and
  -- From A:
  your_condition
```

Not a disaster, but unnecessarily complex. The Join-Sharing Reconciliation rule explicitly defines the resulting postcondition in such a case to be just *your_condition*, with a similar consequence for preconditions and header comments.



Let’s come back to more general properties of the definition of Inherited Features. To understand the definition, note that the lists under consideration are lists of **features**, not of feature names, although the features that remain at the end all have different final names in *D*. The list *inherited* obtained at step 6 of the definition may still contain duplicate features — with different feature names — as a result of repeated inheritance with replication. This is why we define *precursors* as a list rather than a set. (Unlike a set, a list may contain duplicates.)

In fact these observations also yield a new definition of the “precursors” of a feature, equivalent to the original one but more precise:

← “*Precursor (joined features)*”, page 315. See also the first, simplified definition on page 268.



Precursor

A **precursor** of an inherited feature of final name $fname$ is any parent feature — appearing in the list *precursors* obtained through case 1 of the definition of “Inherited features” — that the feature mergings resulting from the subsequent cases reduce into a feature of name $fname$.

In accordance with this definition the successive steps of the definition of “inherited features” may only merge features — elements of the list *precursors* — if they all have the same final name. This is an important property because without it the earlier definition of the final name of an inherited feature would not make sense.

← “*Final name, extended final name, final name set*”, page 186.

Recall that according to this definition the final name m of a feature f obtained from a precursor of name n in a parent B is:

- n in the absence of renaming.
- Otherwise, the m appearing in a **Rename_pair** of the form **rename n as m** in the **Parent** clause for B in D .

Obviously, if f is obtained from two or more precursors, all this is meaningless unless we are sure that m is the same for all these precursors.

This also clarifies the notion of **final name set** of a class, originally introduced — in the same definition as “final name” — as the set of final names of all the features of a class. These final names are:

← “*Final name, extended final name, final name set*”, page 186.

- For immediate features, the names under which the class declares them.
- For inherited features, the inherited names except as overridden by renaming.

Two or more precursors merged into one — because of either a join or sharing under repeated inheritance — yield just one element of the final name set. If a feature from a repeated ancestor yields several features under replication, this adds all the corresponding names to the final name set.

Both the Repeated Inheritance rule and the Join rule require all the merged features to have the same final name.

Finally, we introduce a simple constraint capturing the fundamental rule on choosing feature names:



Feature Name rule

VMFN

It is valid for a feature f of a class C to have a certain final name if and only if it satisfies the following conditions:

- 1 • No other feature of C has that same feature name.
- 2 • If f is shared under repeated inheritance, its precursors all have either no Alias or the same alias.

Condition 1 follows from other rules: the Feature Declaration rule, the Redeclaration rule and the rules on repeated inheritance. It is convenient to state it as a separate condition, as it can help produce clear error messages in some cases of violation.

Two feature names are “the same” if the lower-case version of their identifiers is the same.

← “Same feature name, same operator, same alias”, page 153.
 ← “Inherited features”, page 470.



The important notion in this condition is “**other feature**”, resulting from the above definition of “inherited features”. When do we consider g to be a feature “other” than f ? This is the case whenever g has been declared or redeclared distinctly from f , unless the definition of inherited features causes the features to be merged into just one feature of C . Such merging may only happen as a result of sharing features under repeated inheritance, or of joining deferred features.

Also, remember that if C redeclares an inherited feature (possibly resulting from the joining of two or more), this does not introduce any new (“other”) feature. This was explicitly stated by the definition of “introducing” a feature.

← “Inherited, immediate; origin; redeclaration; introduce”, page 133

Condition 2 complements these requirements by ensuring that sharing doesn’t inadvertently give a feature more than one alias.

The Feature Name rule crowns the discussion of inheritance and feature adaptation by unequivocally implementing the No Overloading Principle: no two features of a class may have the same name. The only permissible case is when the name clash is apparent only, but in reality the features involved are all the same feature under different guises, resulting from a join or from sharing under repeated inheritance.

Consequences of the Feature Name rule includes the following properties, which for convenience we may group into a new constraint:



Name Clash rule

VMNC

The following properties govern the names of the features of a class C :

- 1 • It is invalid for C to introduce two different features with the same name.
- 2 • If C introduces a feature with the same name as a feature it inherits as effective, it must rename the inherited feature.
- 3 • If C inherits two features as effective from different parents and they have the same name, the class must also (except under sharing for repeated inheritance) remove the name clash through renaming.

WARNING: not a validity constraint in the usual form; see comment at bottom of preceding page.



This is not a new constraint but a set of properties that follow from the Feature Name rule and other rules. Instead of Eiffel's customary "This is valid if and only if ..." style, more directly useful to the programmer since it doesn't just tell us how to mess things up but also how to produce guaranteeably *valid* software, the Name Clash rule is of the more discouraging form "You may not validly write ...". It does, however, highlight frequently applicable consequences of the naming policy, and compilers may take advantage of it to report naming errors.

Control structures

17.1 OVERVIEW

The previous discussions have described the “bones” of Eiffel software: the module and type structure of systems. Here we begin studying the “meat”: the elements that govern the execution of applications.

Control structures are the constructs used to schedule the run-time execution of instructions. There are four of them: sequencing (compound), conditional, multi-branch choice and loop. A complementary construct is the **Debug** instruction.



As made clear by the definition of “non-exception semantics” in the semantic rule for **Compound**, which indirectly governs all control structures (since all instructions are directly or indirectly part of a **Compound**), the default semantics assumes that none of the instructions executed as part of a control structure triggers an *exception*. If an exception does occur, the normal flow of control is interrupted, as described by the rules of exception handling in the discussion of this topic.

→ [Chapter 26](#).

17.2 COMPOUND

The first control structure, **Compound**, enables you to specify a list of instructions to be executed in a specified order.

From its inconspicuous syntax, you wouldn’t guess that this is a fundamental program composition mechanism: the instructions of a **Compound** are just written one after another, in the order of their intended execution. You may emphasize the sequencing of the instructions by using a separator, the semicolon, which is not only discreet but optional to boot.

A typical specimen of the **Compound** construct is:



```

window1.display
mouse.wait_for_click (middle)
if not last_event.is_null then
    last_event.handle; screen.refresh
end

```

This **Compound** is made of three instructions; it specifies the execution of these instructions in the order given. The last of the three (a **Conditional** instruction, as studied below) itself includes a two-instruction **Compound**.



The use and non-use of semicolons in this example illustrate the recommended style convention: no semicolon has been included between the three instructions of the outermost **Compound** since they appear on separate lines (the most common case), enough to remove any confusion. The two instructions of the innermost **Compound** — inside the **Conditional** — appear on the same line; here the semicolon should be included for the benefit of the human reader, even though compilers don't need it.

The syntax for **Compound** specified:

Compound \triangleq {**Instruction** ";" ...}*



In the common, non-confusing case, the style rule is to **omit the semicolons** between instructions appearing on separate lines. The semicolon in that case is just visual noise and actually hampers readability. For successive instructions on the same line make sure to **keep** the semicolon. The above example illustrated this style rule, observed throughout this book. → "OPTIONAL SEMICOLONS", 34.10, page 919.

All this does not diminish the role of sequencing as a control structure, even if the only syntactical trace left in the software text is the textual order of instructions, indicating the temporal order in which they should be executed at run time.

There is no validity rule for **Compound**. The semantic specification follows from the above explanations:

Compound (non-exception) semantics

The effect of executing a **Compound** is:

- If it has zero instructions: to leave the state of the computation unchanged.
- If it has one or more instructions: to execute the first instruction of the **Compound**, then (recursively) to execute the **Compound** obtained by removing the first instruction.

This specification, the **non-exception semantics** of **Compound**, assumes that no exception is triggered. If the execution of any of the instructions triggers an exception, the Exception Semantics rule takes effect for the rest of the **Compound**'s instructions.

Less formally, this means executing the constituent instructions in the order in which they appear in the **Compound**, each being started only when the previous one has been completed.

Note that a **Compound** can be empty, in which case its execution has no effect. This is useful for examples when refactoring the branches of a **Conditional**: you might temporarily remove all the instructions of the **Else_part**, but not the **Else_part** itself yet as you think it may be needed later.

Aside from its role as a control structure, the **Compound** construct serves an frequent syntactical need : allowing any construct that involves an instruction — so that it may execute it as part of its own execution — to involve *any number* of instructions, including zero. The syntax of Eiffel consistently adheres to this rule: **Instruction** never appears in the definition of a construct other than **Compound**; other construct definitions use **Compound** instead. They include:

- The body of a non-deferred routine (construct **Internal**). ← Syntax on page [222](#).
- The **Then_part** and **Else_part** of a **Conditional** instruction. → Page [481](#).
- The **When_part** and **Else_part** of a **Multi_branch** instruction. → Page [481](#).
- The **Initialization** and **Loop_body** of a **Loop** instruction. → Page [495](#).
- The **Debug** instruction. → Page [498](#).
- The **Rescue** clause of a non-deferred routine. → Page [701](#).

17.3 CONDITIONAL

A basic algorithmic mechanism is the ability to discriminate between a set of values, executing a different set of instructions in each case. Eiffel provides three variants of this notion: **Conditional**, where discriminating criteria are boolean conditions; **Multi_branch**, comparing an expression to a set of specified values; and **Object_test**, matching a reference against a specified object type. They're studied in this section and the next two.

A **Conditional** instruction prescribes execution of one among a number of possible compounds, the choice being made through boolean conditions associated with each compound.

You should remain alert to an important aspect of the Eiffel method, which de-emphasizes explicit programmed choices between a fixed set of alternatives, in favor of automatic selection at run-time based on the type of the objects to which an operation may be applied. Such an automatic selection is achieved by the object-oriented techniques of inheritance and dynamic binding. This methodological guideline, discussed in more detail [below](#), does not diminish the usefulness of **Conditional** instructions — a widely used mechanism — but should make you wary of complicated decision structures with too many **elseif** branches. This applies even more to the **Multi_branch** instruction studied next.

→ "[USING SELECTION INSTRUCTIONS PROPERLY](#)", 17.6, page [491](#).

An example **Conditional** is

```

if  $x > 0$  then
     $i1; i2$ 
elseif  $x = 0$  then
     $i3$ 
else
     $i4; i5; i6$ 
end

```

whose execution is one among the following: execution of the compound $i1; i2$ if $x > 0$ evaluates to true; execution of $i3$ if the first condition does not hold and $x = 0$ evaluates to true; execution of $i4; i5; i6$ if none of the previous two conditions holds.



PURPOSE



METHOD

CLASS



PLAN

There may be zero or more “**elseif Compound**” clauses. The “**else Compound**” clause is optional; if it is absent, no instruction will be executed when all boolean conditions are false.

The general form of the construct is



Conditionals

Conditional \triangleq **if** Then_part_list [Else_part] **end**
 Then_part_list \triangleq {Then_part **elseif** ...}⁺
 Then_part \triangleq Boolean_expression **then** Compound
 Else_part \triangleq **else** Compound

Two auxiliary notions help define precisely the semantics of this construct. As the syntax specification shows, a **Conditional** begins with

```
if condition1 then compound1
```

where *condition₁* is a boolean expression and *compound₁* is a **Compound**. The remaining part may optionally begin with **elseif**. If so, we may consider that it forms a new, simpler **Conditional**, called its *secondary part*:



Secondary part

The **secondary part** of a **Conditional** possessing at least one **elseif** is the **Conditional** obtained by removing the initial “**if Then_part_list**” and replacing the first **elseif** of the remainder by **if**.

The secondary part of the above example **Conditional** is



```
if x=0 then  
    i3  
else  
    i4; i5; i6  
end
```

The other useful notion is “prevailing immediately”:

Prevailing immediately

The execution of a **Conditional** starting with **if condition₁** is said to **prevail immediately** if *condition₁* has value true.

These conventions enable a simple definition of the semantics:



Conditional semantics

The effect of a **Conditional** is:

- If it prevails immediately: the effect of the first **Compound** in its **Then_part_list**.
- Otherwise, if it has at least one **elseif**: the effect (recursively) of its secondary part.
- Otherwise, if it has an **Else** part: the effect of the **Compound** in that **Else** part.
- Otherwise: no effect.



Like the instruction studied next, the **Conditional** is a “multi-branch” choice instruction, thanks to the presence of an arbitrary number of **elseif** clauses. These branches do not have equal rights, however; their conditions are evaluated in the order of their appearance in the text, until one is found to evaluate to true. If two or more conditions are true, the one selected will be the first in the syntactical order of the clauses.

17.4 MULTI-BRANCH CHOICE



Like the conditional, the **Multi_branch** supports a selection between a number of possible instructions. In contrast with the **Conditional**, however, the order in which the branches are written does not influence the effect of the instruction. Indeed, the validity constraints seen below guarantee that at most one of the selecting conditions may evaluate to true.



Like the **Conditional**, the **Multi_branch** instruction is less commonly used in proper Eiffel style than its counterparts in traditional design and programming languages. This is explained in more detail below.

You may use a **Multi_branch** if the conditions are all of the form

“Is *exp* equal to v_i ?”

or all of the form

“Is *exp* of type T_i ?”

where *exp* is an expression, the same for every branch, the v_i are constant values, different for each branch and (in the second variant) the T_i are all distinct types, not conforming to one another. In such cases, the **Multi_branch** provides a more compact notation than the **Conditional**, and makes a more efficient implementation possible.

→ “USING SELECTION INSTRUCTIONS PROPERLY”, 17.6, page 491.



Here is an example of the first kind, assuming an entity *last_input* of type *CHARACTER*:

```
inspect
    last_input
when 'a' .. 'z', 'A' .. 'Z', '_' then
    command_table.item (upper(last_input)).execute
    screen.refresh
when '0' .. '9' then
    history.item (last_input).display
when Control_L then
    screen.refresh
when Control_C, Control_Q then
    confirmation.ask
    if confirmation.ok then
        cleanup; exit
    end
else
    display_proper_usage
end
```

Depending on the value of *last_input*, this instruction selects and executes one *Compound* among five possible ones. It selects the first (*command_table...*) if *last_input* is a lower-case or upper-case letter, that is to say, belongs to one of the two intervals '*a* .. '*z*' and '*A* .. '*Z*', or is an underscore '*_*'. It selects the second if *last_input* is a digit. It selects the third (refresh the screen) for the character *Control_L*, and the fourth (exit after confirmation) for either one of two other control characters; here *Control_L*, *Control_C* and *Control_Q* must be constant attributes. In all other cases, the instruction executes the fifth compound given (*display_proper_usage*).

This example discriminates on the value of an expression of type *CHARACTER*. Other permitted types include: *INTEGER*; *STRING*; and *TYPE [G]* for some *G*, which describe object types (conforming to *G*). This last possibility allows you to discriminate on the basis of the *type* of the object attached at run time to the value of an arbitrary expression, as illustrated by the following example of dealing with various kinds of exception object:



```

inspect
    last_exception.type
when {DEVELOPER_EXCEPTION} then
    process_developer_exception
when {OS_SIGNAL}, {NO_MORE_MEMORY} then
    cancel_operation
else
    reset
end

```

In this form the “inspect values” — the values listed in the **when** parts — are type descriptors, each listing a type in braces, as `{OS_SIGNAL}`. The instruction examines the type of the object associated with *last_exception*, as given by *last_exception.type*, and if it conforms to one of the types listed executes the corresponding **then** branch; otherwise the instruction executes its **else** branch. The validity rule requires that none of the types listed conform to another, so there can be no ambiguity as to which branch will be executed.

The expression that determines the choice — *last_input* and *last_exception.type* in these two examples — has a name:



Inspect expression

The **inspect expression** of a **Multi_branch** is the expression appearing after the keyword **inspect**.

The inspect expressions of the last two examples are *last_choice* and *last_exception*. The inspect expression may only be of one of the types *CHARACTER*, *INTEGER*, *STRING*, *TYPE*.

The instruction includes one or more **When_part**, each giving a list of one or more **Choice**, separated by commas, and a **Compound** to be executed when the value of the inspect expression is one of the given **Choice** values.

Every **Choice** specifies zero or more inspect values. More precisely, a **Choice** is either a single constant (**Manifest_constant** or constant attribute) or an interval of consecutive constants yielding all the interval’s elements as inspect values. If present, the instruction’s optional **Else_part** is executed when the inspect expression is not equal to any of the inspect values.

As the validity constraint will state precisely, all the inspect values must all be of the same type as the inspect expression: all characters, all integers, all strings or all types. They must all be different, and non-conforming in the case of types; this avoids ambiguity, ensuring that the order of the **When_part** branches has no influence on the semantics of the construct.

Every constant in the preceding examples is either a **Manifest_type**, a **Manifest_constant** such as 'a' whose value is an immediate consequence of the way it is written, or a constant attribute such as *Control_L* whose value is given in a constant attribute declaration such as

```
Control_L: CHARACTER is '%/217'
```

→ On character codes such as *"/217"* see [32.14, page 894](#).

Now the formal rules. First, the syntax of **Multi_branch**:



Multi-branch instructions	
Multi_branch	\triangleq inspect Expression [When_part_list] [Else_part] end
When_part_list	\triangleq When_part ⁺
When_part	\triangleq when Choices then Compound
Choices	\triangleq {Choice ", " ...} ⁺
Choice	\triangleq Constant Manifest_type Constant_interval Type_interval
Constant_interval	\triangleq Constant ".." Constant
Type_interval	\triangleq Manifest_type ".." Manifest_type

Construct **Constant** describes manifest or symbolic constants and is studied in "[GENERAL FORM OF CONSTANTS](#)", 29.2, page 787.

Interval

An **interval** is a **Constant_interval** or **Type_interval**.

To discuss the constraint and the semantics, it is convenient to consider the *unfolded form* of the instruction. First, constant and type intervals have similar properties, justifying a general term:

which enables us to define the unfolded form

DEFINITION

Unfolded form of a multi-branch

To obtain the **unfolded form** of a **Multi_branch** instruction, apply the following transformations in the order given:

- 1 • Replace every constant inspect value by its manifest value.
- 2 • If the type T of the inspect expression is any sized variant of **CHARACTER**, **STRING** or **INTEGER**, replace every inspect value v by $\{T\} v$.
- 3 • Replace every interval by its unfolded form.

Step 2 enables us, with an inspect expression of a type such as **INTEGER_8**, to use constants in ordinary notation, such as **1**, rather than the heavier $\{\text{INTEGER_8}\} 1$. Unfolded form constructs this proper form for us. The rules on constants make this convention safe: a value that doesn't match the type, such as **1000** here, will cause a validity error. → ---- [Add reference]

The last unfolded form is based on another, for intervals:

DEFINITION

Unfolded form of an interval

The **unfolded form** of an interval $a..b$ is the following (possibly empty) list:

- 1 • If a and b are constants, both of either a character type, a string type or an integer type, and of manifest values va and vb : the list made up of all values i , if any, such that $va \leq i \leq vb$, using character, integer or lexicographical order respectively.
- 2 • If a and b are both of type **TYPE [T]** for some T , and have manifest values va and vb : the list containing every **Manifest_type** of the system conforming to vb and to which va conforms.
- 3 • If neither of the previous two cases apply: an empty list.

The “manifest value” of a constant is the value that has been declared for it, ignoring any **Manifest_type**: for example both `1` and `{INTEGER_8} 1` have the manifest value 1. → ---- [Add reference]

The symbol `..` is not a special symbol of the language but an alias for a feature of the Kernel Library class `PART_COMPARABLE`, which for any partially or totally ordered set and yielding the set of values between a lower and an upper bound. Here, the bounds must be constant. → In the Kernel Library specifications see classes

[“PART_COMPARABLE”, page 977](#), and [“INTERVAL”, page 981](#).



A note for implementers: type intervals such as `{U}..{T}`, denoting all types conforming to `T` and to which `U` conforms, may seem to raise difficult implementation issues: the set of types, which the unfolded form seems to require that we compute, is potentially large; the validity (Multi-Branch rule) requires that all types in the unfolded form be distinct, which seems to call for tricky computations of intersections between multiple sets; and all this may seem hard to reconcile with incremental compilation, since a type interval may include types from both our own software and externally acquired libraries, raising the question of what happens on delivery of a new version of such a library, possibly without source code. Closer examination removes these worries:

- There is no need actually to compute entire type intervals as defined by the unfolded form. Listing `{U}..{T}` simply means, when examining a candidate type `Z`, finding out whether `Z` conforms to `T` and `U` to `Z`.
- To ascertain that such a type interval does not intersect with another `{Y}..{X}`, the basic check is that `Y` does not conform to `T` and `U` does not conform to `X`.
- If we add a new set of classes and hence types to a previously validated system, a new case of intersection can only occur if either: a new type inherits from one of ours, a case that won’t happen for a completely external set of reusable classes and, if it happens, should require re-validating since existing **Multi_branch** instructions may be affected; or one of ours inherits from a new type, which will happen only when we modify our software *after* receiving the delivery, and again should require normal rechecking.

An interval may not be empty:



Interval rule

VOIN

An **Interval** is valid if and only if its unfolded form is not empty.

So of the intervals



```
3 .. 5
'i' .. 'n'
"ab" .. "ad"
5 .. 3
```

the first two unfold into

```
3, 4, 5
'i', 'j', 'k', 'l', 'm' 'n'
```

the third into the (infinite) set of strings lexicographically between "ab" and "ad", and the last into an empty **Choices** list. Thanks to unfolding, the constraint and semantics may limit themselves to the case of **Multi_branch** instructions where every **Choice** is a **Constant** or **Manifest_type**.

This definition also enables us to say exactly what “inspect values” means:

Inspect values of a multi-branch

The **inspect values** of a **Multi_branch** instruction are all the values listed in the **Choices** parts of the instruction's unfolded form.



The set of inspect values may be infinite in the case of a string interval, but this poses no problem for either programmers or compilers, meaning simply that matches will be determined through lexicographical comparisons.

A **Multi_branch** must satisfy a validity constraint --- DEFINE CONSTANT MANIFEST TYPE ---:



Multi-branch rule

VOMB

A **Multi_branch** instruction is valid if and only if its unfolded form satisfies the following conditions.

- 1 • Inspect values are all valid.
- 2 • Inspect values are all constants.
- 3 • The manifest values of any two inspect values are different.
- 4 • If the inspect expression is of type *TYPE [T]* for some type *T*, all inspect values are types.
- 5 • If case 4 does not apply, the inspect expression is one of the sized variants of *INTEGER*, *CHARACTER* or *STRING*.

--- IN CLAUSE 2: CHECK THAT DEFINITION OF CONSTANT” FOR TYPES ONLY COVERS CONSTANT TYPES ----

The clauses guarantee that there won't be any ambiguity for choosing the branch to be executed, if any.

--- NOT TRUE ANY MORE, FIX THIS --- For inspect values of the `Manifest_type` kind, such as `{SOME_TYPE}`, clause 4 requires that none of the types listed conform to another. It rules out examples such as



```
inspect
  last_exception
when {YOUR_DEVELOPER_EXCEPTION} then
  "Something"
when {DEVELOPER_EXCEPTION} then
  "Something else"
end
```

WARNING: invalid with the assumed inheritance link.



where the class `YOUR_DEVELOPER_EXCEPTION` inherits from `DEVELOPER_EXCEPTION`. This may appear too strong a constraint until you realize that giving non-ambiguous semantics to such examples would require that we take into account the order of the `When_part` clauses: the rule, presumably, would be to select the first one that matches. This conflicts with the principle stating that the semantics of a `Multi_branch` should never depend on the order of the `when` clauses.

If you do want type-based discrimination with more than one possibly matching type, nest `Multi_branch` instructions, or use a `Conditional` or `Object_conditional`.

To define the semantics of a `Multi_branch` instruction, we will use the concept of matching branch:



Matching branch

During execution, a **matching branch** of a `Multi_branch` is a `When_part wp` of its unfolded form, satisfying either of the following for the value `val` of its inspect expression:

- 1 • `val ~ i`, where `i` is one of the non-`Manifest_type` inspect values listed in `wp`.
- 2 • `val` denotes a `Manifest_type` listed among the choices of `wp`.

The Multi-branch rule is designed to ensure that in any execution there will be at most one matching branch.

In case 1, we look for object equality, as expressed by \sim . Strings, in particular, will be compared according to the function *is_equal* of *STRING*. A void value, even if type-wise permitted by the inspect expression, will never have a matching branch.

In case 2, we look for an exact type match, not just conformance. For conformance, we have type intervals: to match types conforming to some *T*, use $\{NONE\}.. \{T\}$; for types to which *T* conforms, use $\{T\}.. \{ANY\}$.

Case 1 applies to a **Multi_branch** that lists actual inspect values: integers, characters or strings. The matching criterion is equality in the sense of *equal*. → "*OBJECT EQUALITY*", 21.6, page 580

Case 2 covers a **Multi_branch** that discriminates on the type of an object attached to the value of an expression. Note that a void value will never have a matching branch.

The specification of a **Multi_branch**'s effect follows directly from this definition.



Multi-Branch semantics

Executing a **Multi_branch** with a matching branch consists of executing the **Compound** following the **then** in that branch. In the absence of matching branch:

- 1 • If the **Else_part** is present, the effect of the **Multi_branch** is that of the **Compound** appearing in its **Else_part**.
- 2 • Otherwise the execution triggers an exception of type ***BAD_INSPECT_VALUE***.

→ See 26.12, page 709, about exception objects.



Note the difference between the semantics of **Conditional** and **Multi_branch** when there's no **Else_part** and none of the selection conditions holds:

- A **Conditional** just amounts to a null instruction in this case
- **Multi_branch** will **fail**, triggering an exception.

The reason is a difference in the nature of the instructions. A **Conditional** tries a number of possibilities in sequence until it finds one that holds. A **Multi_branch** selects a **Compound** by comparing the value of an expression with a fixed set of constants; the **Else_branch**, if present, catches any other values.

If you expect such values to occur and want them to produce a null effect, you should use an `Else_part` with an empty `Compound`. By writing a `Multi_branch` without an `Else_part`, you state that you do *not* expect the expression ever to take on a value not covered by the inspect values. If your expectations prove wrong, the effect is to trigger an exception — not to smile, do nothing, and pretend that everything is proceeding according to plan.

17.5 OBJECT TEST

--- SECTION REMOVED, BUT MATERIAL WILL BE REUSED FOR NEW MECHANISM REPLACING ASSIGNMENT ATTEMPT S----

17.6 USING SELECTION INSTRUCTIONS PROPERLY



If you have accumulated some experience with some of the traditional design or programming languages, many of which include a "case" or "switch" instruction, you will recognize the `Multi_branch` as similar in syntax and semantics. Similarly, the `Object_test` may remind you of techniques for discriminating between cases based on the type of an object, sometimes known as “Run-Time Type Identification” or RTTI. But when it comes to writing Eiffel applications, you should be careful to not misuse these instructions. This warning extends to `Conditional` instructions with many branches.

Staying away from explicit discrimination is an important part of the Eiffel approach to software construction. When a system needs to execute one of several possible actions, the appropriate technique is usually not an explicit test for all cases, as with `Multi_branch` or `Conditional`, but a more flexible inheritance-based mechanism: **dynamic binding**. With explicit tests, every discriminating software element must list all the available choices — a dangerous practice since the evolution of a software project inevitably causes choices to be added or removed. Dynamic binding avoids this pitfall.

→ [“DYNAMIC BINDING”, 23.12, page 638.](#)

You should reserve `Multi_branch` instructions, then, to simple situations where a single operation depends on a fixed set of well-understood choices.

When the purpose is to apply a different operation to an object depending on its type (for example categories of employees, for which a certain operation, such as paying the salary, has a different effect), then `Multi_branch` is not appropriate: instead, you should define different classes that inherit from a common ancestor — for example `MANAGER`, `ENGINEER` etc. all inheriting from `EMPLOYEE` — and redefine one or more features (such as `pay_salary`) to take care of the local context. Then dynamic binding guarantees application of the proper variant: the call

Caroline.pay_salary

will automatically use the variant of *pay_salary* adapted to the exact type of the object attached to *Caroline* at run time (which may be an instance of *MANAGER*, or *ENGINEER* etc.).

This is more flexible than a **Conditional** or **Multi_branch** that lists the choices explicitly, especially if other operations besides *pay_salary* have variants for the given categories. To add a variant, it suffices to write a new class, say *INTERN*, as a descendant *EMPLOYEE*, equipped with new versions of the operations that differ from the default *EMPLOYEE* version. Unlike a system that makes explicit choices through **Conditional** or **Multi_branch** instructions, a system built with this method will only have to undergo minimal change for such an extension.

Explicit choices do have a role, as illustrated by the earlier examples of **Multi_branch**. The first read



```

inspect
  last_input
when 'a' .. 'z', 'A' .. 'Z', '_' then
  command_table.item (upper(last_input)).execute
  screen.refresh
when '0' .. '9' then
  history.item (last_input).display
when Control_L then
  screen.refresh
when Control_C, Control_Q then
  confirmation.ask
  if confirmation.ok then
    cleanup; exit
  end
else
  display_proper_usage
end

```

This decodes a user input consisting of a single character and executes an action depending on that character. What is interesting is that the **Multi_branch** does only the “easy” part: separating the major categories of characters (letters, digits, control characters).

In the branches for letters and characters, however, the finer choice is made not through explicit instructions but through dynamic binding. For example, letters are used to index a table *command_table* of objects representing command objects with operations such as *execute*. (These objects might be *agents* as studied in a later chapter.) After retrieving the command object associated with the upper-case version of a given letter, the above *Multi_branch* applies *execute* to it, relying on dynamic binding to ensure that the proper action will be selected.

→ *Agents are the topic of chapter 27.*

Using a *Multi_branch* to discriminate between the actions associated with individual letters 'A', 'B' etc. would have resulted in a more complicated and inflexible architecture. At the outermost level, however, the above extract does use a *Multi_branch*, which appears justified because of the small number of cases involved and the diversity of actions in each case, which do not fall into a single category such as “execute the command attached to the selected object”.

See also the Single Choice principle in “Object-Oriented Software Construction”, and, in the present book, “Single choice and factory objects”, page 537.

The second example used *Manifest_type* inspect values:



```
inspect
  last_exception.type
when {DEVELOPER_EXCEPTION} then
  process_developer_exception
when {OS_SIGNAL}, {NO_MORE_MEMORY} then
  cancel_operation
else
  reset
end
```

Even though we are using a *Multi_branch* to select different actions depending on the type of an object, we are not doing anything else with the object in question. The choices, in addition, are from a fixed set of possibilities — exception types — provided by the Kernel Library, not under developer control.

If you do anything else with the inspected object, however, *Multi_branch* will cease to be the better choice and you should look into dynamic binding and associated mechanisms.

17.7 LOOP



The next control structure is the only construct (apart from recursive routine calls) allowing iteration. This is the **Loop** instruction, describing computations that obtain their result through successive approximations.

Loop structure and properties

The following example of a search routine illustrates the **Loop** construct with all possible clauses:



```

search_same_child (sought: like first_child)
  -- Move cursor to first child position where sought
  appears
  -- at or after current position.
  -- If no such position, move cursor after last item.
  require
    sought_child_exists: sought /= Void
  do
    from
      child_start
    invariant
       $0 \leq \textit{position}$ 
       $\textit{position} \leq \textit{arity} + 1$ 
    until
      child_off or else (sought = child)
    loop
      child_forth
    variant
       $\textit{arity} - \textit{child\_position} + 1$ 
    end
  ensure
    (not child_off) implies (sought = child)
  end

```

This example is close to actual tree searching routines in EiffelBase. Actual versions, however, can check for equal as well as '='.

The **Loop** construct extends from the keyword **from** to the first **end**.

The **Initialization** clause (**from**...) introduces actions, here a call to procedure *child_start*, to be executed before the actual iteration starts. The **Loop_body** (**loop**...) introduces the instruction to be iterated, here a call to *child_forth*; this will be executed zero or more times, after the **Initialization**, until the **Exit** condition, introduced in the **until**... clause, is satisfied.

The optional **Invariant** and **Variant** clauses help reason about a loop, ascertain its correctness, and debug it:

← "[LOOP INVARIANTS AND VARIANT](#)", 9.11, page 250.

- The keyword **invariant** introduces an assertion, describing a property that must be satisfied by the initialization and maintained by every execution of the loop body if the exit condition is not satisfied.
- The keyword **variant** introduces an integer expression which must be non-negative after the initialization and will decrease whenever the body is executed, but will remain non-negative; these properties ensure that the loop's execution terminates.

Here is the general form of the **Loop** construct.



Loops

```

Loop ≙ Initialization
      [Invariant]
      Exit_condition
      Loop_body
      [Variant]
      end

Initialization ≙ from Compound
Exit_condition ≙ until Boolean_expression
Loop_body ≙ loop Compound
  
```

← *Invariant and Variant were studied in [9.11](#).*

The **Initialization** (**from** clause) is required. If you do not need any specific initialization, use a **from** clause with an empty **Compound**, as in



```

from
until
    printer.queue_empty
loop
    printer.process_next_job
end
  
```

In general, however, the **Initialization** does introduce a **Compound** of one or more instructions, as in this example from a list duplication routine in EiffelBase:



```

from
    mark
    Result.start
until
    off
loop
    Result.put (item)
    forth
    Result.forth
end

```

Loop semantics



Loop semantics

The effect of a **Loop** is the effect of executing the **Compound** of its **Initialization**, then its **Loop_body**.

The effect of executing a **Loop_body** is:

- If the **Boolean_expression** of the **Exit_condition** evaluates to true: no effect (leave the state of the computation unchanged).
- Otherwise: the effect of executing the **Compound** clause, followed (recursively) by the effect of executing the **Loop_body** again in the resulting state.



The optional **Invariant** and **Variant** parts have no effect on the execution of a correct loop; they describe correctness conditions. Their precise use was explained in the discussion of assertions and correctness. As a reminder:

- The **Invariant** must be ensured by the **Initialization**; any execution of the **Loop_body** started in a state where the **Invariant** is satisfied, but not the **Exit** condition, must produce a state that satisfies the **Invariant** again.
- The **Initialization** must produce a state where the **Variant** expression is non-negative; and any execution of the **Loop_body** started in a state where the **Variant** has a non-negative value v and the **Exit** condition is not satisfied must produce a state in which the **Variant** is still non-negative, but its new value is less than v . Since the **Variant** is an integer expression, this guarantees termination.

← “*LOOP INVARIANTS AND VARIANTS*”, 9.11, page 250.

Ensuring non-void references in a loop

--- [SECTION REMOVED, SOME MATERIAL WILL BE REUSED] ---

17.8 THE DEBUG INSTRUCTION

The **Debug** instruction serves to request the conditional execution of a certain sequence of operations, depending on a compilation option.

The existence of this instruction implies an obligation for Eiffel development environments to include a user option for turning “Debug mode” on and off and, more generally, to set a “Debug key”. The **Lace** control language includes the necessary mechanisms, enabling you to set the option at all relevant levels:

→ Appendix B discusses **Lace**; see [“SPECIFYING OPTIONS”, B.9, page 1028](#).

- Default for an entire system.
- Default for a cluster, overriding the system default.
- Value for a particular class, overriding the cluster default.

The basic form of a **Debug** instruction is



```
debug
  instruction1
  ...
  instructionn
end
```

The instruction will be ignored at execution time if the Debug option is off. If the option is on, the execution of the **Debug** instruction is the execution of all the *instruction_i* in the order given, as with a **Compound**.

A variant of the instruction enables you to exert finer control over the debugging level by specifying one or more “debug key” in the form of a **Manifest_string** in parentheses. For example:



```
debug ("GRAPHICS_DEBUG")
  instruction1
  ...
  instructionn
end
```

This will be executed if and only if the Debug option has been turned on either generally as before or specifically for the given **Debug_key**. This way you can exercise various parts of the software separately by playing with the option, typically in the **Ace file**, without touching the Eiffel text itself.

→ The **Ace** file is the **Lace** control file used to set options. See [appendix B](#).

Here is the syntax of the instruction:



Debug instructions

$\text{Debug} \triangleq \text{debug [("Key_list ")]}$
Compound **end**

Key_list was introduced in connection with the **Once** routine specification: ← Page [222](#).

$\text{Key_list} \triangleq \{\text{Manifest_string " , " ...}\}^+$



Debug semantics

A language processing tool must provide an option that makes its possible to enable or disable **Debug** instructions, both globally and for individual keys of a **Key_list**. Such an option may be settable for an entire system, or for individual classes, or both.

Letter case is not significant for a debug key.

The effect of a **Debug** instruction depends on the mode that has been set for the current class:

- If the **Debug** option is on generally, or if the instruction includes a **Key_list** and the option is on for at least one of the keys in the list, the effect of the **Debug** instruction is that of its **Compound**.
- Otherwise the effect is that of a null instruction.

Attributes

18.1 OVERVIEW

Attributes are one of the two kinds of feature.

← The other is routines, studied in chapter 8.

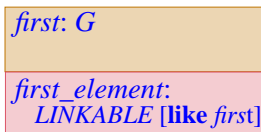
When, in the declaration of a class, you introduce an attribute of a certain type, you specify that, for every instance of the class that may exist at execution time, there will be an associated value of that type.

Attributes are of two kinds: **variable** and **constant**. The difference affects what may happen at run time to the attribute’s values in instances of the class: for a variable attribute, the class may include routines that, applied to a particular instance, will change the value; for a constant attribute, the value is the same for every instance, and cannot be changed at run time.

This chapter discusses the properties of both two kinds of attribute.

18.2 GRAPHICAL REPRESENTATION

In graphical system representations, you may mark a feature that you know is a variable attribute by putting its name in a box.



put_linkable_left:
→ **like** *first_element*
previous: **like** *first_element*
next: **like** *first_element*

Representing attributes

The figure illustrates this convention for attributes *first* and *first_element* in a class *LINKED_LIST* similar to the one from EiffelBase. (This is a partial representation of the class.)

As illustrated by this example, putting the attributes of a class next to each other, each boxed in a rectangle, yields a bigger rectangle that suggests the form of an **instance** of the class with all its fields. So we get a picture of both the class (elliptic) and the corresponding *objects* (rectangular).

→ Principle of uniform access: [23.4, page 624](#).



Not boxing a feature does not mean that it is not a attribute. In some cases, you may choose to leave unspecified whether a particular feature is an attribute or a routine. Then the standard representation for features, unboxed, is appropriate. In the example illustrated above, *previous* and *next* may be attributes just as well as functions without arguments.

18.3 VARIABLE ATTRIBUTES



Declaring a variable attribute in a class prescribes that every instance of the class should contain a field of the corresponding type. Routines of the class can then execute assignment instructions to set this field to specific values.

Here are some variable attribute declarations:

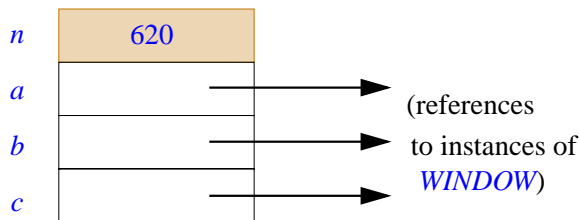


```
n: INTEGER
a, b, c: WINDOW
```

The first introduces a single attribute *n* of type *INTEGER*. The second (equivalent, because of the Multiple Declaration semantics rule, to three separate declarations) introduces three attributes, all of type *WINDOW*.

← “Unfolded form of a possibly multiple declaration”, [page 159](#).

If these declarations appear in the **Features** clause of a class *C*, all instances of *C* will have associated values of the corresponding types; an instance will look like this:



**An instance
with its fields**



More generally, as you may remember, a feature declaration is a variable attribute declaration if it satisfies the following conditions:

← “HOW TO RECOGNIZE FEATURES”, [5.12, page 145](#).

- There is no **Formal_arguments** part.
- There is a **Type_mark** part.
- There is no **Constant_or_routine** part.

18.4 ATTRIBUTES IN FULL FORM



Attribute bodies

Attribute \triangleq **attribute** Compound

The **Compound** is empty in most usual cases, but it is required for an attribute of an attached type (including the case of an expanded type) that does not provide *default_create* as a creation procedure; it will then serve to initialize the corresponding field, on first use for any particular object, if that use occurs prior to an explicit initialization. To set that first value, assign to **Result** in the **Compound**.

Such a **Compound** is executed at most once on any particular object during a system execution.

18.5 CONSTANT ATTRIBUTES



Declaring a constant attribute in a class associates a certain value with every instance of the class. Because the value is the same for all instances, it does not need to be actually stored in each instance.

Since you must specify the value in the attribute's declaration, the type of a constant attribute must be one for which the language offers a lexical mechanism to denote values explicitly. This means one of the following:

- **BOOLEAN**, with values written *True* and *False*.
- **CHARACTER**, with values written as characters in single quotes, such as 'A'.
- **INTEGER**, with values written using decimal digits possibly preceded by a sign, such as *-889*.
- **REAL**, with values such as *-889.72*.
- **STRING**, with values made of character strings in double quotes such as "A SEQUENCE OF \$CHARACTERS#".

The construct Constant_attribute is introduced in [29.2](#), [page 787](#), as part of the discussion of expressions.

*All these types except **STRING** are called basic types. See [page 338](#)*

All these examples use "manifest" constants; see below.



For types other than these, you may obtain an effect similar to that of constants by using a once function. For example, assuming a class → See [23.15, page 647](#), about the effect of calling a once function.

```

class COMPLEX creation
  make_cartesian, ...
feature -- Initialization
  make_cartesian (a, b: REAL)
    -- Initialize to real part a, imaginary part b.
  do
    x := a; y := b
  end
feature -- Access
  x, y: REAL
  ... Other features and invariant ...
end

```

you may, in another class, define the once function

```

i: COMPLEX is
  -- Complex number of real part 0, imaginary part 1
  once
  create Result, makecartesian (0, 1)
end

```

which creates a *COMPLEX* object on its first call; this call and any subsequent one return a reference to that object.

Returning to true constant attributes: the declaration of a constant attribute must determine the attribute's value, using a **manifest** constant.

The next section details this case.

18.6 CONSTANT ATTRIBUTES WITH MANIFEST VALUES

A *Manifest_constant* is a constant given by its explicit value. It may be a *Boolean_constant*, *Character_constant*, *Integer_constant*, *Real_constant* or *Manifest_string*. Chapter [32](#) describes the precise form of manifest constants.

Here are some constant attribute declarations using `Manifest_constant` values:



Terminal_count: INTEGER is 247
Cross: CHARACTER is 'X'
No: BOOLEAN is False
Height: REAL is 1.78
Message: STRING is "No such file"



More generally, a feature declaration is a constant attribute declaration if it satisfies the following conditions:

← ["HOW TO RECOGNIZE FEATURES"](#), 5.12, page 145.

- There is no `Formal_arguments` part.
- There is a `Type_mark` part.
- There is a `Constant_or_routine` part, which contains a `Manifest_constant`.

A straightforward validity constraint governs such declarations:



Manifest Constant rule

VQMC

A declaration of a feature f introducing a manifest constant is valid if and only if the `Manifest_constant` m used in the declaration matches the type T declared for f in one of the following ways:

- 1 • m is a `Boolean_constant` and T is `BOOLEAN`.
- 2 • m is a `Character_constant` and T is one of the sized variants of `CHARACTER` for which m is a valid value.
- 3 • m is an `Integer_constant` and T is one of the sized variants of `INTEGER` for which m is a valid value.
- 4 • m is a `Real_constant` and T is one of the sized variants of `REAL` for which m is a valid value.
- 5 • m is a `Manifest_string` and T is one of the sized variants of `STRING` for which m is a valid value.
- 6 • m is a `Manifest_type`, of the form $\{Y\}$ for some type Y , and T is `TYPE [X]` for some stand-alone type X to which Y conforms.

The “valid values” are determined by each basic type’s semantics; for example 1000 is a valid value for *INTEGER_16* but not for *INTEGER_8*.

In case 6, we require the type listed in a *Manifest_type {Y}* to be *constant*, meaning that it does not involve any formal generic parameter or anchored type, as these may represent different types in different generic derivations or different descendants of the original class. This would not be suitable for a constant attribute, which must have a single, well-defined value.

Objects, values and entities

19.1 OVERVIEW

The execution of an Eiffel system consists of creating, accessing and modifying **objects**.

The following presentation discusses the structure of objects and how they relate to the syntactical constructs that denote objects in software texts: **expressions**. At run time, an expression may take on various *values*; every value is either an object or a reference to an object.

Among expressions, **entities** play a particular role. An entity is an identifier (name in the software text), meant at execution time to denote possible values. Some entities are **read-only**: the execution can't change their initial value. Others, called **variables**, can take on successive values during execution as a result of such operations as creation and assignment.

The description of objects and their properties introduces the *dynamic model* of Eiffel software execution: the run-time structures of the data manipulated by an Eiffel system.

This chapter and the following one will illustrate the dynamic model through figures representing values and objects. These figures and the conventions only serve explanatory purposes. In particular:

- Although they may suggest the actual implementation techniques used to represent values and objects at run time, they should not be construed as *prescribing* any specific implementation.
- Do not confuse these conventions for representing *dynamic* (that is to say, run-time) properties of systems with the graphical conventions for representing classes, features, the client relation, inheritance, and other *static* properties of software texts.

We saw that it is often convenient, in these representations of the static model, to picture *attribute* features in a form that resembles the representation of objects in the dynamic model. But this should cause no ambiguity since one convention applies to classes and the other to run-time objects.

← “[GRAPHICAL REPRESENTATION](#)”, 18.2, page 499.



19.2 OBJECTS AND THEIR TYPES

During its execution, an Eiffel system will create one or more objects.

There will always be at least one: the root object created on execution start.

← “*System execution*”,
page 114

A clear correspondence exists between objects, the dynamic (run-time) notion, and on the other side types and classes, the static (programming-time) notions. Every object proceeds from a type, itself based on a class. The following definitions capture this correspondence:



Type, generating type of an object; generator

Every run-time object is a direct instance of exactly one stand-alone type of the system, called the **generating type** of the object, or just “the type of the object” if there is no ambiguity.

The base class of the generating type is called the object’s **generating class**, or **generator** for short.

← “*Direct instance*”
was defined on page 329.

An object may be an *instance* of many types: if it is an instance of *TC*, it is also an instance of any type *TB* to which *TC* conforms. But it is a *direct instance* of only one type, and so has just one generating type.

← “*Instance of a type*”,
page 330.

To obtain the generating type of the object attached to *x*, you may use:

```
x.type
```

whose value is an object denoting a type. The query *type*, which comes from the universal class *ANY*, returns an object denoting a type, with the associated feature; with *x* declared of type *TX*, the type of *x.type* itself is

→ “*OBJECT PROPERTIES*”, 35.4, page 929

```
TYPE [TX]
```

based on the library class *TYPE*. More precisely, *TYPE [TX]* covers all objects representing types that *conform* to *TX*, including *TX* itself.

19.3 VALUES AND INSTANCES

We saw in the discussion of types that any possible value for an entity is either an object or a *reference*. The notion of reference has a precise definition:

← “*Direct instances and values of a type*”,
page 329.



Reference, void, attached, attached to

A **reference** is a value that is either:

- **Void**, in which case it provides no more information.
- **Attached**, in which case it gives access to an object. The reference is said to be **attached to** that object, and the object attached to the reference.



A non-void reference is “attached to” exactly one object, but an object may be attached to several references.

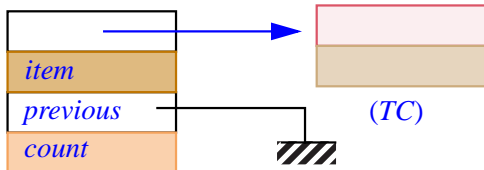
The reserved word *Void* denotes a void reference. To find out if the value of *e* is void, use the boolean expression

```
e = Void
```

See chapter 35 about the features of class ANY. Void may be implemented as an attribute or a once function.

Values of an *expanded* type can never be void.

The following figure shows conventions for representing a reference: by an arrow — more precisely, a blue arrow in this book — if attached to an object, by a special “grounding” symbol if void. Below an object you may write its generating type, here *TC*.



Picturing references, attached and void



The four values on the figure are the fields of the object on the left. The first and the third value from the top, labeled *next* and *previous*, are references; *next* is attached to the object on the right, and *previous* is void. The figure gives no information about the values in the expanded fields *item* and *count*, or about the fields of the object on the right.

The following property is essential to the consistency of the Eiffel type system and the dynamic model:

← Following directly from the “*Instance principle*”, page 331.

Object principle

Every non-void value is either an object or a reference attached to an object.

In particular, simple values such as integers, booleans and reals are objects.

19.4 BASIC TYPES

A number of object types come from classes of the Kernel Library: *BOOLEAN*; *CHARACTER* (64-bit) and *CHARACTER_8*; *INTEGER* and its sized variants *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *NATURAL*, *NATURAL_8*, *NATURAL_16*, *NATURAL_32*, *NATURAL_64*; *REAL* and its sized variants *REAL_32* and *REAL_64*; and *POINTER*.

The specification of their direct instances — boolean values, characters, integers, floating-point numbers, and addresses for passing to external software — appears in the [chapter on basic types](#).

→ [Chapter 30](#).

The specifications of direct instances appearing in the rest of the present chapter exclude the case of basic types.

19.5 REFERENCE AND COPY SEMANTICS

Object semantics

Every run-time object has either **copy semantics** or **reference semantics**.

An object has copy semantics if and only if its generating type is an expanded type.

This property determines the role of the object when used as source of an assignment: with copy semantics, it will be copied onto the target; with reference semantics, a reference will be reattached to it.

19.6 COMPOSITE OBJECTS AND THEIR FIELDS

We will use specific terminology for non-basic types:

Non-basic class, non-basic type, field

Any class other than the basic types is said to be a **non-basic class**. Any type whose base class is non-basic is a **non-basic type**, and its instances are **non-basic objects**.

A direct instance of a non-basic type is a sequence of zero or more values, called **fields**. There is one field for every attribute of the type's base class.

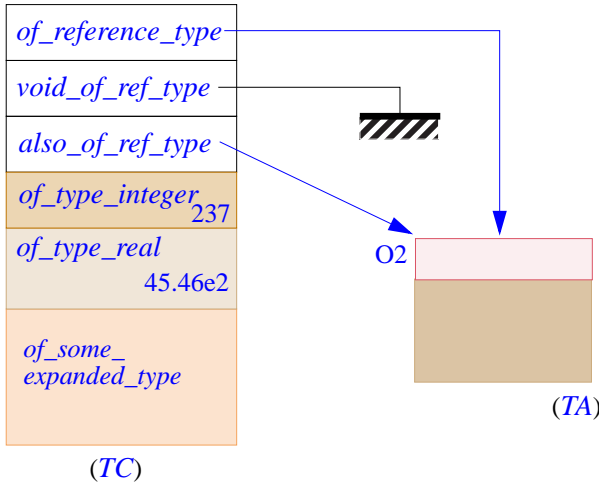
This definition makes no difference between variable and constant attributes. See the end of this section.

Consider a class type *TC*, of base class *C*, and an attribute *a* of class *C*; let *TA* be the type of *a*. The possible values for the field corresponding to attribute *a* in a direct instance of *TC* depend on the nature of *TA*. There are three possible cases for *TA*:

- 1 • Reference type.
- 2 • Expanded type.
- 3 • Formal generic parameter of class *C*.



In case 1, the field corresponding to attribute *a* is a reference. That reference may be void, or it may be attached to an instance of *TA*'s base type — not necessarily a direct instance. In the figure on the following page, the first and third fields from the top are attached to the same object, called *O2*.



An object and its fields

This represents a partial snapshot taken during the execution of a possible system, illustrating some of the various kinds of field.

In case 2, the field corresponding to attribute *a* is an instance of the expanded type *TA*. That field, then, is itself an object, called a **subobject** of the enclosing object. There are two cases:

- *TA* may be a basic type; then the subobject is a basic object of that type; the figure shows fields of type *INTEGER* and *REAL*.
- If *TA* is a non-basic expanded type, the subobject is itself a non-basic object. This applies to the last field of the left object on the figure. In this case the enclosing object is a **composite** object.



Subobject, composite object

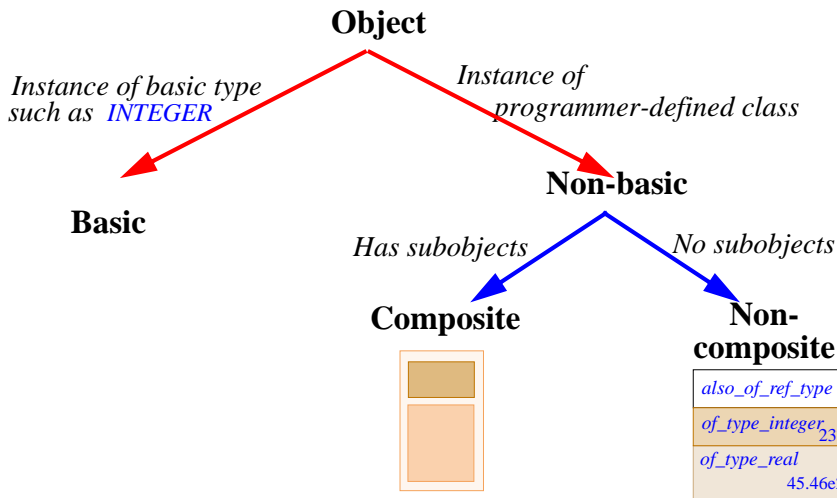
Any expanded field of an object is a **subobject** of that object.
 An object that has a non-basic subobject is said to be **composite**.

Finally, in case 3, *TA* is a formal generic parameter of class *C*, the base class of *TC*. Depending on whether the actual generic parameter is a reference type or an expanded type, this will in fact yield either case 1 or case 2.



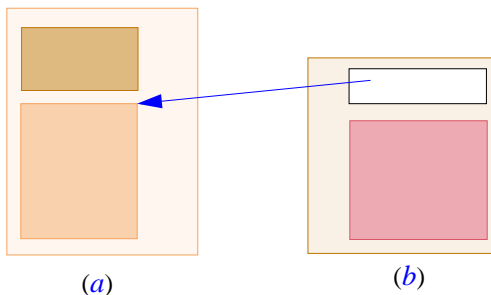
The above definition of fields makes no difference between constant and variable attributes: an attribute of either kind yields a field in every instance. In a reasonable implementation, fields for constant attributes, being the same value for every instance of a class, will not occupy any run-time space. This indicates again that figures representing objects (such as the ones in this chapter) do not necessarily show actual object implementations. This book uses "field" in the precise sense defined above, which does not always imply an actual memory area in an object's representation.

Here is a summary of the classification of objects:



19.7 REFERENCE ATOMICITY

The dynamic model as illustrated above has both composite objects, containing subobjects, and references to objects. How do these notions combine? In particular, can a system produce the run-time situation shown on the following figure, where a reference is attached to a subobject of another object?



Reference to subobject

WARNING: This illustrates an impossible situation.

The answer is no. The semantics of reattachment operations (**Assignment**, formal-actual argument association) will guarantee that a reference can only become attached to a full object. Although objects themselves are not “atomic”, since clients can modify individual fields by calling the appropriate routines, the level of atomicity for attaching *references* is an entire object.

→ See case [2] of attachment semantics table on page 598, and the discussion that follows.



It is possible to conceive of a model that supports references to subobjects, as was in fact the case in ISE Eiffel 2. But this significantly complicates the dynamic model and the implementation, garbage collection in particular, without bringing a clearly useful improvement in expressive power.

19.8 EXPRESSIONS AND ENTITIES

The discussion so far has defined the object structures that can be created during system execution. To denote the objects and their fields in software texts, you may use expressions — specimens of the construct **Expression**.

There are several forms of expression, which subsequent chapters cover in detail. One form, the simplest, is of immediate interest: entities, which consist of a single name.



Entity, variable, read-only

An **entity** is an **Identifier**, or one of two reserved words (**Current** and **Result**), used in one of the following roles:

- 1 • Final name of an attribute of a class.
- 2 • Local variable of a routine or **Inline_agent**, including **Result** for a query.
- 3 • Formal argument of a routine or inline agent.
- 4 • Object Test local.
- 5 • **Current**, the predefined entity used to represent a reference to the current object (the target of the latest not yet completed routine call).

Names of non-constant attributes and local variables are **variable** entities, also called just **variables**. Constant attributes, formal arguments, Object Test locals and **Current** are **read-only** entities.

→ See [8.6, page 225](#), about local variables and **Result**. Inline agents are as described in chapter 27 and Object Test locals in [24.3, page 658](#).

Two kinds of operation, creation and reattachment, may modify the value of a variable (a non-constant attribute, part of category **1**, or local variable, category **2**). In the other four cases — constant attributes, formal arguments (**3**), Object Test locals (**4**) and **Current** (**5**) — you may not directly modify the entities, hence the name *read-only* entity.

→ Creation: chapter [20](#); reattachment: chapter [22](#).

The term “*constant* entity” wouldn’t do, not so much because you can modify the corresponding objects but because read-only entities (other than constant attributes) do change at run time: a qualified call reattaches **Current**, and any routine call reattaches the formal arguments.

→ “*Current object, current routine*”, page [649](#)

Result appearing in the **Postcondition** of a constant attribute cannot be changed at execution time, but for simplicity is considered part of local variables in all cases anyway.

Here is the corresponding syntax specification:



Entities and variables

Entity \triangleq Variable | Read_only

Variable \triangleq Variable_attribute | Local

Variable_attribute \triangleq Feature_name

Local \triangleq Identifier Result
Read_only \triangleq Formal Constant_attribute Current
Formal \triangleq Identifier
Constant_attribute \triangleq Feature_name

The constraint on entities indicates that an entity must be of one of the five forms listed above. In addition, local variables, formal arguments and Object Test locals are only permitted in certain contexts:



Entity rule	VEEN
<p>An occurrence of an <u>entity</u> e in the text of a class C (other than as the feature of a qualified call) is valid if and only if it satisfies one of the following conditions:</p>	
<p>1 • e is Current.</p>	
<p>2 • e is the <u>final name</u> of an attribute of C.</p>	
<p>3 • e is the local variable Result, and the occurrence is in a <u>Feature_body</u>, <u>Postcondition</u> or <u>Rescue</u> part of an <u>Attribute_or_routine</u> text for a <u>query</u> or an <u>Inline_agent</u> whose <u>signature</u> includes a result type.</p>	
<p>4 • e is Result appearing in the <u>Postcondition</u> of a <u>constant attribute</u>'s declaration.</p>	
<p>5 • e is listed in the <u>Identifier_list</u> of an <u>Entity_declaration_group</u> in a <u>Local_declarations</u> part of a feature or <u>Inline_agent</u> fa, and the occurrence is in a <u>Local_declarations</u>, <u>Feature_body</u> or <u>Rescue</u> part for fa.</p>	
<p>6 • e is listed in the <u>Identifier_list</u> of an <u>Entity_declaration_group</u> in a <u>Formal_arguments</u> part for a routine r, and the occurrence is in a <u>declaration for</u> r.</p>	
<p>7 • e is listed in the <u>Identifier_list</u> of an <u>Entity_declaration_group</u> in the <u>Agent_arguments</u> part of an <u>Agent</u> a, and the occurrence is in the <u>Agent_body</u> of a.</p>	
<p>8 • e is the <u>Object-Test Local</u> of an <u>Object_test</u>, and the occurrence is in its <u>scope</u>.</p>	

→ “*Scope of an Object-Test Local*”, page 661.



“Other than as feature of a qualified call” excludes from the rule any attribute, possibly of another class, used as feature of a qualified call: in $a.b$ the rule applies to a but not to b . The constraint on b is the General Call rule, requiring b to be the name of a feature in D 's base class.

→ *The General Call rule is on page 681.*

A related rule defines what it means for an entity to be a **Variable**:



Variable rule

VEVA

A **Variable** entity v is valid in a class C if and only if it satisfies one of the following conditions:

- 1 • v is the final name of a variable attribute of C .
- 2 • v is the final name of a local variable of the immediately enclosing routine or agent.

This will determine whether you may use e as the target of an **Assignment**. → *Assignment is discussed in chapter 22.*
 Note that v in clause 2 has to be a local variable (including, as usual **Result**) of the *immediately* enclosing routine or agents. Routines may not be nested, but an agent appears in a routine (and possibly in another agent); only the local variables of the immediately enclosing scope are assignable.

19.9 SEMANTICS: EVALUATING AND INITIALIZING ENTITIES



The semantic purpose of an entity is to be ready at execution time to deliver an associated value whenever queried, or **evaluated**. The validity and semantic rules of the language must ensure that whenever this happens the entity denotes *exactly one* value, and to define what that value will be.

For read-only entities this is achieved through simple properties, whose details appear in other chapters:

- A constant attribute has the value specified in its declaration. → *29.10, page 813.*
- **Current** gets attached to the root object on system start, and at the start of a qualified call $x.f(\dots)$ denotes the value of the target x . → *"Current Semantics", page 651.*
- On entry to a routine, a formal argument gets attached to the value of the corresponding actual. → *"PRECISE CALL SEMANTICS", 23.17, page 652.*

For a variable, the picture is a bit more subtle. The result of the evaluation is a consequence of the operations that may have affected the variable:

- **Initialization**, as it occurs on object creation (for an attribute) or a routine call (for a local variable).
- Any **assignment** using the variable as its target.

Assignment has a well-defined semantics, discussed in detail in the corresponding chapter. But the execution might evaluate the variable before it has been the target of any explicit assignment; it is the task of initialization rules to ensure that even in such a case every variable has one well-defined value. → *Chapter 22.*



This is not the case in all programming languages; many leave it to the programmer to ensure that every variable is assigned before use. In Eiffel, it is a language design principle that the rules must be sufficient to deduce, for any evaluation of any variable, a well-defined result.

The value of a variable that hasn't yet been the target of an assignment will be determined by the **initialization rules** that we will now study. These rules determine *which value* a variable will hold prior to assignment, and *when* exactly that value will be set.

---- TO BE REDONE ---There are two possibilities, depending on the type of the variable:

- The most common case covers variables of *basic types* as well as non-attached ones of *reference types*. An attribute of such a type denotes a field in the corresponding objects, and will accordingly be initialized as part of object creation. A local variable (including *Result* for a function) is initialized anew for each call of its routine. In both cases the initial values are language-specified: zero for numbers, false for booleans, null character for characters, and for references — covering all other possibilities — a void reference.
- *Expanded types* raise a special issue because their semantics require variables, when evaluated, always to be attached to an object of the corresponding type. Such an object cannot just follow from the declaration of the variable (like the value 0, in the previous case, follows from the declaration of an *INTEGER* variable); it has to come out of a creation instruction. The rule then is to create an object on *first evaluation* of the variable — meaning for an attribute the first evaluation for any given object, and for a local variable the first evaluation in any given call. The evaluation will cause creation of an object of the appropriate type, using the procedure *default_create*, which must be one of the creation procedures of the type.

This is the gist of the rules. Let now see their precise form. First we name ---- REWRITE our two type categories:

Self-initializing type

A type is **self-initializing** if it is one of:

- 1 • A detachable type.
- 2 • A self-initializing formal parameter.
- 3 • An attached type (including expanded types and, as a special case of these, basic types) whose creation procedures include a version of *default_create* from *ANY* available for creation to *C*.

A self-initializing type enables us to define a default initialization value:

- Use *Void* for a detachable type (case 1, the easiest but also the least interesting)
- Execute a creation instruction with the applicable version of *default_create* for the most interesting case: 3, attached types, including expanded types. This case also covers basic types, which all have a default value given by the following rule.

A “self-initializing formal parameter” (case 2) is a generic parameter, so we don’t exactly know which one of these three semantics will apply; but we do require, through the Generic Derivation rule, that any attached type used as actual generic parameter be self-initializing, meaning in this case that it will provide *default_create*.

In the definition, the “creation procedures” of a *type* are the creation procedures of its base *class* or, for a formal generic parameter, its “constraining creators”, the features listed as available for creation in its constraining type.

The more directly useful notion is that of a self-initializing *variable*, appearing below.

The term “self-initializing” is justified by the following semantic rule, specifying the actual initialization values for every self-initializing type.

:



Default Initialization rule

Every self-initializing type *T* has a **default initialization value** as follows:

- 1 • For a detachable type: a void reference.
- 2 • For a self-initializing attached type: an object obtained by creating an instance of *T* through *default_create*.
- 3 • For a self-initializing formal parameter: for every generic derivation, (recursively) the default initialization value of the corresponding actual generic parameter.
- 4 • For *BOOLEAN*: the boolean value false.
- 5 • For a sized variant of *CHARACTER*: null character.
- 6 • For a sized variant of *INTEGER*: integer zero.
- 7 • For a sized variant of *REAL*: floating-point zero.
- 8 • For *POINTER*: a null pointer.
- 9 • For *TYPED_POINTER*: an object representing a null pointer.

This rule is the reason why everyone loves self-initializing types: whenever execution catches an entity that hasn't been explicitly set, it can (and, thanks to the Entity Semantics rule, will) set it to a well-defined default value. This idea gains extra flexibility, in the next definition, through the notion of attributes with an explicit initialization.

The notion generalizes ---- COMPLETE

Self-initializing variable

A variable is **self-initializing** if one of the following holds:

- 1 • Its type is a self-initializing type.
- 2 • It is an attribute declared with an **Attribute** part such that the entity **Result** is properly set at the end of its **Compound**.

If a variable is self-initializing, we don't need to worry about finding it with an undefined value at execution time: if it has not yet been the target of an attachment operation, automatic initialization can take over and set it to a well-defined default value. That value is, in case 1, the default value for its type, and in case 2 the result of the attribute's own initialization. That initialization must ensure that **Result** is "properly set" as defined next (partly recursively from the above definition) .

T

---- EXPLAIN

Evaluation position, precedes

An **evaluation position** is one of:

- In a **Compound**, one of its **Instruction** components.
- In an **Assertion**, one of its **Assertion_clause** components.
- In either case, a special **end position**.

A position **p** **precedes** a position **q** if they are both in the same **Compound** or **Assertion**, and either:

- **p** and **q** are both **Instruction** or **Assertion_clause** components, and **p** appears before **q** in the corresponding list.
- **q** is the end position and **p** is not.

This notion is needed to ensure that entities are properly set before use.

In a compound *i1; i2; i3* we have four positions; *i1* precedes *i2*, *i3* and the end position, and so on.

The relation as defined only applies to **first-level** components of the compound: if *i2* itself contains a compound, for example if it is of the form **if *c* then *i4*; *i5* end**, then *i4* is not an evaluation position of the outermost compound, and so has no “precedes” relation with any of *i1*, *i2* and *i3*.

Setter instruction

A **setter instruction** is an assignment or creation instruction.

If *x* is a variable, a setter instruction is a **setter for *x*** if its assignment target or creation target is *x*.

Properly set variable

At an evaluation position *ep* in a class *C*, a variable *x* is **properly set** if one of the following conditions holds:

- 1 • *x* is self-initializing.
- 2 • *ep* is an evaluation position of the **Compound** of a feature or **Inline_agent** of the **Internal** form, one of whose instructions precedes *ep* and is a setter for *x*.
- 3 • *x* is a variable attribute, and is (recursively) properly set at the end position of every creation procedure of *C*.
- 4 • *ep* is an evaluation position in a **Compound** that is part of an instruction *ep'*, itself belonging to a **Compound**, and *x* is (recursively) properly set at position *ep'*.
- 5 • *ep* is in a **Postcondition** of a routine or **Inline_agent** of the **Internal** form, and *x* is (recursively) properly set at the end position of its **Compound**.
- 6 • *ep* is an **Assertion_clause** containing **Result** in the **Postcondition** of a constant attribute

The key cases are [2](#), particularly useful for local variables but also applicable to attributes, and [3](#), applicable to attributes when we cannot deduce proper initialization from the enclosing routine but find that every creation procedure will take care of it. Case [4](#) accounts for nested compounds. For assertions other than postconditions, which cannot use variables other than attributes, [3](#) is the only applicable condition. The somewhat special case [6](#) is a consequence of our classification of **Result** among local variables even in the **Postcondition** of a constant attribute.

As an artefact of the definition's phrasing, every variable attribute is "properly set" in any effective routine of a deferred class, since such a class has no creation procedures. This causes no problem since a failure to set the attribute properly will be caught, in the validity rule below, for versions of the routine in effective descendants.



Variable Initialization rule *VEVI*

It is valid for an **Expression**, other than the target of an **Assigner_call**, to be also a **Variable** if it is properly set at the evaluation position defined by the closest enclosing **Instruction** or **Assertion_clause**.

This is the fundamental requirement guaranteeing that the value will be defined if needed.

Because of the definition of “properly set”, this requirement is pessimistic: some examples might be rejected even though a “smart” compiler might be able to prove, by more advanced control and data flow analysis, that the value will always be defined. But then the same software might be rejected by another compiler, less “smart” or simply using different criteria. On purpose, the definition limits itself to basic schemes that all compilers can implement.

If one of your software elements is rejected because of this rule, it’s a sign that your algorithms fail to initialize a certain variable before use, or at least that the proper initialization is not clear enough. To correct the problem, you may:

- Add a version of *default_create* to the class, as creation procedure.
- Give the attribute a specific initialization through an explicit **Attribute** part that sets **Result** to the appropriate value.

Variable setting and its value

A **setting** for a variable x is any one of the following run-time events, defining in each case the **value** of the setting:

- 1 • Execution of a setter for x . (*Value*: the object attached to x by the setter, or a void reference if none.)
- 2 • If x is a variable attribute with an **Attribute** part: evaluation of that part, implying execution of its **Compound**. (*Value*: the object attached to **Result** at the end position of that **Compound**, or a void reference if none.)
- 3 • If the type T of x is self-initializing: assignment to x of T ’s default initialization value. (*Value*: that initialization value.)

As a consequence of case 2, an attribute a that is self-initializing through an **Attribute** part ap is *not* set until execution of ap has reached its end position. In particular, it is not invalid (although definitely unusual and perhaps strange) for the instructions ap to use the value a : as with a recursive call in a routine, this will start the computation again at the beginning of ap . For attributes as for routines, this raises the risk of infinite recursion (perhaps higher for attributes since they have no arguments) and it is the programmer's responsibility to avoid this by ensuring that before a recursive call the context will have sufficiently changed to ensure eventual termination. No language rule can ensure this (in either the routine or attribute cases) since this would amount to solving the "halting problem", a provably impossible task.

Another consequence of the same observation is that if the execution of ap triggers an exception, and hence does not reach its end position, any later attempt to access a will also restart the execution of ap from the beginning. This might trigger the same exception, or succeed if the conditions of the execution have changed.

Execution context

At any time during execution, the current **execution context** for a variable is the period elapsed since:

- 1 • For an attribute: the creation of the current object.
- 2 • For a local variable: the start of execution of the current routine.

Variable Semantics

The value produced by the run-time evaluation of a variable x is:

- 1 • If the execution context has previously executed at least one setting for x : the value of the latest such setting.
- 2 • Otherwise, if the type T of x is self-initializing: assignment to x of T 's default initialization value, causing a setting of x .
- 3 • Otherwise, if x is a variable attribute with an **Attribute** part: evaluation of that part, implying execution of its **Compound** and hence a setting for x .
- 4 • Otherwise, if x is **Result** in the **Postcondition** of a constant attribute: the value of the attribute.

This rule is phrased so that the order of the first three cases is significant: if there's already been an assignment, no self-initialization is possible; and if T has a default value, the **Attribute** part won't be used.

The Variable Initialization rule ensures that one of these cases will apply, so that x will always have a well-defined result for evaluation. This property was our main goal, and its achievement concludes the discussion of variable semantics.

The previous rule applies only to variables. We now generalize it to a general rule governing all entities:



Entity Semantics rule

Evaluating an entity yields a **value** as follows:

- 1 • For **Current**: a value attached to the current object.
- 2 • For a formal argument of a routine or **Inline_agent**: the value of the corresponding actual at the time of the current call.
- 3 • For a constant attribute: the value of the associated **Manifest_constant** as determined by the Manifest Constant Semantics rule.
- 4 • For an Object-Test Local: as determined by the Object-Test Local Semantics rule.
- 5 • For a variable: as determined by the Variable Semantics rule.

← "*Current object, current routine*", page 649.

This rule concludes the semantics of entities by gathering all cases. It serves as one of the cases of the semantics of expressions, since an entity can be used as one of the forms of **Expression**.

The Object-Test Local Semantics rule appears in the discussion of the **Object_test** construct.

Creating objects

20.1 OVERVIEW

The dynamic model, whose major properties were reviewed in the preceding presentations, is highly flexible; your systems may create objects and attach them to entities at will, according to the demands of their execution. The following discussion explores the two principal mechanisms for producing new objects: the **Creation_instruction** and its less frequently encountered sister, the **Creation_expression**.

A closely related mechanism — **cloning** — exists for duplicating objects. This will be studied separately, with the mechanism for copying the contents of an object onto another.

The creation constructs offer considerable flexibility, allowing you to rely on language-defined initialization mechanisms for all the instances of a class, but also to override these defaults with your own conventions, to define any number of alternative initialization procedures, and to let each creation instruction provide specific values for the initialization. You can even instantiate an entity declared of a generic type — a non-trivial problem since, for x declared of type G in a class $C [G]$, we don't know what actual type G denotes in any particular case, and how one creates and initializes instances of that type.

In using all these facilities, you should never forget the methodological rule governing creation, as expressed by the following principle.



Creation principle

Any execution of a creation operation must produce an object that satisfies the invariant of its generating class.

Such is the theoretical role of creation: to make sure that any object we create starts its life in a state satisfying the corresponding invariant. The various properties of creation, reviewed next, are designed to ensure this principle.

20.2 FORMS OF CREATION: AN OVERVIEW



You may use a **Creation_instruction** to produce a totally new object, initialize its variable fields to preset values, and attach it to a **Variable** entity called the **target** of the creation and named in the instruction.

The examples which follow assume that the target is of a reference (non-expanded) type. As will be seen below, the **Creation_instruction** is also applicable to expanded types, although with a less interesting effect.

See [20.8, page 542](#) below, about *Creation instructions applied to expanded types*.

Syntactically, a **Creation_instruction** always begins with the keyword **create**, followed by the target. Here are some examples:



create <i>account1</i>	[1]
create <i>point1</i> . <i>make_polar</i> (1, Pi / 4)	[14]
create { <i>SAVINGS_ACCOUNT</i> } <i>account1</i>	[15]
create { <i>SEGMENT</i> } <i>figure1</i> . <i>make</i> (<i>point1</i> , <i>point2</i>)	[16]

The respective targets are *account1*, *point1*, *account1*, *figure1*.

With form [1] you create an object of the type declared for *account1*, initialize it to default values, and attach it to *account1*. The default initialization is language-defined, although you can override it for any class.

With form [14] you create an object of the type declared for *point1*, apply the standard default initialization, complement the initialization by calling *make_polar* (a procedure of the class, designated as one of its “creation procedures”) with the given arguments, and attach the object to *point1*.

Cases [15] and [16] are respectively similar to the first two, but specify an explicit type, in braces, for the new object. So if *account1* is of type *ACCOUNT*, form [1] creates an instance of that class, but form [15] creates an instance of *SAVINGS_ACCOUNT*. This requires *SAVINGS_ACCOUNT* to be a descendant of *ACCOUNT*. Similarly, in form [16], *SEGMENT* must be a descendant of the type, say *FIGURE*, declared for *figure1*.

--- ADD INTRO TO CREATION EXPRESSIONS ---

Since the run-time effect of a creation instruction or expression is essentially the same, it is convenient to have a name covering both:

Creation operation

A **creation operation** is a creation instruction or expression.

20.3 BASIC FORM OF CREATION INSTRUCTIONS

Even though example [1] shows the most concise variant, a better place to start studying the `Creation_instruction` is the more general variant illustrated by [14]: `create x.creation_procedure (...)`. Its effect is, in order, to:

- 1 • Create a new object — a direct instance of the type *T* of *x*.
- 2 • Initialize all the variable fields of that object to default values.
- 3 • Call `creation_procedure` on the object, with the arguments given, to complete its initialization.
- 4 • Attach *x* to the object.

The default initialization values used in step 2 are adapted to the type of each field corresponding to a variable attribute: zero for numbers, false for booleans, void for references and so on. The full rule will appear later.

→ “[Default Initialization rule](#)”, page 516.

This form of the instruction is only valid if the base class *C* of *x*'s type *T* lists `creation_procedure` in its `Creators` part.



Such a `Creators` part is permitted only in an effective class (since it makes no sense to create direct instances of a deferred class). We have seen that it comes towards the beginning of a class text — just before `Features` but after `Inheritance` — and consists of at least one `Creation_clause`, each beginning with the keyword `create` followed by a list of zero or more procedures of the class, as in

← “[PARTS OF A CLASS TEXT](#)”, 4.7, page 119.



```
class C ... inherit
    ...
create
    make, execute, ...
feature
    ...
end
```

where `make, execute ...` are procedures of *C*. For the moment we are restricting ourselves to just one `Creation_clause` (the vast majority of cases). By including such a clause, the author of *C* specifies that any `Creation_instruction` producing direct instances of the class must be of one of the two forms

→ You can use more than one `Creation_clause`; also, each one may restrict clients' creation privileges. See below “[RESTRICTING CREATION AVAILABILITY](#)”, 20.7, page 539 for full details.



```
create x.make (...)
create x.execute (...)
```

which will initialize the new object by calling the specified creation procedure — with actual arguments whose types and number match those of the formal arguments declared for the procedure.



The two creation-related constructs, `Creators` and `Creation_instruction`, both use the same keyword `create`. This makes things easier to remember than if you had to learn two keywords. No confusion can result since the constructs appear in completely different syntactic contexts.

Creation procedures (also known as “*constructors*” from C++ terminology) serve to apply initializations beyond the default ones if these do not suffice. For example, the author of a class `POINT` in a graphics system may wish to offer a creation mechanism that not only allocates a new object but also initializes its fields according to coordinates provided by the client. Here is an outline of such a class:



```

class POINT inherit
  TRIGONOMETRY
create
  make_polar, make_cartesian
feature -- Access
  ro, theta: REAL
  x, y: REAL
feature -- Element change
  make_polar (r, t: REAL)
    -- Set to polar coordinates r, t.
    do
      ro := r; theta := t
      reset_from_polar
    end
  make_cartesian (a, b: REAL)
    -- Set to cartesian coordinates a, b.
    do
      x := a; y := b
      reset_from_cartesian
    end
  ... Other exported features ...
feature {NONE} -- Implementation
  consistent_attributes: BOOLEAN
    -- Do polar and cartesian attributes
    -- represent same point?
    do
      Result := (x = ro * cos (theta)) and
        (y = ro * sin (theta))
    end
end

```

This example assumes a library class `TRIGONOMETRY` offering functions such as `cos` and `sin`. The equality in `consistent_attributes` should be changed to an approximate equality to account for numerical precision issues.

```

reset_from_polar
    -- Update cartesian coordinates from polar ones.
do
    x := ro * cos (theta); y := ro * sin (theta)
ensure
    consistent_attributes
end

reset_from_cartesian
    -- Update polar coordinates from cartesian ones.
do
    ...
ensure
    consistent_attributes
end

invariant
    consistent: consistent_attributes
end

```

With this design, the author of class *POINT* provides clients with two creation mechanisms: one initializes a point by its polar coordinates, the other by its cartesian coordinates. Examples of *Creation_instruction*, assuming that *point1* is a *Variable* entity of type *POINT*, are

```

create point1.make_polar (2, Pi / 4)
create point1.make_cartesian (Sqrt2, Sqrt2)

```

If Pi and Sqrt2 are real constants with the values suggested by their names, these instructions will have the same effect.



Names of the form *make_something* are common practice for creation procedures, although by no means required. When a class has just one creation procedure, or one more fundamental than the others, the convention is to call it just *make* — although if the procedure has no arguments your clients can ignore it altogether, if you use *default_create* as will now be seen.

20.4 OMITTING THE CREATION PROCEDURE

In some common cases you can avoid specifying a creation procedure. This gives the simplest possible form of *Creation_instruction*, illustrated by the first of our initial examples:

```

create x

```

This form is applicable when the base class *C* of *x*'s type does *not* have a **Creators** part. This is particularly useful for simple classes which do not need particularly flexible creation mechanisms, but just provide clients with a standard way to create instances without providing any specific information. These instances will all be initialized in the same way. A simple example is



```
note
  description: "%[Binary trees with nodes containing
               information of type G%]"
class BINARY_TREE [G]... feature -- Access
  item: G
    -- Node information
  left, right: BINARY_TREE [G]
    -- Left and right children
feature -- Element change
  ... Features to set node information and attach children...
end
```

Here a creation instruction, for *bt* of type *BINARY_TREE [SOME_TYPE]*, will simply be



```
create bt
```

and will set all the fields of the resulting object to their default values: void references for *left* and *right*, the default value of the actual generic parameter (whatever it may be) for *item*.

This simple form of the **Creation_instruction** is appropriate when the object-creating client is happy to rely on a standard initialization. But even in this case you may need more fine-tuning, because the language-defined default initializations might not suit all classes. Consider



```

class EMPLOYEE inherit
  PERSON
  feature -- Access
    Unknown_marital_status, Single, Widowed, Divorced:
      INTEGER -- !!!!! REDO EXAMPLE !!!!!
    marital_status: INTEGER
  feature
    ... Other features ...
  invariant
    meaningful_marital_status:
      marital_status >= Unknown_marital_status and
      marital_status <= Divorced
  end

```

We require, as expressed by the invariant, that *marital_status* have one of the values listed. Because this attribute is of type *INTEGER*, the universal default initializations would set it to zero — not compatible with the invariant! Remember the **Creation principle**: it is creation’s responsibility to ensure that every new object satisfies the invariant.

Creation principle:
page 523.

One solution is to use a creation procedure:



```

class EMPLOYEE inherit
  PERSON
  create
    make
  feature -- Initialization
    make
      -- Initialize by setting marital status to “Unknown”.
    do
      marital_status := Unknown_marital_status
    end
  feature -- Access
    ... Other features and invariant as before ...
  end

```

Since the class now has a **Creators** part, the abbreviated form **create emp** (for *emp* of type *EMPLOYEE*) is no longer valid: we are back to the previous technique and must write



```
create emp.make
```

This approach works but is a bit tedious for the clients since they must specify a creation procedure for no clear benefit: only one such procedure is available, *make*, and it takes no argument.

In such a case — providing a standard initialization, but not necessarily the universal language-defined one — you can still make the simple creation form **create** *x* valid for your clients. Do not include a **Creators** part; just redefine the procedure *default_create* which, coming from class *ANY*, is a feature of all classes. This redefinition will specify your desired initializations.

This technique relies on a simple convention: any class *C* without a **Creators** part is treated as if it had one of the form

```
create
  default_create
```

(If *default_create* has been renamed, this should use the new name instead.) In other words, a class which doesn't list any creation procedures is considered to have just one — its version of *default_create*.

Correspondingly, a **Creation_instruction** of the form **create** *x*, which doesn't specify a creation procedure, is treated as a shorthand for

```
create x.default_create
```

for *x* of a type based on *C* (again with the understanding that, if *default_create* has been renamed, this unfolded form uses the new name).

With this technique we can adapt class *EMPLOYEE* so that its clients can create instances by writing just



```
create emp
```

with no creation procedure. The new form of the class is almost the same as the last one seen, but instead of a specific creation procedure *make* we don't include any **Creators** part and just redefine *default_create*:



```

class EMPLOYEE inherit
  PERSON
  redefine default_create end

feature -- Initialization
  default_create
    -- Initialize by setting marital status to "Unknown".
    do
      marital_status := Unknown_marital_status
    end

feature -- Access
  ... Other features and invariant as before ...
end

```



Because such a class redeclares a feature *default_create* which it inherits in non-deferred form, it must state **redefine** *default_create* in some **Inheritance** part. Here *EMPLOYEE* inherits from *PERSON*, so we just stick this clause into the corresponding **Inheritance** part. If the class didn't have any **Inheritance** part — meaning that it only has an implicit parent, *ANY* — we would have to use the standard idiom enabling such a class to redefine a feature coming from *ANY*: include an **Inheritance** part making *ANY* an explicit rather than implicit parent. This would give:



```

class EMPLOYEE inherit
  -- Here we make ANY an explicit parent:
  ANY
  redefine default_create end

feature -- Initialization
  ... Feature clauses and invariant as before ...
end

```

Let's review the two schemes studied in the previous section and this one:

- 1 • To provide clients with specific creation procedures, which may take arguments, include at the beginning of the class a **Creators** part, of the form **create** *cp1*, *cp2*, ... , where the *cp_i* are procedures of the class. A **Creation_instruction** in this case must be of the form **create** *x.cp* (...) where *cp* is one of the specified *cp_i*.
- 2 • To make the simplified form **create** *x* valid, you do not need to include any **Creators** part: this form is equivalent to the previous case using for *cp* the procedure *default_create*; and an absent **Creators** part is equivalent to one that lists only that procedure.

At first these two cases may seem incompatible, but if you examine them more closely you will realize they are not. The rule is simply that the simplified form `create x` is valid if and only if `default_create`, in its local version, is one of the creation procedures of the class. You can achieve this property by not listing any creation procedures at all: this is equivalent to listing `default_create` only. But you can also have a **Creators** part, provided it lists `default_create`, possibly among other procedures. This observation yields a third case, combining the previous two:

- 3 • To make both forms of creation instruction valid — the form with an explicit procedure, `create x.cpi(...)` for some `cpi`, and the procedure-less form, `create x` — simply include a **Creators** part that lists both the desired `cpi` and the class's version of `default_create`.

Here is an example of this last scheme, a variation on an earlier class text:

← See the original version on page [526](#).



```
class POINT inherit
  TRIGONOMETRY
create
  make_polar, make_cartesian, default_create
feature
  ... Features as before ...
invariant
  consistent: consistent_attributes
end
```

Then all of the following four creation instructions are valid:

```
[1]
  create your_point.make_polar (2, Pi/4)
[1]
  create your_point.make_cartesian (Sqrt2, Sqrt2)
[2]
  create your_point.default_create
[3]
  create your_point
```

Forms [2](#) and [3](#) are exactly equivalent, so there is usually little reason to use [2](#) except if you insist on including the creation procedure for clarity.



Note that including `default_create` among the creation procedures, hence permitting [3](#), makes sense only because the default initializations ensure the invariant `consistent_attributes`, which states that cartesian and polar coordinates agree — true if they are all zero, the default. When thinking about creation, always keep in mind the [Creation principle](#).

← Creation principle: page [523](#).

As a variation on this example, assume that you write a class *C* that inherits from a parent *B* a procedure *set* without arguments, and want *C* to offer its clients the procedure-less form **create** *x* so that it will call *set* for initialization. A simple technique is:



```
class C inherit
  B
    rename
      default_create as discarded
    end
  ANY
    rename
      default_create as set
    undefine
      set
    select
      set
    end
feature
  ...
end
```

This uses a **join** to merge two inherited features, undefining *default_create* along one of the branches so that its joined feature *set* can override its previous implementation. Corresponding creation instructions may be written **create** *x*.

← See “[THE JOIN MECHANISM](#)”, 10.21, page 292.

We can now summarize the basic rule for validity of a creation instruction: the *instruction’s creation procedure* must be one of the *class’s creation procedures*, with the understanding that:

- 1 • Every creation instruction uses a creation procedure — either explicit, as in **create** *x.cp (...)*, or implicit, as in **create** *x*, where the instruction’s creation procedure is *default_create*.
- 2 • Every class lists a set of creation procedures — either explicit, if the class has a **Creators** part, or implicitly taken to be *default_create* in the absence of a **Creators** part.

This also suggests, as a special case, what you should do if for some reason you do **not** want clients of a class to create any direct instances of it. Simply include a **Creators** part, but make it empty:



```
class NOT_INSTANTIABLE create
  -- Nothing at all listed here!
feature
  ...
end
```

WARNING: not the recommended style; see next.

This falls under the “explicit” case of observation 1 above, so that under observation 2 a creation instruction could only be valid if it were of the form `create x.cp (...)` where `cp` is a creation procedure of the class; but there is no such `cp` since the **Creators** part, although present, is empty.



The style guideline in such a case is actually to write

```
class NOT_INSTANTIABLE create {NONE}

feature

end ...
```

which has exactly the same effect but emphasizes the creation ban by listing `NONE` as the single creation (rather, non-creation) client, based on conventions, seen below, for restricting creation availability. → [“RESTRICTING CREATION AVAILABILITY”, 20.7, page 539.](#)

Another way to make a class non-instantiable is to declare it as deferred. But you might want to prohibit instantiation of a class even if it is effective. Then you can use the technique just seen.

20.5 CREATORS AND INHERITANCE



(This section is a discussion of the *absence* of dependency between two language concepts, so it introduces no new mechanism; it is a “comment” and “methodology” section meant to dispel a possible confusion, which might in particular follow from experience with other languages.)

You may have been wondering what effect the inheritance structure has on the creation procedures of a class. The short answer is: *no effect*. Each class is free to choose the procedures it wants to offer to its clients for creation, regardless of its parents’ choices. The creation mechanism does of course take full advantage of inheritance: creation procedures may be obtained from parents and adapted through the usual inheritance mechanisms of redefinition, renaming, effecting and so on. And in some cases a class’s choice of creation procedures is directly connected to its parents’ choices:

- A class may list as creation procedures (in its **Creators** part) some or even all of a parent’s own creation procedures.
- A redefined creation procedure may need, as part of its execution, to call the parent’s version, usually through the **Precursor** mechanism.

But all this is optional, not required, and neither theoretical analysis nor analysis of practical examples suggests an obligatory connection. Counterexamples indeed abound. Just think of a class `POLYGON`, where a typical creation procedure will take a list of vertices; for its heir `RECTANGLE` this is most likely inappropriate, as we might use a center, an orientation and two side lengths; then for a grandchild `SQUARE` we will again need something different since we can dispense with one of these lengths.

So the set of creation procedures of a class is entirely determined by its **Creators** clause (or lack thereof, as we have seen), without interference from the parents' own clauses. This yields a simple semantics and avoids confusion. Based on the needs of each class, you decide what creation privileges you award to *your* clients; you may reuse the parents' creation procedures, unchanged or extended, but only if you find them useful for your own needs.

Eiffel's policy on relating *creation status* to inheritance is similar to its policy on relating *export status* to inheritance. There too every class is free to make its own decisions for inherited features, regardless of its parents' choices. The only difference is the default: inherited features retain their original export status unless the heir explicitly overrides it (through a **New_exports** clause); in contrast, a creation procedure loses its creation status unless the heir explicitly reaffirms it (by listing the procedure in its own **Creators** part). This difference follows from an analysis of what designers most commonly need, in each case, in the practice of building systems.

← "*Adapting the export status of inherited features*", . . . page 204.

20.6 USING AN EXPLICIT TYPE

In the variants seen so far, the type of the object created by a creation instruction **create** *x* . . . , with or without an explicit creation procedure, is the type *T* declared for *x*, the instruction's target. You may want to use another type *V* instead; this will be permitted if *V* conforms to *T*. The form of the instruction in this case is one of

```
create {V} x.cp (..)
create {V} x
```

with the first one valid only if *cp* is a creation procedure of *V*, and the second only if *default_create* is a creation procedure of *V* (in particular if *V*'s base class has no **Creators** part).

Specifying the creation type



Assume class *SEGMENT* is a descendant of *FIGURE*, and has a creation procedure *make*, with two formal arguments of type *POINT* representing the end points of a segment. The following will be valid:

```
[1]
  fig: FIGURE
  point1, point2: POINT
  ...
  create {SEGMENT} fig.make (point1, point2)
```

and will have exactly the same effect on *fig* as

```
[2]
  fig: FIGURE; seg: SEGMENT
  point1, point2: POINT
  ...
  create seg.make (point, point2)
  fig := seg
```

where the last instruction is a polymorphic assignment, permitted by the [Assignment rule](#) since *seg* conforms to *fig*.

→ The Assignment rule, stating that the type of an assignment's source must conform to that of its target, is on page 590.

The explicitly typed form [1](#) brings nothing fundamentally new; it is just an abbreviation for the implicitly typed form [2](#), avoiding the need to introduce intermediate entities such as *seg*.

As a consequence of this new form, we can define the **creation type** of a creation instruction — the type of the object that it will create: in the previous form `create x ...`, the creation type is the type declared for the target, *x*; in the explicit form `create {V} x ...`, the creation type is *V*.

→ The formal definition will appear on page 551.

Choosing between types

To become really useful the example should include more than one case: after all, if all you ever want to obtain is an instance of *SEGMENT*, then you do not need *fig*; *seg* suffices. Things become more interesting with a scheme of the following kind, using a local variable *fig* of type *FIGURE*:

```
[3]
inspect
  icon_selected_by_user
when Segment_icon then
  create {SEGMENT} fig.make (point1, point2)
when Triangle_icon then
  create {TRIANGLE} fig.make (point1, point2, point3)
when Circle_icon then
  create {CIRCLE} fig.make (point1, radius)
when ...
  ...
end
```

Here *SEGMENT*, *TRIANGLE*, *CIRCLE*, ... are descendants of *FIGURE*, all with specific creation procedures, and *Segment_icon*, *Triangle_icon*, *Circle_icon*, ... are integer constants with different values. Depending on the icon selected by an interactive user, the above instruction creates an object of the appropriate type, and attaches *fig* to it.

Were the explicitly typed form of the creation instruction not available, you could still use the equivalence illustrated by [2](#), rather unpleasant here because you need to declare a temporary entity (*seg*, *tri*, *circ*, ...) for each of the possible icon types.

Creation and deferred classes



Scheme [3](#) helps understand the role of **deferred classes and types** vis-à-vis creation. A class must be declared as **deferred** if it has at least one deferred feature (introduced in the class itself, or inherited from a parent, and not effected — made effective — in the class). A deferred type is one based on a deferred class. In our example we may assume *FIGURE* to be deferred, but the concrete descendants used in the creation instructions — *SEGMENT* and so one — to be effective. The rule is that:

← Although a class may be declared as **deferred** even without deferred features, the common case is for a deferred class to have one or more deferred features. See [10.11, page 272](#).

- We *never* permit a creation instruction to use a deferred type as creation type. As noted in the last chapter, creating direct instances of a deferred type would be asking for trouble, since clients could then call unimplemented operations on these instances. The creation rules of this chapter exclude this possibility; with *fig* of type *FIGURE*, we are not permitted to write **create** *fig* ... , with or without a creation procedure.
- We may, however, use *fig* as target of a creation instruction such as **create** {*SEGMENT*} *fig*.*make* (*point1*, *point2*) or any of the others above, even though the type of *fig* is deferred: that's fine as long as the **creation type** of the instruction is explicit and effective, like *SEGMENT* here. The instruction will create a direct instance of that type, so everything is in order. Attaching this object to an entity *fig* of a deferred type is also in order: it's simply an application of polymorphism.

“Direct instance” is in fact not even defined for deferred types. See “[INSTANCES AND VALUES](#)”, [11.5, page 329](#).

In summary: we cannot create **objects** of deferred types, but we can have **entities** of such types, which will become attached to instances of conforming effective types.

Single choice and factory objects



Beyond its applicability to polymorphic entities of deferred types, what makes scheme [3](#) especially interesting is its connection with **dynamic binding**: after executing the above *Multi_branch* instruction, you normally should never have to discriminate again on the type of *fig*; instead, to apply an operation with different variants for the figures involved, you should use a call of the form

fig.*display*

where the operation, here *display*, is redefined in various ways in descendants of *FIGURE*. This will select the appropriate version depending on the exact type of the object to which *fig* is attached, as a result of the variable-type creation achieved by 3.

This example illustrates an important concept of Eiffel software development: the **Single Choice principle**. The principle states that in a software system that handles a number of variants of the same notion (such as the figure types in a graphics system) any exhaustive knowledge of the set of possible variants should be confined to just **one component** of the system. This is essential to prevent future additions and modifications from requiring extensive system restructuring.

See also 17.6, page 491, on explicit discrimination. For further discussion of these issues see “Object-Oriented Software Construction”, in particular the Open-Closed Principle. .

Often, the component that performs the “Single Choice” will be the one that initially creates instances of the appropriate objects; 3 illustrates one of the possible schemes.

There is a simpler scheme, avoiding any explicit control structure: the *clonable array technique*, implementing what the Design Pattern literature calls the **Factory Pattern**, although it was described in Eiffel literature and widely used in Eiffel programs many years before that term appeared in print.

Here is how it would work in this example. You assign a unique code to every variant

Low_id, Segment_id, Triangle_id, Circle_id, ... , High_id:
-- REDO EXAMPLE -----

and create a data structure, most conveniently an array, containing one direct instance of each variant:

```
[4]
  figure_factory: ARRAY [FIGURE]
  local
    fig: FIGURE
  once
    Result.make (Low_id, High_id)

    -- Create and enter a SEGMENT instance:
    create {SEGMENT} fig.make (...)
    Result.put (fig, Segment_id)

    -- Create and enter a TRIANGLE instance:
    create {TRIANGLE} fig.make (...)
    Result.put (fig, Triangle_id)

    ... Do the same for each variant ...

  end
```

*WARNING: there is a much more concise way to express this, using creation expressions and avoiding altogether the need to declare a local variable *fig*. See 1, page 559, which is the model you should use for this pattern.*



Instead of making `figure_factory` a once function you can declare it as an attribute, and then initialize it accordingly (with the instructions of the above routine body, substituting `figure_factory` for `Result`) in an initialization module. But initialization modules that take care of initializations for many different aspects of a system are not good for modular, extensible software construction. Using a once function is usually a better approach since it has the same effect but lets the initialization happen automatically the first time any part of the system needs to access `figure_factory`.

Then, whenever you actually need to select an alternative, you can avoid the explicit discrimination of **3**: replace the *entire* `Multi_branch` instruction by

[5]

```
fig := clone (figure_factory @ code)
```

where `code` is the desired figure code (one of `Segment_id`, `Triangle_id` etc.). The function `clone` appearing on the right-hand side produces a new object copied from its argument; so each time you use **5** you get a new object which, depending on the value of the index `code`, will be a `SEGMENT`, or a `TRIANGLE` and so on.

figure_factory @ code denotes the item of index `code`, also written `figure_factory.item(code)`; see [36.4, page 934](#).

→ “[CLONING AN OBJECT](#)”, [21.4, page 575](#).

20.7 RESTRICTING CREATION AVAILABILITY



The `Creators` parts in the preceding examples had at most one `Creation_clause`, and any client could create direct instances through any of the creation procedures listed there. It is also possible to define more restrictive client creation privileges. Let us take a look at this simple facility which, although not needed in elementary uses, helps build well-engineered systems that thoroughly apply the principle of [information hiding](#).

← See [7.8, page 200](#), on information hiding.

You may indeed write a `Creators` part with one or more `Creation_clause` listing procedures available for creation by specific clients, as in



```
class C ... create
  make
create {A, B}
  jump_start, bootstrap
feature
  ...
end
```

The first `Creation_clause` has no restriction, so that any client can create a direct instance of `C` through an instruction `create x.make (...)` for `x` of type `C`. Because of the restriction in the second clause, however, only the descendants of `A` and `B` may use the given procedures for creation, in instructions `create x.jump_start (...)` or `create x.bootstrap (...)`.

Remember that descendants of a class include the class itself.

This possibility of including more than one **Creation_clause**, each specifying that certain procedures of the class are creation procedures and giving a creation availability status, is, as you will certainly have noted, patterned after the convention for making the features of a class available to clients with a specified export status for calls. In the same way that a **Feature_clause** may begin with one of



- [1]
 - feature**
 - ... Declaration of features callable by all clients ...
 - feature {NONE}**
 - ... Declaration of features callable by no clients ...
- [2]
 - feature {X, Y}**
 - ... Declaration of features callable by descendants of X and Y ...

a **Creation_clause** may begin with one of



- [1]
 - create**
 - ... List of procedures available for creation to all clients ...
- [2]
 - create {NONE}**
 - ... List of procedures available for creation to no clients ...
- [3]
 - create {X, Y}**
 - ... List of procedures available for creation to descendants of X and Y ...



Note, however, that such flexibility is not as essential for creation as it is for feature call. As part of the fundamental O-O principles of abstraction and information hiding, it is common to have several feature clauses specifying different levels of call availability: to all clients, to some clients, to no clients. This is less frequently useful for creation, and in practice many classes have just one **Creation_clause**, or none.



The language supports the full generality of the mechanism anyway, partly for consistency with the other mechanism, and partly because the extra control over creation availability is occasionally useful.



Make sure not to confuse the two forms of specifying availability. When you list a set of creation procedures, as in **1**, **2** and **3** for a class *C*, you are only controlling the validity of a **Creation_instruction** involving a creation call, such as

```
[1]
create x.cp (...)
```

for *x* of type *C*: valid everywhere in case **1**, invalid everywhere with **2**, and valid only in descendants of *X* and *Y* with **3**. This is completely independent of the availability status for plain (non-creation) calls such as

```
[1]
x.cp (...)
```

valid everywhere in case **1**, invalid everywhere with **1**, and valid only in descendants of *X* and *Y* with **2**. For the same *cp*, the two properties are separate. They reflect different semantics:

- The creation call **create** *x.cp* (...) creates an object and initializes it using *cp*.
- The plain call *x.cp* (...) uses *cp* to reinitialize an existing object – a right which, as the designer of a class, you may decide to grant or not to grant to clients, regardless of the right you have granted regarding the use of *cp* for creation-time initialization.

You may indeed be justified in deciding on different privileges in each case. Consider a class manipulating bank accounts:



```
class
  ACCOUNT
create
  make
feature {NONE} -- Initialization
  make (initial: AMOUNT)
    -- Set balance to initial.
  is do ... end
feature -- Element change
  withdraw (a: AMOUNT)
    -- Record removal of a units of currency.
  do ... end
  deposit (a: AMOUNT)
    -- Record addition of a units of currency.
  do ... end
  ... Other features, invariant ...
end-- class ACCOUNT
```

The use of **feature** {*NONE*} for the declaration of the class's creation procedure is a common Eiffel idiom, but surprising at first here: why hide this fundamental operation on the class? The reason is that we are hiding it for call, not for creation. The **Creation_instruction**

```
create your_account.make (some_amount)
```

is indeed valid since *make* appears in an unrestricted **Creators** clause (lines 3 and 4, highlighted in the class above). What is **not** valid is a plain call

```
your_account.make (some_amount)
```

WARNING: not valid with class text as given.

which would reinitialize the account to *some_amount*. The author of class *ACCOUNT* has decided that the only way to affect the balance of an account is to deposit or withdraw money (adding a value, positive or not, to the balance, rather than setting it to a specified value). Such policies are often legitimate and explain why **feature** {*NONE*} is a common style for declaring a creation procedure, even one that is unrestrictedly available for creation.

20.8 THE CASE OF EXPANDED TYPES

---- THIS SECTION IS NOW WRONG, REWRITE (lazy initialization) --

The preceding examples assumed that the type of the target entity was a reference (non-expanded) type. What if it is expanded?

In this case there is no need to create an object, since the value of the target is already an object, not a reference to an object that a **Creation_instruction** must allocate dynamically.

Rather than disallowing **Creation_instruction** for expanded targets, it is convenient to define a simple semantics for the instruction in this case, limited to the steps of the above process that still make sense: the instruction will execute the default initializations on the object attached to the target, then call the appropriate version of *default_create*. This convention also has the advantage that if you change your mind about the expanded status of a class you can change it without to worry about its **Creation_clause** becoming invalid.

As a consequence of this rule, if we have a class whose instances contain sub-objects, as in

```
class COMPOSITE feature
  a: SOME_REFERENCE_TYPE
  b: SOME_EXPANDED_TYPE
  ...
end
```



then the default initialization rule for the *b* field of a *COMPOSITE* instance will be to apply a *Creation_instruction*, recursively, to the corresponding sub-object. This creation instruction will use as creation procedure the version of *default_create* in the corresponding base class.

--- NO LONGER QUITE TRUE, REWRITE -----This semantic rule justifies a basic constraint on expanded types (given in the chapter on classes as the *Class Header rule*): the base class of an expanded type **must** have its version of *default_create* as one of its creation procedures (either explicitly in its *Creators* part, or implicitly by not having a *Creators* part). This does not prevent the class from having other creation procedures if desired; but for automatic initialization of sub-objects such as *b* the procedure to be applied is *default_create*, as any other choice would require further information from the client (choice of creation procedure and actual arguments). ← Page 126,

20.9 CREATING INSTANCES OF FORMAL GENERICS

More delicate than the expanded types is the case in which we would like to create an instance of one of the *Formal_generic* parameter types of a class, as in *create x..* where *x* is of type *G* in a class *C [G]*.

The problem is that *G*, in the class text, denotes not a known type but a placeholder for many possible types or, in the case of unconstrained genericity, *any* valid type. So we have no way to know what creation procedures will be available on the corresponding instances.

This seems at first to preclude any hope of allowing creation instructions in this case. Fortunately, constrained genericity allows an elegant solution.

As you know, constrained genericity is the mechanism that allows us to declare a class as

```
class C [G → CONST] ...
```

where *CONST* is a type, known as the constraining type for the formal generic parameter *G*. Then you may only write a generic derivation *C [T]*, using a type *T* as actual generic parameter, if *T* conforms to *G*. The benefit is that, within class *C*, you know that any entity *x* of type *G* represents objects of type *T* or conforming, so you may apply to *x* any of the features of *T* — rather than being limited, as in the unconstrained case *C [G]*, to the features of class *ANY*, applicable to all types.

A small syntactic extension enables us to take advantage of constrained genericity to allow creation of objects of generic type. Declare the class as

```
class D [G → CONST create cp1, cp2, ... end] ...
```

← “*CONSTRAINED GENERICITY*”, 12.6, page 354,



to state that G represents any type that both:

- (As always with constrained genericity) conforms to $CONST$.
- Admits as creation procedures its versions of $cp1, cp2, \dots$, which must be procedures of $CONST$.

These obligations are enforced: a generic derivation $D [T]$ will only be valid if (as always) T conforms to $CONST$ and, in addition, the given procedures $cp1, cp2, \dots$ are creation procedures of T . More precisely, their **versions** in T — which may differ from the originals versions in $CONST$ as a result of renaming, redefinition and effecting — must be listed among the creation procedures of T .

With D declared as shown, it becomes possible, for x declared of type G in the text of class D itself, to use a creation instruction

```
create x.cpi (args)
```

where cp_i is one of the procedures of D listed in the **create ... end** part for $CONST$ as shown above, and $args$ is a valid argument list for that procedure. The instruction will always make sense dynamically since, thanks to the preceding rule, the type T of x — in any valid generic derivation $D [T]$ — will always be a descendant of $CONST$, so that:

- cp_i will be one of its procedures, taking the appropriate arguments.
- T will have listed cp_i as one of its creation procedures (hence, among other properties, we may expect that cp_i ensures the invariant of T).

As a special case, you can permit the procedure-less form **create** x by including *default_create* (rather, its name in $CONST$) among the cp_i .

What's particularly useful in this mechanism is that at the level of D we only require the listed cp_i to be **procedures** of the constraining type $CONST$ — so that we can ascertain, from D 's text only, the validity of $args$ as arguments in the creation call **create** $x.cp_i (args)$: we do not require the cp_i to be **creation procedures** of $CONST$. This last requirement will only come up where it matters: in types T , descendants of $CONST$ used in actual generic derivations $D [T]$. In such a T , the local version of cp_i must indeed be one of T 's creation procedures.

This means in particular that the above scheme will work even if *CONST* is deferred, as in



```

class
  D [G → CONST create cp end]
feature
  some_routine
  local
    x: G
  do
    create x.cp (3)
  end
end
deferred class CONST feature
  cp (n: INTEGER)
  ... Could be effective or deferred ...
  end
  ... Other features, possibly including deferred ones ...
end

```

We don't care that the boxed creation instruction works on a target x whose type G is based on a deferred class *CONST*, and that the creation procedure *cp* might itself be deferred in *CONST*: any type T used for G in practice must make its version of *cp* a creation procedure. This implies among other things that T is an effective class and *cp* an effective procedure, so everything will work properly.



Note that this creation mechanism for formal generics assumes *constrained* genericity. In a class $C [G]$, where G is an *unconstrained* generic parameter, no creation instruction **create** $x \dots$ is valid for x of type G . This includes the procedure-less form **create** x : making it valid would mean assuming that *default_create* will be a creation procedure in all possible types — certainly not true. You can, however, write the class as

```

class C [G → ANY create default_create end]

```

thereby unfolding unconstrained genericity into its constrained equivalent. Then the generic derivation $C [T]$ will be valid for a type T if and only if T 's base class doesn't list any creation procedures, or lists *default_create* among its creation procedures. With this form of C 's declaration, **create** x is valid in the text of class C .



More generally, remember that the procedure-less form `create x` is only valid, for x of a formal generic type, if you have explicitly listed `default_create` (under its local name) in a `create` subclause after the constraint. There is no equivalent here to the implicit rule of the `Creators` part, where requesting no creation procedures means requesting `default_create` only. For generic parameters, you don't get creation privileges unless you specify them expressly.

20.10 PRECONDITIONS OF CREATION PROCEDURES

The creation process, when it involves a creation procedure, applies it to an object caught in its virginal state, just after default initializations. Such a state does not, in general, satisfy the class invariant; it is indeed the very purpose of the creation procedure to ensure the invariant from the first time.

A consequence of dealing with an object in such a fragile temporary state is that the creation procedure must refrain, if it has a precondition, from including in it certain properties that are meaningful only in later stages of the object's life. In particular

- The precondition should not use any feature of the object, since the client could not legitimately access the value of that feature to ensure the precondition. Assume for example a creation procedure `cp` with a precondition clause $a > 0$ where a is an attribute; the client should be able, before a creation instruction `create x.cp (...)`, to test for $x.a > 0$, but this makes no sense since the required object doesn't exist yet. So we must prohibit the use of any `Unqualified_call`, to a feature of any kind, in the precondition.
- For the same reason, we must prohibit any use of `Current`, denoting a current object that doesn't exist yet.

The precondition can still refer to any properties of the creation procedure's arguments, including through feature calls on these arguments.

In addition, we have a requirement similar to the general rule for feature availability in feature calls. That rule specified that any feature p used in the precondition of a feature f must be available to all the clients to which f itself is available, so that any client that may call $x.f(...)$ may also check for $x.p$. In the case of a creation instruction `create x.cp (...)`, we have seen that a creation procedure `cp` must be "available for creation" to the client; to any such client, p has to be available (for call). This is a new requirement since it is possible for `cp` to be "available for creation" to a client, but not available for call.

← *Precondition Export rule: VAPE, page 237.*

← *"RESTRICTING CREATION AVAILABILITY", 20.7, page 539*

These observations lead to a rule on the precondition clauses of any routine used as a creation procedure:



Creation Precondition rule *VGCP*

A **Precondition** of a routine *r* is **creation-valid** if and only if its unfolded form *uf* satisfies the following conditions:

- 1 • The predefined entity **Current** does not appear in *uf*.
- 2 • No **Unqualified_call** appears in *uf*.
- 3 • Every feature whose final name appears in the *uf* is available to every class to which *r* is available for creation.

This definition is not itself a validity constraint, but is used by condition 5 of the Creation Clause rule below; giving it a code as for a validity constraint enables compilers to provide a precise error message in case of a violation. → *VGCC, page 548.*

Requiring preconditions to be creation-valid will ensure that a creation procedure doesn't try to access, in the object being created, fields whose properties are not guaranteed before initialization.

The definition relies on the “unfolded form” of an assertion, which reduces it to a boolean expression with clauses separated by **and then**. Because the unfolded form uses the Equivalent Dot Form, condition 3 also governs the use of operators: with *plus alias* “+”, the expression *a + b* will be acceptable only if the feature *plus* is available for creation as stated.

20.11 CREATION SYNTAX AND VALIDITY



Here now are the precise rules applying to **Creators** parts and **Creation** instructions. This section only formalizes previously introduced concepts, so on first reading you may skip this section and the next two (which formalize the semantics). *If skipping, go to “CREATION EXPRESSIONS AND ANONYMOUS OBJECTS”, 20.14, page 558.*

First, the syntax of a **Creators** part, an optional component of the **Class** text, appearing towards the beginning of a class, after **Inheritance** and before **Features**: *The structure of a Class text, with all its parts, is on page 119.*



Creators parts

$$\text{Creators} \triangleq \text{Creation_clause}^+$$

$$\text{Creation_clause} \triangleq \text{create} [\text{Clients}]$$

$$\qquad\qquad\qquad [\text{Header_comment}]$$

$$\qquad\qquad\qquad \text{Creation_procedure_list}$$

$$\text{Creation_procedure_list} \triangleq \{\text{Creation_procedure} \text{ , } \dots \}^+$$

$$\text{Creation_procedure} \triangleq \text{Feature_name}$$

The optional Header_comment emphasizes the similarity with the syntax of a Feature_clause, given page 137.

To talk about the validity and semantics of creation clauses and creation instructions, it is useful to take care once and for all of the special case of *default_create* as creation procedure through the following definition:



Unfolded Creators part of a class

The **unfolded creators part** of a class *C* is a **Creators** defined as:

- 1 • If *C* has a **Creators** part *c*: *c*.
- 2 • If *C* is deferred: an empty **Creators** part.
- 3 • Otherwise, a **Creators** part built as follows, *dc_name* being the final name in *C* of its version of *default_create* from **ANY**:
 create
 dc_name

For generality the definition is applicable to any class, even though for a deferred class (case 2) it would be invalid to include a **Creators** part. This causes no problem since the rules never refer to a deferred class actually extended with its unfolded creators part.

Case 3 reflects the convention that an absent **Creators** part stands for **create** *dc_name* — normally **create** *default_create*, but *dc_name* may be another name if the class or one of its proper ancestors has renamed *default_create*.

← Discussed informally in previous sections.

With this we can define the constraint on **Creators** part of a class:



Creation Clause rule

VGCC

A **Creation_clause** in the unfolded creators part of a class *C* is valid if and only if it satisfies the following conditions, the last four for every **Feature_name** *cp_name* in the clause's **Feature_list**:

- 1 • *C* is effective.
- 2 • *cp_name* appears only once in the **Feature_list**.
- 3 • *cp_name* is the final name of some procedure *cp* of *C*.
- 4 • *cp* is not a once routine.
- 5 • The precondition of *cp*, if any, is creation-valid.

As a result of conditions [1](#) and [4](#), a creation procedure may only be of the **do** form (the most common case) or **External**.

The prohibition of **once** creation procedures in condition [4](#) is a consequence of the Creation principle: with a once procedure, the first object created would satisfy the invariant (assuming the creation procedure is correct), but subsequent creation instructions would not execute the call, and hence would limit themselves to the default initializations, which might not ensure the invariant.

As a corollary of condition [4](#), a class that has no explicit **Creators** part may not redefine *default_create* into a once routine, or inherit *default_create* as a once routine from one of its deferred parents. (Effective parents would themselves violate the condition and hence be invalid.)

Condition [5](#) is the rule on preconditions of creation procedures, whose rationale was discussed in the preceding [section](#).

← “[PRECONDITIONS OF CREATION PROCEDURES](#)”, [20.10](#), [page 546](#).



To complement this study of the syntax and semantics of **Creators** parts, it is useful to remind ourselves of their counterpart for generic parameters: the **Constraint_creators** subclause of the syntax for generic constraints, a simplified form of the **Creators** part. Here is the relevant syntax:



```

Formal_generics  $\triangleq$  "["Formal_generic_list"]"
Formal_generic_list  $\triangleq$  [Formal_generic", "...]
Formal_generic  $\triangleq$  [frozen] Formal_generic_name
                    [Constraint]
Formal_generic_name  $\triangleq$  Identifier
Constraint  $\triangleq$  "->"Class_type [Constraint_creators]
Constraint_creators  $\triangleq$  create Feature_list end
  
```

← This was first seen in the chapter on types; syntax on [page 351](#), validity in “[CONSTRAINED GENERICITY](#)”, [12.6](#), [page 354](#).

The applicable validity rule there was that the elements of the **Feature_list** must be the names of distinct procedures of the constraining type — corresponding to clauses [1](#) and [2](#) of the Creation Clause rule above. There was no need for an equivalent to the other clauses since they are taken care of by the Creation Clause rule itself when we provide an actual generic parameter conforming to the constraining type.



A language design note: it would have been possible to use **Creators** for **Constraint_creators**, permitting a more flexible form of creation availability specification for a generic parameter — with more than one **Creation_clause**, each listing specific clients and procedures. This would in fact make the language definition simpler by avoiding the construct **Constraint_creators**. The extra capabilities, however, seems useless, and could yield unduly complicated **Formal_generics** parts, so the language sticks to a primitive form of **Constraint_creators** for generic parameters.

The Creation Clause rule allows us to define the set of creation procedures of a class:



Creation procedures of a class

The **creation procedures** of a class are all the features appearing in any **Creation_clause** of its unfolded creators part.

If there is an explicit **Creators** part, the creation procedures are the procedures listed there. Otherwise there is only one creation procedure: the class's version of *default_create*.

The following property is a consequence of the definitions of “unfolded creators part” and “creation procedures of a class”.

Creation procedure property

An effective class has at least one creation procedure.

Those explicitly listed if any, otherwise *default_create*.

Only in the first case (explicit **Creators** part) can the set of creation procedures be empty: this is achieved, as we have seen, by including a **Creators** part, but an empty one, listing no name at all.

← See the example class *NOT_INSTANTIABLE* on page 533.

We need a small refinement of this definition to extend it to the case of types, to support the mechanism for creation on generic parameters:

← See “*CREATING INSTANCES OF FORMAL GENERICS*”, 20.9, page 543.



Creation procedures of a type

The **creation procedures** of a type *T* are:

- 1 • If *T* is a **Formal_generic_name**, the constraining creators for *T*.
- 2 • Otherwise, the creation procedures of *T*'s base class.



The definition of case 2 is not good enough for case 1, because in the scheme **class *D* [*G* → *CONST* create *cp1*, *cp2*, ... end]** it would give us, as creation procedures of *G*, the creation procedures of *CONST*, and what we want is something else: the set of procedures *cp1*, *cp2*, ... specifically listed after *CONST* — the “*constraining creators for G*”. These are indeed procedures of *CONST*, but they are not necessarily *creation* procedures of *CONST*, especially since *CONST* can be deferred. What matters is that they must be creation procedures in any instantiatable descendant of *CONST* used as actual generic parameter for *G*.

Other useful definitions:



Available for creation; general creation procedure

A creation procedure of a class *C*, listed in a *Creation_clause* *cc* of *C*'s unfolded creators part, is **available for creation** to the descendants of the classes given in the *Clients* restriction of *cc*, if present, and otherwise to all classes.

If there is no *Clients* restriction, the procedure is said to be a **general creation procedure**.

As with a Feature_clause, the absence of a Clients restriction is equivalent to a restriction of the form {ANY}.



Remember, once again, that the descendants of a class include the class itself. A *Creation_clause* with no *Clients* part, as in **create** *cp1*, *cp2*, ..., is a shortcut for one with a *Clients* part listing only *ANY*, as in **create** {*ANY*} *cp1*, *cp2*, ...

Now for the *Creation_instruction* itself, starting with its syntax:



Creation instructions

Creation_instruction \triangleq **create** [*Explicit_creation_type*]
Creation_call

Explicit_creation_type \triangleq "{" *Type* "}"

Creation_call \triangleq *Variable* [*Explicit_creation_call*]

Explicit_creation_call \triangleq "." *Unqualified_call*

Every creation instruction has a *creation type*, explicit or implicit:



Creation target, creation type

The **creation target** (or just "target" if there is no ambiguity) of a *Creation_instruction* is the *Variable* of its *Creation_call*.

The **creation type** of a creation instruction, denoting the type of the object to be created, is:

- The *Explicit_creation_type* appearing (between braces) in the instruction, if present.
- Otherwise, the type of the instruction's target.

so that in



```
account1: ACCOUNT; point1, point2: POINT; figure1: FIGURE
...
create account1
create point1.make_polar (1, Pi/4)
create {SAVINGS_ACCOUNT} account1
create {SEGMENT} figure1.make (point1, point2)
```

the creation types for the four instructions are *ACCOUNT*, *POINT*, *SAVINGS_ACCOUNT* and *SEGMENT*. The targets are *account1*, *point1*, *account1* and *figure1*.

The creation type of a *Creation_instruction* is the type of the objects that it may create. It will always satisfy the following property:

Creation Type theorem

The creation type of a creation instruction is always effective.

This theorem is corollary **1** of the Creation Instruction rule, seen next. That rule will need one more auxiliary definition: → *The corollary is on page 555.*



Unfolded form of a creation instruction

Consider a *Creation_instruction* *ci* of creation type *CT*. The **unfolded form** of *ci* is a creation instruction defined as:

- 1 • If *ci* has an *Explicit_creation_call*, then *ci* itself.
- 2 • Otherwise, a *Creation_instruction* obtained from *ci* by making the *Creation_call* explicit, using as feature name the final name in *CT* of *CT*'s version of *ANY*'s *default_create*.

This definition parallels the earlier one of “unfolded creators part of a class” and expresses the property, stated informally before, that we understand the procedure-less form of creation **create** *x* as a shortcut for **create** *x.default_create* (with the new name for *default_create* if different).



A final notion that the Creation Instruction rule will need is a property defined only in a subsequent chapter, but already presented informally in page 634: → *“Argument rule”*. the discussion of calls, and in fact rather obvious: the concept of a call being **argument-valid**. This property is part of the more complete definition of call validity; it states that in a call *x.f(a, b, c)* where *x* is of type *T* and *f* is a feature of *T* with formal arguments *u1: T1; u2: T2; u3: T3*,

the number of actual arguments a, b, c must be the same as the number of these formal arguments, here three, and each actual's type must conform to the corresponding formal's type — here the type of a to $T1$, of b to $T2$, and of c to $T3$. We of course expect this fundamental property to hold for all calls, and must enforce it for a creation instruction `create x.f(a, b, c)` involving a `Creation_call`. This is clause 3 of the following rule.

We indeed by now have enough preparation to express the validity rule for creation instructions:



Creation Instruction rule

VGCI

A `Creation_instruction` of creation type CT , appearing in a class C , is valid if and only if it satisfies the following conditions:

- 1 • CT conforms to the target's type.
- 2 • The feature of the `Creation_call` of the instruction's unfolded form is available for creation to C .
- 3 • That `Creation_call` is argument-valid.
- 4 • CT is generic-creation-ready.

→ Another version of this rule appears below, page 555, with clauses labeled by numbers rather than letters.

I can see that puzzled look on your face: surely, with all the possibilities seen in this chapter, the complete validity constraint for creation instructions must be longer?

In spite of its compactness, the Creation Instruction rule suffices in fact to capture all properties of creation instructions thanks to the auxiliary definitions of “creation type”, “unfolded form” of both a `Creation_instruction` and a `Creators` part, “available for creation” and others. The rule captures in particular the following cases:

- The procedure-less form `create x` is valid only if CT 's version of `default_create` is available for creation to C ; this is because in this case the unfolded form of the instruction is `create x.dc_name`, where `dc_name` is CT 's name for `default_create`. On CT 's side the condition implies that there is either no `Creators` part (so that CT 's own unfolded form lists `dc_name` as creation procedure), or that it has one making it available for creation to C (through a `Creation_clause` with either no `Clients` specification or one that lists an ancestor of C).
- If CT is a `Formal_generic_name`, its creation procedures are those listed in the `create` subclause after the constraint. So `create x` is valid if and only if the local version of `default_create` is one of them, and `create x.cp (...)` only if `cp` is one of them.
- If CT is generically derived, and its base class needs to perform creation operations on targets of some of the formal generic types, the last condition (generic-creation readiness) ensures that the corresponding actual parameters are equipped with appropriate creation procedures.

The very brevity of this rule may make it less suitable for one of the applications of validity constraints: enabling compilers to produce precise diagnostics in case of errors. For this reason a complementary rule, conceptually redundant since it follows from the Creation Instruction rule, but providing a more explicit view, appears next. It is stated in “*only if*” style rather than the usual “*if and only if*” of other validity rules, since it limits itself to a set of necessary validity conditions.

All together, these conditions do come close to the full set of sufficient conditions listed in the first variant, but we don’t really care, since that first variant gives us the “*if and only if*” property that we need.



Creation Instruction properties VGCP

A Creation_instruction *ci* of creation type *CT*, appearing in a class *C*, is valid only if it satisfies the following conditions, assuming *CT* is not a Formal_generic_name and calling *BCT* the base class of *CT* and *dc* the version of *ANY*'s *default_create* in *BCT*:

- 1 • *BCT* is an effective class.
- 2 • If *ci* includes a Type part, the type it lists (which is *CT*) conforms to the type of the instruction's target.
- 3 • If *ci* has no Creation_call, then *BCT* either has no Creators part or has one that lists *dc* as one of the procedures available to C for creation.
- 4 • If *BCT* has a Creators part which doesn't list *dc*, then *ci* has a Creation_call.
- 5 • If *ci* has a Creation_call whose feature *f* is not *dc*, then *BCT* has a Creators part which lists *f* as one of the procedures available to *C* for creation.
- 6 • If *ci* has a Creation_call, that call is argument-valid.

If *CT* is a Formal_generic_name, the instruction is valid only if it satisfies the following conditions:

- 7 • *CT* denotes a constrained generic parameter.
- 8 • The Constraint for *CT* specifies one or more procedures as constraining creators.
- 9 • If *ci* has no Creation_call, one of the constraining creators is the Constraint's version of *default_create* from *ANY*.
- 10 • If *ci* has a Creation_call, one of the constraining creators is the feature of the Creation_call.

WARNING: although this rule looks complicated, it is in fact just a series of consequences of a short and simple rule: the original "VGCI", page 553.

Compiler writers may refer, in error messages, to either these "Creation Instruction Properties" or the earlier "Creation Instruction rule" of which they are consequences. For the language definition, **the official rule is the Creation Instruction rule**, which provides a necessary and sufficient set of validity conditions.



The number of clauses in this second variant justifies a *contrario* using the first variant as the official definition. Fundamentally, the rule is straightforward once you have defined the "creation type", explicit or implicit and "unfolded" both the creation instruction and the creation type's base class to take care of the *default_create* convention, so that every class has a list of creation procedures and every creation instruction lists a creation procedure. Then the rule is simply that the creation type must be OK for the creation's

target, that the creation procedure must be available for creation, and that the call must have valid arguments. That's all. The “corollaries” form is long because it expands the various simplifications (creation type, creation procedures of a class, creation procedure of an instruction) for the various possible cases, and treats all these cases individually — accounting for various errors that an absent-minded developer might make.

20.12 CREATION SEMANTICS

SEMANTICS With the preceding validity rules, we can define the precise semantics of a `Creation_instruction`.

Creation Instruction Semantics

The effect of a creation instruction of target x and creation type TC is the effect of the following sequence of steps, in order:

- 1 • If there is not enough memory available for a new direct instance of TC , trigger an exception of type NO_MORE_MEMORY in the routine that attempted to execute the instruction. The remaining steps do not apply in this case.
- 2 • Create a new direct instance of TC , with reference semantics if CT is a reference type and copy semantics if CT is an expanded type.
- 3 • Call, on the resulting object, the feature of the Unqualified_call of the instruction's unfolded form.
- 4 • Attach x to the object.

← See [19.3, page 506](#) about a reference being attached to an object.

The rule requires the *effect* described by this sequence of steps; it does not require that the implementation literally carry out the steps. In particular, if the target is expanded and has already been set to an object value, the implementation (in the absence of cycles in the client relation between expanded classes) may **not have to allocate new memory**; instead, it may be able simply to reuse the memory previously allocated to that object. (Because only expanded types conform to an expanded type, no references may exist to the previous object, and hence it is not necessary to preserve its value.) In that case, there will always at step 1 be “enough memory available for a new direct instance” — the memory being reused — and so the exception cannot happen.

One might expect, between steps 2 and 3, a step of *default initialization* of the fields of the new object, since this is the intuitive semantics of the language: integers initialized to zero, detachable references to void etc. There is no need, however, for such a step since the Variable Semantics rule ← Page [521](#). implies that an attribute or other variable, unless previously set by an explicit attachment, is automatically set on first access. The rule implies for example that an integer field will be set to zero. More generally, the semantics of the language guarantees that in every run-time circumstance any object field and local variable, even if never explicitly assigned to yet, always has a well-defined value when the computation needs it.

About step 3, remember that the notion of “unfolded form” allows us to consider that every creation instruction has an Unqualified call; in the procedure-less form `create x`, this is a call to *default_create*. ← Page [552](#).

Also note the order of steps: attachment to the target *x* is the last operation. Until then, *x* retains its earlier value, void if *x* is a previously unattached reference.

In step 2, “not enough memory available” is a precise notion (the definition appears below); it means that even after possible *garbage collection* the memory available for the system’s execution is not sufficient for the requested object creation. → Page [564](#).

20.13 REMOTE CREATION



The syntax of creation instructions does not support “remote creation” instructions as in:

```
create x1.y1.cp (...)
```

WARNING: syntactically incorrect.

To obtain an equivalent effect, assuming that *xI* is of type *X* and that *yI* is an attribute of type *Y* in *X*, you must introduce a specific procedure in *X*



```
make_yI (arguments: ...)
  -- Attach yI to new instance of Y.
do
  create yI.cp (arguments)
end
```

so that instead of the above attempt at remote creation clients will use the instruction

```
xI.make_yI (...)
```



This is in line with the principle of information hiding: deciding whether or not clients of *X* may directly “create” the *yI* field is the privilege of the designer of *X* who, if the answer is positive, will write a specific procedure to grant this privilege — restricting its availability if desired.

20.14 CREATION EXPRESSIONS AND ANONYMOUS OBJECTS

We have seen all there is to see about creation instructions, but there remains to study a variant of the mechanism: creation *expressions*.

Creation expressions will provide us with *anonymous objects*. The objects that we produce with a creation instruction **create** *x*... have a name — *x* — in the software text. This is usually what we want, because after we have created the object we will start manipulating it in the same routine, or others of the same class. But in some cases the name is useless because all we do with the newly created object is to pass it to another software element. Having to declare a local variable *x* just for the purpose of a creation instruction is a nuisance. A small nuisance to be sure, but whatever the language can do to avoid writing useless elements will be good for the quality of your software and your schedule.

We saw an example of such a situation when examining the clonable array technique. We had the following scheme

“Language terseness and family vacations”, in SPOOF 84 (Sociology and Psychology of Object-Oriented Fanatics), Martha’s Vineyard, 1999, pp. 6574-6598.



```

figure_factory: ARRAY [FIGURE]
  local
    fig: FIGURE
  once
    Result.make (Low_id, High_id)

    -- Create and enter a SEGMENT instance:
    create {SEGMENT} fig.make (...)
    Result.put (fig, Segment_id)

    -- Create and enter a TRIANGLE instance:
    create {TRIANGLE} fig.make (...)
    Result.put (fig, Triangle_id)

    ... Do the same for each variant ...
  end

```

← This was example 4, page 538. See simpler formulation next.

All we use *fig* for is to create successive objects — instances of descendants of *FIGURE*. But as soon as we have produced such an object with a creation instruction, we store it into the corresponding entry of the *Result* array (by passing it to the corresponding assignment procedure), and we will never, in this routine, need the object again! This is why we can reuse the same local variable, *fig*, for every *FIGURE* variant.

In this case the entity *fig* is not needed; neither is a separate creation instruction. All we really want is an expression denoting the new object, which we can directly pass to a routine or, as here, assign to an array element.

Creation expressions serve this need. They look like one of

```

[1]
  create {SOME_TYPE}

[1]
  create {SOME_TYPE}.creation_procedure (...)

```

The first variant, as you have guessed, is applicable if *SOME_TYPE*'s base class has no *Creators* part, or one that includes *default_create*; the second, if *creation_procedure* is one of its creation procedures.

Note how both variants look like a *Creation_instruction*:

- The first recalls the instruction **create** {*SOME_TYPE*} *target*, with no explicit *Creation_call*.
- The second recalls **create** {*SOME_TYPE*} *target*.*creation_procedure* (...).

You see the idea: starting from a creation instruction, you will get a creation expression simply by removing the *target* — a natural convention, since what you want is an anonymous object.

The constructs given (in any of the two forms **[1]** and **[1]**) are **expressions**, denoting values that can be assigned to a **Variable** entity, as in



```
x := create {SEGMENT}.make (point1, point2)
```

or, more commonly, passed as arguments to a routine, as in



```
segment_operation (create {SEGMENT}.make (point1, point2))
```

Expression form.

which has exactly the same effect as



```
create {SEGMENT} seg.make (point1, point2)
segment_operation (seg)
```

Instruction form.

with *seg* declared of type *FIGURE* (or directly of the ancestor type *SEGMENT*, in which case we can write the first line as just **create seg.make (point1, point2)**). With the creation expression we write a single call instead of three components — the declaration of *seg*, the creation instruction, and the call.

A difference with creation instructions is that for creation expressions you may not omit the **Explicit_creation_type**, *SOME_TYPE* or *SEGMENT* in the examples above. This is precisely because the created objects are anonymous. In the instruction **create target ...**, if no type is specified, we use as creation type the type of *target*; but for a creation expression there is no named *target*, so you **must** specify **{SOME_TYPE}** in all cases.

Here is the clonable array extract rewritten with creation expressions:



```
figure_factory: ARRAY [FIGURE]
  once
    Result.make (Low_id, High_id)

    -- Create and enter an instance of each desired kind:
    Result.put (create {SEGMENT}.make (...), Segment_id)
    Result.put (create {TRIANGLE}.make (...), Triangle_id)
    ... Similarly for each variant ...

  end
```

← The original was example 4, page 538, repeated above on page 559. To use the array, use clone operations; see

The comparison with the original form clearly shows the advantage of creation expressions in such a case. It's not so much a matter of writing *less*, since Eiffel is happy to be verbose when needed, as when specifying type properties of every entity, or expressing clear control structures. Rather, it's about avoiding elements that bring no useful information and can in fact, through their verbosity, obscure the text.



Note, however, that creation expressions are useful only in the special case of creating an object for the sole purpose of passing it to another software element, without using it further in the given routine. In every other situation — that is to say, in the vast majority of object creation needs — you should use a creation *instruction*.

Do not then be misled by the observation that you can rewrite any creation instruction

[1]
create *x...*

Instruction form.

as

[1]
x := **create** {*X_TYPE*} ...

Expression form.
WARNING: *this is not the recommended style.*

If you are going to do anything else with *x*, you should stay with the first form. In any case it saves you the need to specify *X_TYPE*, which you have already specified as the type of *x* in its declaration.

In summary: reserve creation expressions for anonymous objects. This important methodological note is in line with the general Eiffel principle that the language should provide *one* good way to address any specific need. Both creation expressions and creation instructions are useful, each appropriate in a different situation.

The syntax, validity and semantics of creation expressions will now follow without further comment, since they are directly deduced from the corresponding properties of creation instructions.



Creation expressions

Creation_expression \triangleq **create** Explicit_creation_type
[Explicit_creation_call]

← Explicit_creation_type, was defined on page 551 as {Type}.

The concepts introduced for creation instructions transpose directly here:

Properties of a creation expression

The **creation type** and **unfolded form** of a creation expression are defined as for a creation instruction.

The validity rule is also similar:

← “*Creation Instruction rule*”, page 553.



Creation Expression rule VGCE

A *Creation_expression* of creation type *CT*, appearing in a class *C*, is valid if and only if it satisfies the following conditions:

- 1 • The feature of the *Creation_call* of the expression’s unfolded form is available for creation to *C*.
- 2 • That *Creation_call* is argument-valid.
- 3 • *CT* is generic-creation-ready.

Here too it is useful to have an “*only if*” version:



Creation Expression Properties VGCX

A *Creation_expression* *ce* of creation type *CT*, appearing in a class *C*, is valid only if it satisfies the following conditions, assuming *CT* is not a *Formal_generic_name* and calling *BCT* the base class of *CT* and *dc* the version of *ANY*’s *default_create* in *BCT*:

- 1 • *BCT* is an effective class.
- 2 • If *ce* has no *Explicit_creation_call*, then *BCT* either has no *Creators* part or has one that lists *dc* as one of the procedures available to C for creation.
- 3 • If *BCT* has a *Creators* part which doesn’t list *dc*, then *ce* has an *Explicit_creation_call*.
- 4 • If *ce* has an *Explicit_creation_call* whose feature *f* is not *dc*, then *BCT* has a *Creators* part which lists *f* as one of the procedures available to C for creation.
- 5 • If *ce* has an *Explicit_creation_call*, that call is argument-valid.

If *CT* is a *Formal_generic_name*, the expression is valid only if it satisfies the following conditions:

- 6 • *CT* denotes a constrained generic parameter.
- 7 • The *Constraint* for *CT* specifies one or more procedures as constraining creators.
- 8 • If *ce* has no *Creation_call*, one of the constraining creators is the *Constraint*’s version of *default_create* from *ANY*.
- 9 • If *ce* has a *Creation_call*, one of the constraining creators is the feature of the Creation_call.

WARNING: a more concise form of this rule appears just before.

← See “*Creation Instructionproperties*”, page 555.

As with the corresponding “Creation Instruction Properties”, this is not an independent rule but a set of properties following from previous constraints, expressed with more detailed requirements that may be useful for error reporting by compilers. ← “Creation Instruction properties”, , page 555.



Finally, the semantics:

Creation Expression Semantics

The value of a creation expression of creation type *TC* is — except if step 1 below triggers an exception, in which case the expression has no value — a value attached to a new object as can be obtained through the following sequence of steps:

- 1 • If there is not enough memory available for a new direct instance of *TC*, trigger an exception of type *NO_MORE_MEMORY* in the routine that attempted to execute the expression. In this case the expression has no value and the remaining steps do not apply.
- 2 • Create a new direct instance of *TC*, with reference semantics if *CT* is a reference type and copy semantics if *CT* is an expanded type.
- 3 • Call, on the resulting object, the feature of the Unqualified_call of the expression’s unfolded form.

The notes appearing after the Creation Instruction Semantics rule also apply here. ← “Creation Instruction Semantics”, , page 556.

20.15 GARBAGE COLLECTION

DEFINITION

Garbage Collection, not enough memory available

Authors of Eiffel implementation are required to provide **garbage collection**, defined as a mechanism that can reuse for allocating new objects the memory occupied by unreachable objects, guaranteeing the following two properties:

- 1 • *Consistency*: the garbage collector never reclaims an object unless it is unreachable.
- 2 • *Completeness*: no allocation request for an object of a certain size s will fail if there exists an unreachable object of size $\geq s$.

Not enough memory available for a certain size s means that even after possible application of the garbage collection mechanism the memory available to the program is not sufficient for allocating an object of size s .

Comparing and duplicating objects

21.1 OVERVIEW

The just studied **Creation** instruction is the basic language mechanism for obtaining new objects at run time; it produces fresh direct instances of a given class, initialized from scratch.

Sometimes you will need instead to copy the contents of an existing object onto those of another. This is the **copying** operation.

A variant of copying is **cloning**, which produces a fresh object by duplicating an existing one.

For both copying and cloning, the default variants are “shallow”, affecting only one object, but **deep** versions are available to duplicate an object structure recursively.

A closely related problem is that of *comparing* two objects for shallow or deep equality.

The copying, cloning and comparison operations rely on only one language construct (the object equality operator `~`) and are entirely defined through language constructs but through routines that developer-defined classes inherit from the universal class [ANY](#). This makes it possible, through feature redefinitions, to adapt the semantics of copying, cloning and comparing objects to the specific properties of any class.

← “[ANY](#)”, 6.5, page 172; see also chapter 35 for more details.

21.2 COPYING AN OBJECT

--- MOVE AND REWRITE The first operation copies the fields of an object onto those of another. It is provided by the procedure [copy](#) from class [ANY](#). Descendant classes may redefine [copy](#) to provide a form of copy specific to object of the corresponding types, but the original version is always available through the frozen variant [identical_copy](#).

21.3 EQUALITY EXPRESSIONS

--- MOVED FROM EXPRESSION CHAPTER, NOT UPDATED ---

Object comparison features from *ANY*

The features whose contract views appear below are provided by class *ANY*.

default_is_equal (*other*: **like** *Current*)

- Is *other* attached to object field-by-field equal
- to current object?

ensure

same_type: **Result implies** *same_type* (*other*)

symmetric: **Result =** *other.default_is_equal* (**Current**)

consistent: **Result implies** *is_equal* (*other*)

is_equal (*other*: **? like** *Current*)

- Is *other* attached to object considered equal
- to current object?

ensure

same_type: **Result implies** *same_type* (*other*)

symmetric: **Result =** *other.is_equal* (**Current**)

consistent: *default_is_equal* (*other*) **implies Result**

The original version of *is_equal* in *ANY* has the same effect as *default_is_equal*.

These are the two basic object comparison operations. The difference is that *default_is_equal* is frozen, always returning the value of field-by-field identity comparison (for non-void *other*); any class may, on the other hand, redefine *is_equal*, in accordance with the pre- and postcondition, to reflect a more specific notion of equality.

Both functions take an argument of an attached type, so there is no need to consider void values.



An **Equality** expression serves to test equality of values with the symbol =, or their inequality with the symbol /=. Typical examples are

```
border_color = Black_color
window.height /= 0
```

The syntax is straightforward:



Equality expressions

Equality \triangleq Expression Comparison Expression

Comparison \triangleq "=" | "/=" | "~" | "/~"



The operators =, /= and ~ have the same precedence as relational operators such as < and >=, higher than the boolean operators such as **and** and **or**, and lower than arithmetic operators such as + and *.

→ [“Precedence and Parenthesized Form”](#), page 767



There is no constraint on equality expressions. In particular it is not necessary that either of the operands conform to the other. If they don't (or if one is void and the other attached to an object) the result will be false, but that doesn't make the expression illegal: whatever the answer, it's permitted to ask the question.

Equality Expression Semantics

The Boolean_expression $e \sim f$ has value true if and only if the values of e and f are both attached and such that $e.is_equal(f)$ holds.

The Boolean_expression $e = f$ has value true if and only if the values of e and f are one of:

- 1 • Both void.
- 2 • Both attached to the same object with reference semantics.
- 3 • Both attached to objects with copy semantics, and such that $e \sim f$ holds.

The form with ~ always denotes object equality. The form with = denotes reference equality if applicable, otherwise object equality. Both rely, for object equality, on function *is_equal* — the version that can be redefined locally in any class to account for a programmer-defined notion of object equality adapted to the specific semantics of the class.



The semantics of the equality operators and ~ was explored in detail as part of the discussion on reattachment. As a reminder, $e \sim f$ is true if and only if e and f are attached to equal objects, according to the *is_equal* function from class *ANY*; as to $e = f$:

See [22.16, page 618](#).

- 1 • If both e and f are of reference types, the expression denotes reference equality, true if and only if e and f are either both void or attached to the same object.
- 2 • If either e or f is of an expanded type, the expression denotes object equality; it returns the same result as .

If you need a different notion of equality you will, instead of $e = f$, use *equal* (e, f) which takes into account possible redefinitions of *equal*.

Inequality is defined in terms of equality:

SEMANTICS

Inequality Expression Semantics

The expression $e \neq f$ has value true if and only if $e = f$ has value false.

The expression $e \sim f$ has value true if and only if $e \sim f$ has value false.

Copying and cloning features from *ANY*

The features whose contract views appear below are provided by class *ANY* as secret features.

copy (*other*: ? **like** *Current*)

- Update current object using fields of object
- attached to *other*, to yield equal objects.

require

exists: *other* /= *Void*

same_type: *other*.same_type (*Current*)

ensure

equal: *is_equal* (*other*)

frozen *default_copy* (*other*: ? **like** *Current*)

- Update current object using fields of object
- attached to *other*, to yield identical objects.

require

exists: *other* /= *Void*

same_type: *other*.same_type (*Current*)

ensure

equal: *default_is_equal* (*other*)

frozen *cloned*: **like** *Current*

- New object equal to current object
- (relies on *copy*)

ensure

equal: *is_equal* (*Result*)

frozen *default_cloned*: **like** *Current*

- New object equal to current object
- (relies on *default_copy*)

ensure

equal: *default_is_equal* (*Result*)

The original versions of *copy* and *cloned* in *ANY* have the same effect as *default_copy* and *default_cloned* respectively.

Procedure *copy* is called in the form *x.copy* (*y*) and overrides the fields of the object attached to *x*. Function *cloned* is called as *x.cloned* and returns a new object, a “clone” of the object attached to *x*. These features can be adapted to a specific notion of copying adapted to any class, as long as they produce a result equal to the source, in the sense of the — also redefinable — function *is_equal*. You only have to redefine *copy*, since *cloned* itself is frozen, with the guarantee that it will follow any redefined version of *copy*; the semantics of *cloned* is to create a new object and apply *copy* to it.

In contrast, *default_copy* and *default_cloned*, which produce field-by-field identical copies of an object, are frozen and hence always yield the original semantics as defined in *ANY*.

All these features are **secret in their original class ANY**. The reason is that exporting copying and cloning may violate the intended semantics of a class, and concretely its invariant. For example the correctness of a class may rely on an invariant property such as

some_circumstance **implies** (*some_attribute* = *Current*)

stating that under *some_circumstance* (a boolean property) the field corresponding to *some_attribute* is cyclic (refers to the current object itself). Copying or cloning an object will usually not preserve such a property. The class should then definitely not export *default_copy* and *default_cloned*, and should not export *copy* and *cloned* unless it redefines *copy* in accordance with this invariant; such redefinition may not be possible or desirable. Because these features are secret by default, software authors must decide, class by class, whether to re-export them.

Deep equality, copying and cloning

The feature *is_deep_equal* of class *ANY* makes it possible to compare object structures recursively; the features *deep_copy* and *deep_cloned* duplicate an object structure recursively.

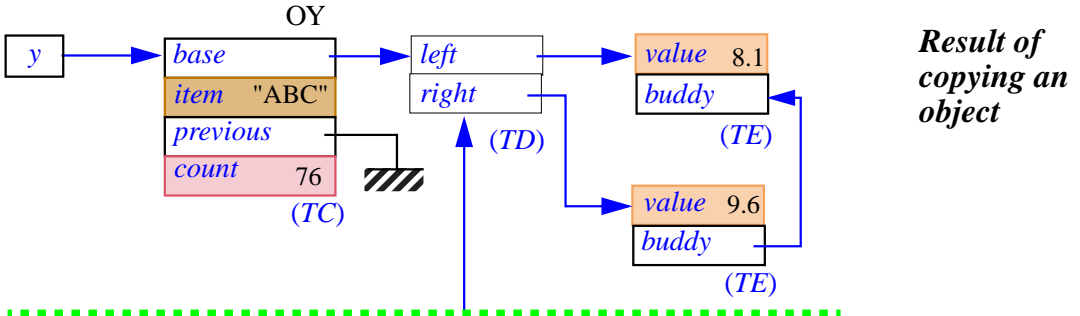
The default versions of the earlier features — *default_is_equal*, *default_copy*, *default_cloned* and the original versions of their non-*default* variants — are “shallow”: they compare or copy only one source object. The “deep” versions recursively compare or copy entire object structures.

Detailed descriptions of the “deep” features appear in the specification of ELKS.

Effect of a copy operation

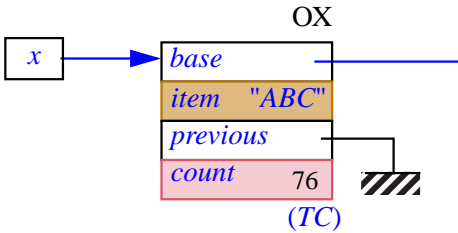
For the copy operation to succeed, both the source and the target must be attached to objects. (Cloning, however, will work for void sources or targets.)

Pre-existing structure



Result of copying an object

Effect on target object of copy operation $x.copy(y)$



The figure illustrates the effect of a copy operation with x as target and y as source. If `copy` has not been redefined for the generating class of the object OX attached to x , you may obtain this effect through the call

```
x.copy(y)
```

Before the call, y was attached to the object labeled OY ; x was attached to the object labeled OX . What the fields of OX contained then does not matter (since the call overwrites them), but this object must exist. The call copies every field of OY onto the corresponding field of OX .

Since the argument of `copy` is declared of type *like Current*, the type of OY conforms to the type of OX , but actually a precondition of `copy` requires more: the types of the two objects must be identical, so that their fields will be in one-to-one correspondence. On the figure, the type of OX and OY is called **TC**.

The fields of OY include expanded values, such as the integer *count*, of value 76, and references such as *base* and *previous*. In both cases, the copy operation will simply copy the field. For reference fields, no attempt is made to duplicate the data structure recursively: as a result, the *base* fields of both OX and OY will, after the call, be attached to the same object of type *TD*. Applying *copy* to any object containing reference fields will, indeed, always cause sharing of references; later in this chapter we will encounter recursive copy routines, *deep_copy* and *deep_clone*, which duplicate an entire object structure, following references recursively.

→ See [21.5, page 579](#) below, about *deep copy* and *clone*.

--- FIX --- As noted, *copy* requires a non-void source and target. For the target, this is simply part of the general requirement on **Call** instructions: in the above example, *x*, like the target of any other call, must be non-void under penalty of raising an exception. For the source, the requirement is expressed by the precondition of *copy*. A void source will trigger an exception if the execution monitors preconditions.

Specification of default copy

We can now examine the exact specification of *copy*. First, the interface of the procedure's version in class *ANY*:

```
copy, frozen identical_copy (other: like Current)
    -- Copy fields of other onto corresponding fields
    -- of current object.
require
    other_not_void: other /= Void
    type_identity: same_type (other)
ensure
    equal: is_equal (other)
```

→ See next about the function *same_type* used in the precondition and [21.6, page 580](#) about the function *is_equal* used in the postcondition.

Setting the style for other duplication and comparison routines, *copy* has two versions: one redefinable, the other (whose name begins with *identical_*) frozen.

← Chapter 5 discussed frozen features.

The second precondition clause uses function *same_type* of *ANY*. For *x* and *y* attached to objects OX and OY, *x.same_type* (*y*) has value true if and only if the type of OX has exactly the same type as OY.

→ *same_type* is discussed in "[OBJECT PROPERTIES](#)", [35.4, page 929](#)



Here now is the precise effect of the standard version. Assume *copy* has not been redefined and consider a call *x.copy* (*y*).

- 1 • As with any call, the target *x* must be non-void (if it were void the call would cause an exception); the first precondition clause of *copy* states that *y* must also be non-void. Let OX and OY be the attached objects at the time of the call.
- 2 • The precondition *same_type* requires that OX and OY have the same type; let *T* be that type.
- 3 • If *T* is a basic type (*BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL* or *POINTER*), the effect of the call is to copy the value of OY onto OX.
- 4 • If OX and OY are special objects (sequences of values used to represent strings or arrays), it is the implementation's responsibility to ensure that whenever such a situation arises — as a result of copying other objects — the size of OX is at least as large as the size of OY. Then the call copies the value of OY onto OX.

Special objects are not directly accessible to software texts. See [19.2, page 506](#).

- 5 • In the remaining cases OX and OY are objects made of zero or more fields, and the second precondition clause, *other_same_type*, implies that the types of OX and OY are identical, so that for every field of OX there is a field of the same type in OY. Then the call copies onto every field of OX the corresponding field of OY.

*With repeated inheritance, an attribute of TX may yield two fields in OY. The *Select* sub-clause, [16.5, page 442](#), determines which one is the field "corresponding" to the relevant OX field.*

As a consequence of the precondition *other_same_type*, you cannot use a copy operation to perform a conversion; a call *your_real.copy* (*your_integer*) is incorrect.

→ "[CONVERSIONS](#)", [22.6, page 591](#).



Tuning copy semantics

Any class may redefine *copy* to provide a copying operation consistent with the notion of object equality that has been deemed appropriate for the class.



Copy and equality are indeed intricately connected: the postcondition of *copy*, given on the previous page, states that the copy must make the target object equal to the source in the sense of function *is_equal*, another feature of *ANY* covered in detail later in this chapter. Clearly, if you redefine either one of *copy* and *is_equal*, you must redefine the other as well, to maintain consistent semantics for copying and equality according to the postcondition redefinition rules.

← "[REDECLARATION AND ASSERTIONS](#)", [10.17, page 283](#).

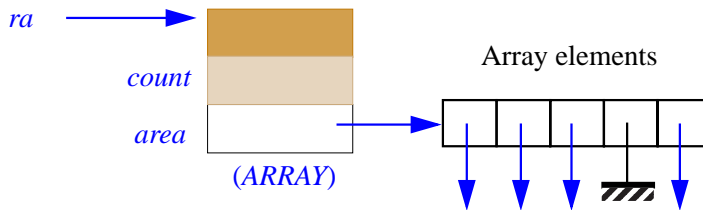


Redefinitions of *copy* and *is_equal* are of two kinds, going in reverse directions: one makes the semantics more “shallow” and the other makes it more “deep”:

- Sometimes you want to loosen the condition under which two instances of a class are considered equal, by ignoring some fields. Then *copy* can be redefined to copy only the relevant fields.
- You may instead want *copy* and equality to involve not only the original objects but also others to which they contain references.

In some cases you might want both: ignore some fields of the original objects, but involve some other objects as well.

Many classes of the Kernel Library and EiffelBase provide examples of the second kind, as they describe objects which are just headers for complex structures; *copy* and equality will then involve complete structures, not just the headers. For example the semantics of class *ARRAY* suggests an implementation as illustrated:



Array descriptor and array elements

(See chapter [36](#).)

The array object on the left is a header containing some general information such as the number of elements, *count*, and a reference *area* to the special object containing the array elements (which are references on the figure, but could be expanded values, for example of basic types). The default *copy* would only copy the *ARRAY* object; the procedure *copy* as redefined in class *ARRAY* also duplicates the special object containing the array elements. The same scheme applies to class *STRING*; the version of *copy* in list classes such as *LINKED_LIST* and *ARRAYED_LIST* copies not only the list headers but the list cells themselves.

Arrays, strings and the supporting Kernel Library classes are covered in chapter [36](#) and [10](#).



The *copy* algorithm stops there, however: it doesn't recursively duplicate the actual contents of the list. It's the same for arrays: in the above figure, *rb.copy (ra)* will copy the special object shown under “Array elements”, but not the objects to which its references are attached. For fully recursive duplication, you can use *deep_copy*, presented [later](#) in this chapter.

→ “[DEEP COPYING AND CLONING](#)”, 21.5, page 579.

Part of the reason for redefining *copy* is indeed that sometimes the default version — available as *identical_copy* — doesn't duplicate enough, while *deep_copy* duplicates too much. By redefining *copy* you can prescribe the exact depth you want — in accordance with your desired notion of equality, expressed by a redefinition of *is_equal* — for habitual *copy* operations.

21.4 CLONING AN OBJECT

Instead of copying an object, you can clone it; this creates a new object rather than updating the fields of an existing object. In class *ANY*, feature *clone* is a function, so a call of the form

```
clone (y)
```

is syntactically an expression; evaluating it will return a new object, which is a copy of the object attached to *y* if any. If *y* is void, the result is void.

Using cloning

The most obvious use of *clone* is in an assignment:

```
x := clone (y)
```

where the type of *y* must be a descendant of the type of *x*. The figure used to illustrate *x.copy (y)* also describes the effect of this assignment; only now the object OX represents a new object created by the assignment. ← Page 571.

Another use of *clone* is to pass a fresh copy of an existing object as argument to a call, as in

```
some_routine (... , clone (y), ...)
```

Although closely related, copy and clone differ in three respects:

- C1 • Copy modifies an existing object, whereas clone creates a new object. In the above assignment, any earlier attachment between *x* and some object is lost.
- C2 • For copy to work, the target must be non-void; this is expressed syntactically by the nature of *copy*, a procedure in *ANY*. In contrast, *clone* is a function and does not by itself have a target; it simply produces a result. When used as part of an assignment of target *x* as above, it does not care whether *x* is void or attached.
- C3 • Finally, because *clone* does not presuppose an existing target object, it can handle a void source. The result in this case is simply a void reference.

Like *copy*, *clone* does not attempt to follow references for fields of reference types, but simply copies the fields; a "deep" version is available. → "*DEEP COPYING AND CLONING*", 21.5, page 579.

As with a *Creation* instruction, a call to *clone* will fail, triggering an exception (the same one, of type *NO_MORE_MEMORY*) if it attempts to create a new object and no memory is available for it. ← "*CREATION SEMANTICS*", 20.12, page 556.

Twin

The description of *clone* indicates (property C3 above) that *Void* is a valid argument, for which the function will return *Void* as its result. This is convenient in the vast majority of cases. If you do know that the source of the clone operation is not void, you may, instead of *clone (y)*, use

```
y.twin
-- Defined only if y /= Void; then has same value as clone (y)
```

The only advantage of *twin* over *clone* — apart from being a little more concise — is that its implementation doesn't need to test for *Void*, so it will normally be slightly faster. But you should make sure to reserve *twin* for cases in which the target is known for sure to be non-void, since a void target would cause a run-time exception. If the case may arise, use *clone*, which handles void references gracefully.

Whenever one of the routines of this chapter handles a certain type of value and it is possible to define a reasonable default response for cases in which that value is void, the routine follows the example of *clone* and treats that value as an argument, not as the target of calls.

For a non-void *y*, *clone (y)* and *y.twin*, both applicable, are guaranteed always to yield the same value, thanks to the rules seen next.

Specification of default cloning

Here are the interfaces of function *clone* and its *twin* variant:

```
frozen clone (other: ANY): like other
--Void if other void; otherwise, new object equal to
-- object attached to other.
ensure
equal: equal (Result, other)
preserves_void: (other = Void) implies (Result = Void)
same_as_twin: (other /= Void) implies
equal (Result, other.twin)
```

The function 'equal' used in the postcondition is derived from 'is_equal'. See below.

```
frozen twin: like Current
-- New object equal to current object
ensure
not_void: Result /= Void
equal: Result.is_equal (Current)
same_as_clone: identical (Result, clone (Current))
```

Why are *clone* and *twin* frozen? The reason is not that their effect is immutable, but that you can change that effect without redefining the functions. To guarantee compatible semantics for cloning and copying, *clone* and *twin* are defined in terms of *copy*, and so will follow any redefinition of *copy*.

A frozen routine may, of course, call routines which are not frozen; it will then be affected by their redefinitions.

SEMANTICS

More precisely, here is the definition of the semantics of a call *clone* (*y*):

- 1 • If the value of *y* is void, the call returns a void value.
- 2 • If the value of *y* is attached to an object OY, the call returns a newly created object of the same type as OY, initialized by applying *copy* to that object with OY as source.

The second case also defines the semantics of *y.twin*. (For void *y* the general rules on routine call imply that the call will trigger an exception.)

In exactly the same way, function *equal*, used in the postcondition of *clone*, will automatically follow any redefinition of *is_equal*, used in the postcondition of *copy*. As we'll see in the discussion of equality, *equal* is to *is_equal* like *clone* to *twin*: it accepts a void target, but for non-void target returns the same result.

→ "OBJECT EQUALITY", 21.6, page 580.

To guarantee the original semantics of field-by-field duplication and ignore any redefinition of *copy*, you may use function *identical_clone*, which has the same signature as *clone*, and is defined in terms of *identical_copy* exactly as *clone* is defined from *copy*.



In principle, *clone* is superfluous: you could in most cases use a **Creation** and *copy* instead, replacing

```
create y ...
y.copy(x)
```

In practice, however, several reasons justify a separate *clone* facility:

- It's more concise to use *clone* than a creation followed by a copy, particularly in an expression, or in an argument to a routine call *r* (... , *clone* (*x*), ...) where the other form would be much more verbose, requiring the declaration of a local variable *y* and two extra instructions.
- If the associated class has two or more creation procedures, a **Creation** instruction forces you to choose one, although the choice is irrelevant.
- The creation procedure may do some extra work, justified when you create an object from scratch, but unneeded or harmful when all you need is a duplicate of an existing object.
- A **Creation** forces the client to specify the exact type of the new object, whereas a call to *clone*, as emphasized next, may dynamically produce an object of one among several possible types, depending on the type of the source, selected at run time. This is especially interesting for **Formal_generic_name** types, since *clone* may be applicable even when plain creation isn't.

← "CREATING INSTANCES OF FORMAL GENERICS", 20.9, page 543.

Cloning, types and factories

If x is an expression of type T , and its value is not void, the generating type of the object created by a call to `clone(x)` is not necessarily T : it is the type U of the object to which x is attached. U will always (ignoring the conversion case) conform to T , but may be based on a proper descendant. In fact T might be deferred, in which case there are no objects of generating type T .

The generating type of an object is the type of which it is a direct instance. See [19.2, page 506](#).



Assume `fig1` and `fig2` declared of the deferred type `FIGURE`, with `fig1` attached, at some point during execution, to an instance of an effective descendant of `FIGURE`, such as `CIRCLE`. Executing

```
fig2 := clone (fig1)
```

will attach `fig2` to another `CIRCLE`.

In such cases you don't need to know the exact dynamic type of the source (here `fig1`) when writing the instruction; because of polymorphism, that type may be different for successive executions of the same instruction.

An [earlier discussion](#) introduced an important application of these properties: how to implement a **factory of objects** through the **clonable array technique**. The idea was simply to obtain a fresh instance of a type, selected from a set of variants by a certain `code`, by writing

← “[Single choice and factory objects](#)”, [page 537](#); final, simplified form on [page 538](#).

```
x := clone (factory @ code)
```

where the `factory` is an array automatically created on first use — thanks to the beauties of once functions and creation expressions — through a simple function, worth showing again:



```
factory: ARRAY [FIGURE]
once
  Result.make (Low_id, High_id)

  -- Create and enter an instance of each desired kind:
  Result.put (create {SEGMENT} .make (...), Segment_id)
  Result.put (create {TRIANGLE} .make (...), Triangle_id)
  ... Similarly for each variant ...

end
```

← First shown on [page 538](#). The example involves a set of figure types.

This provide a simple and easily extendible scheme, compatible with the Single Choice principle and much preferable to the first form shown, which used explicit discrimination through a `Multi_branch`.

← “[Single choice and factory objects](#)”, [page 537](#); original form [\[1\]](#), [page 535](#).

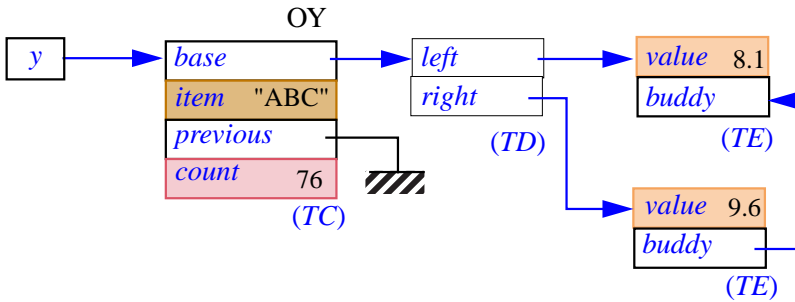
21.5 DEEP COPYING AND CLONING

The default *clone* and *copy* are, as noted, *shallow*: they do not follow references, just copy fields of the source object as they appear.

You may in some cases need deep versions of these operations, which will recursively duplicate an entire structure. The routines *deep_clone* and *deep_copy* of class *ANY*, with the same signatures as *clone* and *copy* respectively, fulfill this need. They will replicate an entire data structure, creating as many new objects as needed.

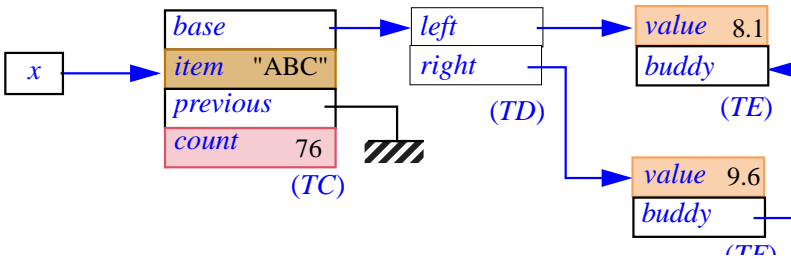
If we take as an illustration the original example used to present shallow copying, here is the result of a deep clone on the same structure: ← Page 571.

Pre-existing structure



Result of deep cloning

Structure created by a deep-clone operation $x := \text{deep_clone}(y)$



Your usual supplier of memory upgrades and discount disks will be happy to provide your staff, at no charge, with a full training session on the use of *deep_copy* and *deep_clone*.

Unlike their shallow counterparts *clone* and *copy*, *deep_clone* and *deep_copy* cannot cause sharing of references between source and target.

The deep versions are frozen. Their postconditions involve *deep_equal*, → "*DEEP EQUALITY*", 21.7, page 582. studied below.

21.6 OBJECT EQUALITY

---- OBSOLETE SECTION, REWRITE WITH ~ ----The discussion of cloning and copying transposes readily to the problem of comparing objects for equality. To determine if the objects attached to x and y are equal, you may use the expression

$equal(x, y)$



Here is the result of applying the default *equal* to two values x and y .

For convenience, the short form of the function `equal` appears after this semantic specification.

- 1 • If any one of x and y is void, the result is true if the other is also void, and false otherwise. Cases [2](#) to [5](#) assume that both arguments are attached to respective objects OX and OY.
- 2 • If the types of OX and OY are not identical, the result is false. For cases [3](#) to [5](#) let T be their common type.
- 3 • If T is a basic type, the result is true if and only if OX and OY are the same value ← “[Basic types](#)”, [page 338](#)
- 4 • If OX and OY are special objects (sequences of values used to represent strings or arrays), the result is true if and only if the sequences have the same length, and every field in one is identical to the field at the same position in the other. *Special objects are not directly accessible to software texts. See [19.2](#), [page 506](#).*
- 5 • Otherwise OX and OY are standard complex objects, and conformance of TY to TX implies that for every field of OX there is a corresponding field in OY. Then the result is true if and only if every reference field of OX is attached to the same object as the corresponding field in OY, and every subobject field of OX is (recursively) equal to the corresponding field in OY.

This definition of *equal*'s semantics closely parallels the semantic definition of *copy*; the five cases in both specifications match each other. The two are indeed designed to be compatible since, as noted, a call of the form $x.copy(y)$ must ensure the postcondition $equal(x, y)$.

Like copying, equality does not take conversions into account. The expression $equal(0.0, 0)$ — with a real argument and an integer argument — will return false. To get different behavior you must take care of the conversion yourselves.



The rejection of any conversions is part of a more general decision reflected in clause [2](#) above: equality may only hold for objects of the exact same type. You may be interested to know that the policy was more lax in early versions of Eiffel (as reflected in the first edition of this book): it specified the value true for *equal* (*x*, *y*) if the type of *y* conforms to the type of *x* and two objects have equal fields for the attributes of *x*'s type, even though OY may have more fields. This policy was more flexible, and did not cause any major problems; it went well, in particular, with the use of *is_equal* as the basic equality operation, explained next. It was abandoned, however, when critics pointed out that it made *equal* a non-symmetric property — it could result in *equal* (*x*, *y*) being true while *equal* (*y*, *x*) is not — whereas equality, in mathematics, is always symmetric. Hence the change to a more restrictive view of equality.

The short form of *equal* has not yet been given because in its postcondition it mentions the next function of interest, *is_equal*. Here it is:

```
frozen equal (some: ANY; other: like some): BOOLEAN
  -- Are some and other either both void or attached
  -- to equal objects?
ensure
  definition: Result = (some = Void and other = Void) or
    (some /= Void and other /= Void and then
      some.is_equal (other))
  symmetric: Result = equal (some, other)
```

Function *is_equal*, for its part, has a frozen synonym *is_identical*, and the interface form

```
is_equal (other: like Current): BOOLEAN
  -- Is other attached to an object equal to current object?
ensure
  only_if_not_void: Result implies other /= Void
  same_type: Result implies same_type (other)
  symmetric: Result = other.is_equal (Current)
  consistent: is_identical (other) implies Result
```

To change the semantics of equality in a particular class, just redefine *is_equal*; you cannot directly redefine *equal* — as you can see above, it's frozen — but its postcondition guarantees that *equal* will follow automatically. An obvious way to implement *equal* is indeed to rely on *is_equal*:

```
if some = Void then
  Result := (other = Void)
else
  Result := some.is_equal (other)
end
```

Function *is_equal* has the same relationship to *equal* as *twin* to *clone*: it works on a target and an argument as in *x.is_equal(y)*, where *equal* uses two arguments as in *equal(x, y)*. For non-void *x*, the two will always yield the same result, as defined above, but only *equal* accepts a void *x*; *is_equal* requires a non-void target. So it is the more basic of the two, but *equal* is more general.

Use *equal(x, y)* when there is any chance that *x* could be void. Otherwise you can still use *equal* except if you are concerned about the small overhead of testing for *Void*. Function *is_equal* is the one to redefine to introduce a specific semantics of equality for instances of a certain class. As noted, this almost always implies an associated redefinition of *copy*.

Earlier on, we encountered library classes — *ARRAY*, *STRING*, list implementations — that redefine *copy* to duplicate not just the header of an object structure but some of its contents too. These same classes redefine *is_equal* to compare the contents and not just the header.

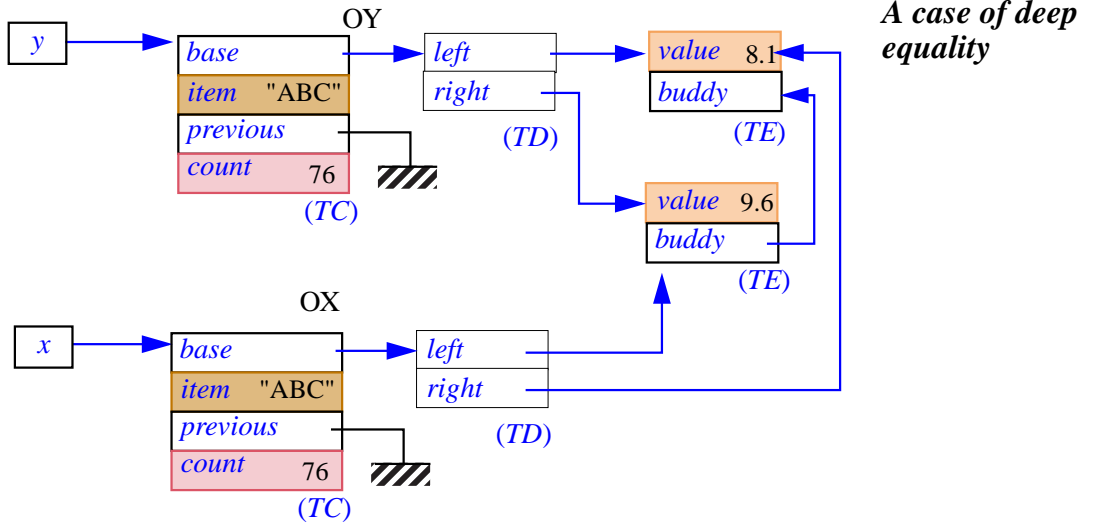
Like *copy* and *clone*, functions *equal* and *is_equal* have frozen synonyms: *identical* and *is_identical*, both guaranteeing the original semantics of exact field-by-field comparison.

21.7 DEEP EQUALITY

Like the shallow forms of copy and clone, the just explored shallow form of equality testing has a deep counterpart in *ANY*:

frozen *deep_equal* (some: *ANY*; other: **like some**): *BOOLEAN*
 -- Are *some* and *other* either both void
 -- or attached to isomorphic object structures?

What exactly are “isomorphic structures”? Clearly, *deep_equal* should yield true if one of the arguments results from a *deep_clone* or *deep_copy* applied to the other, as *x* and *y* on the figure that illustrated *deep_clone*. But we shouldn’t limit ourselves to this case, because it excludes any sharing between the two object structures, as in the following figure below, where we are entitled to expect that *deep_equal(x, y)* will yield true.



Here is the definition of deep equality (yielding true for such cases). It is convenient to define the notion separately for references and for objects.

Two references x and y are deep-equal if and only if they are either both void or attached to deep-equal objects.

Two objects OX and OY are deep-equal and only if they satisfy the following four conditions:

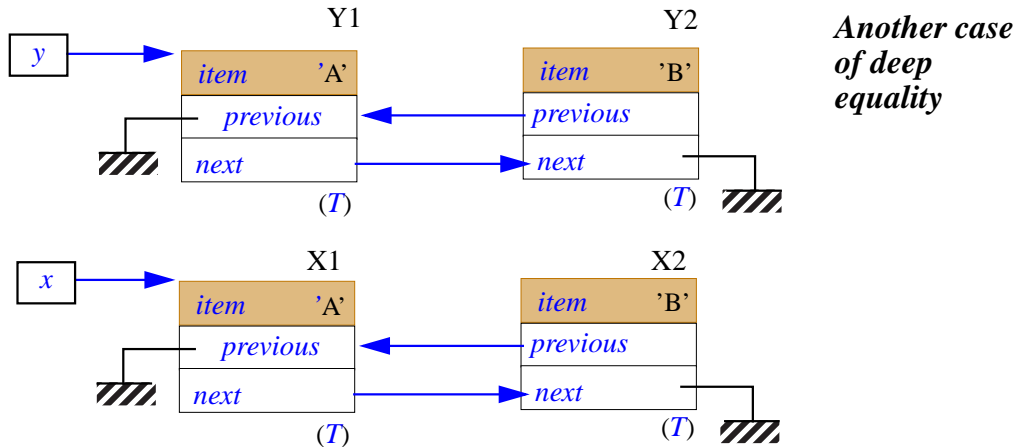
- 1 • OX and OY have the same exact type.
- 2 • The objects obtained by setting all the reference fields of OX and OY (if any) to void references are equal.
- 3 • For every void reference field of OX, the corresponding field of OY is void.
- 4 • For every non-void reference field of OX, attached to an object PX, the corresponding field of OY is attached to an object PY, and it is possible (recursively) to show, under the assumption that OX is deep-equal to OY, that PX is deep-equal to PY.



Condition **1** is the same as for *equal*: we want the types to be identical.

Conditions **2** and **3** express that every expanded or void field must be equal to the field in the other object.

Condition **4** handles the non-void reference fields. It is a bit subtle, as often when recursion is involved. The phrasing seems strange: why not just state that in this case PX must recursively be deep-equal to PY ?



The problem is that such a condition, although not wrong, would be impossible to prove, or disprove, for any cyclic data structures. Consider the situation picture above, which might be the result of a *deep_clone* operation. How can we check that the objects labeled X1 and Y1 are deep-equal — which they clearly should be?

Condition **1** will raise no problem since all objects are of the same type T . Condition **2** is readily satisfied since the only non-reference fields in X1 and Y1, the *item* fields, are equal. Condition 3 is also immediate since both *previous* fields are void. For condition 4, we must check recursively that the objects X2 and Y2 are deep-equal.

Conditions **2** and **3** again hold trivially, covering fields *item* and *next*. There remains to check condition **4**, in other words, that the *previous* fields of X2 and Y2 are attached to deep-equal objects. But now you see the problem: those attached objects are none other than X1 and Y1, and we are back to square one.

The phrasing of condition **4** gets us out of this potentially endless reasoning loop: when checking condition **4** on the original objects $X1$ and $Y1$, we only have to check that $X2$ and $Y2$ are deep-equal **under the assumption** that $X1$ and $Y1$ are themselves deep-equal. So here the equality of the *item* and *next* fields suffices to terminate the proof.



If you are looking at this with a programmer's rather than a mathematician's eyes, you will have understood this clause as meaning that in an abstract traversal algorithm designed to check deep-equality of objects, you may *mark* every previously encountered object so as not to explore it again, avoiding infinite looping.

If, on the other hand, you also master the theoretical background, you will have recognized the idea of self-conditional recursive proof: a technique whereby, to prove a property R , you must first prove a property of the form “if R holds, then P holds” for some other property P . This is exactly the scheme used, in *axiomatic* specifications of programming language semantics, to prove the correctness of a recursive routine.

On this theoretical perspective, see the book “Introduction to the Theory of Programming Language”, particularly its section : 9.10.6 and the example in 9.10.9.

Attaching values to entities

22.1 OVERVIEW

At any instant of a system’s execution, every entity of the system has a certain attachment status: it is either attached to a certain object, or void (attached to no object). Initially, all entities of reference types are void; one of the effects of a [Creation instruction](#) is to attach its target to an object. ← *Chapter 20.*

The attachment status of an entity may change one or more times during system execution through a **attachment** operations, in particular:

- The association of an actual argument of a routine to the corresponding formal argument at the time of a call.
- The [Assignment](#) instruction, which may attach an entity to a new object, or remove the attachment.

The validity and semantic properties of these two mechanisms are essentially the same; we study them jointly here.

----- REWRITE ---- You already know everything about the last case. This chapter explores the other three. It will also examine a closely related problem, for which the last chapter did the advance work: how to determine that two entities have the same attachment, or are **equal**, in any of the possible interpretations of this general notion.

22.2 ROLE OF REATTACHMENT OPERATIONS

Every reattachment operation has a **source** (an expression) and a **target** (a **Variable** entity). When the reattachment is valid, its effect will be ----

Reattachment, source, target

A **reattachment** operation is one of:

- 1 • An **Assignment** $x := y$; then y is the attachment's source and x its target.
- 2 • The run-time association, during the execution of a routine call, of an actual argument (the source) to the corresponding formal argument (the target).

We group assignment and argument passing into the same category, reattachment, because their validity and semantics are essentially the same:

- Validity in both cases is governed by the type system: the source must conform to the target's type, or at least convert to it. The Conversion principle guarantees that these two cases are exclusive.
- The semantics in both cases is to attach the target to the value of the source or a copy of that value.

← Chapter 14 presented both conformance and convertibility. See "Conversion principle", page 408.

This chapter explores reattachment operations: their constraints, semantics, and syntactic forms.

22.3 FORMS OF UNCONDITIONAL REATTACHMENT

As noted, the two forms of unconditional reattachment, **Assignment** instructions and actual-formal association, have similar constraints and essentially identical semantics, studied in the following sections.

The syntax is different, of course. An assignment appears as



$x := y$

where x , the target, is a **Variable** entity and y , the source, is an expression.

Very informally, the semantics of this instruction is to replace the value of x by the current value of y ; x will keep its new value until the next execution, if any, of a reattachment (unconditional, conditional, or new **Creation**) of which it is the target.

Actual-formal association arises as a byproduct of routine calls. A **Call** to a non-external routine r with one or more arguments induces an unconditional reattachment for each of the argument positions.

Consider any one of these positions, where the routine declaration (appearing in a class C) gives a formal argument x :

$r(\dots, x: T, \dots)$ **is** ...

For an external routine, written in another language, the exact semantics depends on the other language's rules.

Then consider a call to r , where the actual argument at the given position is y , again an expression. The call must be of one of the following two forms, known as unqualified and qualified:

See chapter 23 for the details of Call instructions and expressions.



$r(\dots, y, \dots)$

$t.r(\dots, y, \dots)$

-- In this second form, t must conform to a type based on C .

Qualified or not, the call causes an unconditional reattachment of target x and source y for the position shown, and similarly for all other positions.

A qualified **Call** also has a “target”, appearing to the left of the period, t in the second example. Do not confuse this with the target of the actual-formal attachment induced by the call, x in this discussion.

Informally again, the semantics of this unconditional reattachment is to set the value of x , for the whole duration of the routine's execution caused by this particular call, to the value of y at the time of call. No further reattachment may occur during that execution of the routine. Any new call executed later will start by setting the value of x to the value of the new actual argument.

22.4 SYNTAX AND VALIDITY OF ASSIGNMENT

Here is the syntax of an **Assignment** instruction:



Assignments

Assignment \triangleq Variable **":="** Expression

Actual-formal association does not have a syntax of its own; it is part of the **Call** construct.

→ See chapter 23 about Call. Syntax page 626.

The syntax of **Assignment** requires the target to be a **Variable**. **Recall** that a **Variable** entity is either an attribute of the enclosing class or a local variable of the immediately enclosing routine or agent. The latter case includes, in a function, the predefined entity **Result**. A formal routine argument is *not* a **Variable**; this property is discussed further in the next section.

← 19.8 introduced Variable entities, with syntax on page 512 and the associated Variable rule on page 514.

The principal validity constraint in both cases is that the source must conform or convert to the target. For **Assignment** this is covered by the following rule:



Assignment rule

VBAR

An **Assignment** is valid if and only if its source expression is compatible with its target entity.

To be “compatible” means to conform or convert.

← “Compatibility between types”, page 384.



This also applies to actual-formal association: the actual argument in a call must conform or convert to the formal argument. The applicable rule is **argument validity**, part of the general discussion of call validity.

The two cases, conformance and convertibility, are complementary:

→ “THE CALL VALIDITY RULE”, 25.10, page 681.



- **Conformance** is the more common situation. As you will remember, type *U* conforms to type *T* — and, as a consequence, an expression of the first type to an entity of the second one — if the base class of *U* is a descendant of the base class of *T* and, if generic parameters are present, they also conform; the conformance chapter gave the details.

← Chapter 14.

- **Convertibility** allows reattachments that also perform a conversion, as when you are assigning an integer value to a real target.

← Chapter 15.

22.5 THE STATUS OF FORMAL ROUTINE ARGUMENTS



The syntax of **Assignment** requires the target to be a **Variable**. This includes, as noted, attributes and local variables, but not formal arguments of the enclosing routine. So in the body of a routine



```
r (x : SOME_TYPE)
```

```
  ...
  do
    ...
  end
```

an assignment $x := y$, for some expression *y*, would not be valid. The only reattachments to a formal argument occur at call time, through the actual-formal association mechanism.



It is indeed a general rule of Eiffel that routines may not change the values of their arguments. A routine is an operation to be performed on certain operands; arguments enable callers to specify what these operands should be in a particular application of the operation. Letting the operation change the operands would be confusing and error-prone.

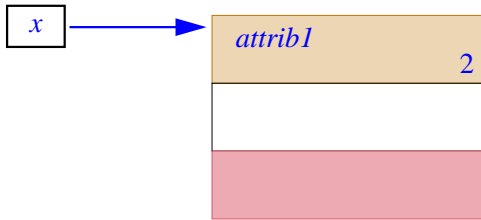
Although some programming languages offer “out” and “in-out” modes for arguments, they are a notorious source of trouble for programmers, and complicate the language; for example:

- You must have special rules for the corresponding actual arguments (they must be variable).
- You must prohibit using the same actual argument twice, as in $r(e, e)$, but only if both of the affected argument positions are “out” or “in out”.

The Eiffel rule does not prohibit a routine r from modifying the **objects** that it is passed: if a formal argument x is a non-void reference, r has access to the attached object and can perform any valid feature call on it. In the situation pictured below the body of r may include a procedure call

```
x.set_attrib1 (2)
```

where *set_attrib1* will update the value of the integer field *attrib1*. What is **not** permitted is an **Assignment** of target x , which would affect the reference rather than the object.



Object may change, reference not

22.6 CONVERSIONS



All that beginning Eiffel programmers really need to know about convertibility is that commonly accepted mixed-type arithmetic assignments with no loss of information, such as *your_real := your_integer* (but not the other way around, which requires using a truncation or rounding function) are OK and will cause the proper conversions. So on first reading you should skip this section.

Skip to “SEMANTICS OF REATTACHMENT”, 22.7, page 593.

Conformance and convertibility are, as noted, mutually exclusive cases. Let us start our study of reattachment semantics by the second one — even though conformance is by far the more common case — because the discussion of convertibility already told us most of what we need to know.

← Chapter 15. See also “The Target Conversion mechanism deserves some justification...”, page 770.

In that discussion we saw that it is possible for a class to declare, through its creation procedures, one or more **creation types**, as in:

```
class DATE create
  from_tuple convert { TUPLE [INTEGER, INTEGER, INTEGER]}
  ...
```



This is intended to permit attachments from any of the conversion types (here only one) to the current type, so that you may write

```
compute_revenue ([1, "January", 2000], [1, "January", 2001])
```

where `compute_revenue` expects two date arguments. Argument passing in this case will cause, prior to actual attachment, the creation of a new object of type `DATE` and its initialization through the given creation procedure `from_tuple`. As was noted in the earlier discussion, this means that the call is equivalent to

```
compute_revenue (create {DATE}.from_tuple ([1, "January", 2000]),
                 create {DATE}.from_tuple ([1, "January", 2001]))
```

Similarly, a call `your_date := [1, "January", 2000]` is equivalent to `create your_date.from_tuple ([1, "January", 2000])`.

It is also possible to specify conversion through a function in the source type, rather than a procedure in the target type. Between any two given types, at most one of these possibilities may apply. If it is possible to convert an expression `exp` to an entity `e`, we say that `exp` converts to `x`, through a conversion routine (procedure or function).

← “[EXPRESSION CONVERTIBILITY: THE ROLE OF PRE-CONDITIONS](#)”, 15.10, page 420.

This semantic specification and the supporting definition rely on the properties of the conversion mechanism, expressed by the [Conversion Procedure rule](#) and the associated [definitions](#) (convertible types of a class), which guarantee that everything is unambiguous:

← “[Conversion Procedure rule](#)”, page 411; “[Converting to and from a type](#)”, page 415;

- The definition of “convertible types” tells us that `SOURCE` must appear among the `Conversion_types` of a creation procedure of the base class of `TARGET`.
- Clause 4 of the Conversion Procedure rule, requiring all the convertible types of a class to be different, guarantees that there is only one such procedure, making the definition of “applicable conversion procedure” legitimate.
- Clauses 6 and 7 of the rule guarantee that this procedure has exactly one formal argument, of a type `ARG` to which `SOURCE` must conform or convert.

If `SOURCE` converts (rather than conforms) to `ARG`, then the attachment will, as was noted in the earlier discussion, cause two conversions rather than one, since to the conversion procedure must convert its argument to type `ARG`. As was also noted, things stop here: a conversion reattachment may cause one conversion (the usual case), or two (if the `SOURCE` type converts to the `ARG` type), but no more.

← See discussion of clause 6 of the Conversion Procedure rule on page 411.



This discussion completes the specification of reattachment in the convertibility case. Since the Conversion principle tells us that a type may not both convert and conform to another, we may limit our attention, for the rest of this chapter, to the more common case: reattachments in which the source of an assignment or argument passing *conforms* to the target.

← “*Conversion principle*”, page 408.

--- TEXT BELOW MAY HAVE TO BE TRANSFERRED ELSEWHERE

22.7 SEMANTICS OF REATTACHMENT

Let us examine the precise effect of executing an unconditional reattachment of either of the two forms, for a source conforming to the target.

Because that effect is the same in both cases — an **Assignment** $x := y$ and a call that uses y as actual argument for the formal argument x of a routine — we can use the first as our working example: the assignment

```
x := y
```

where x is of type TX and y of type TY , which must conform to TX .

The effect depends on the nature of TX and TY : reference or expanded? Here is the basic rule, covering the vast majority of practical cases:

- If both TX and TY are expanded, the assignment copies the value of the object attached to the source onto the object attached to the target.
- If both are reference types, the operation attaches x to the object attached to y , or makes it void if y is void.

As an example of the first case, in



```
x, y: INTEGER
...
y := 4
x := y
```

the resulting value of x will be 4, but the last **Assignment** does not introduce any long-lasting association between x and y ; this is because $INTEGER$ is an expanded type.

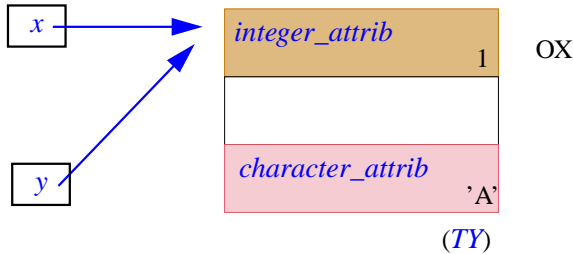
As an example of the second case, if TC is a reference type, then



```
x, y: TC
...
create y ...
x := y
```

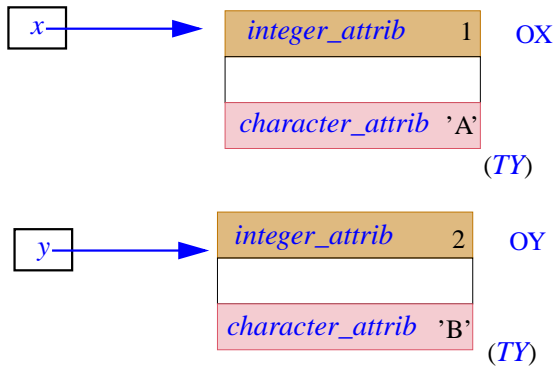
*Effect of
reference
reattachment*

will result in x and y becoming attached to the same object:



This rule addresses the needs of most applications. There remains, of course, to see what happens when one of *TX* and *TY* is expanded and the other reference. But it is more important first to understand the reasons for the rule by exploring what potential interpretations make sense in each case.

Consider first the case of references. We start from the run-time situation pictured below, with two objects labeled OX and OY, assumed for simplicity to be of the same type *TY*, and accessible through two references *x* and *y*. Of course, since the Eiffel dynamic model is fully based on objects, *x* and *y* themselves will often be reference fields of some other objects, or of the same object; these objects, however, are of no interest for the present discussion and so they will not appear explicitly.

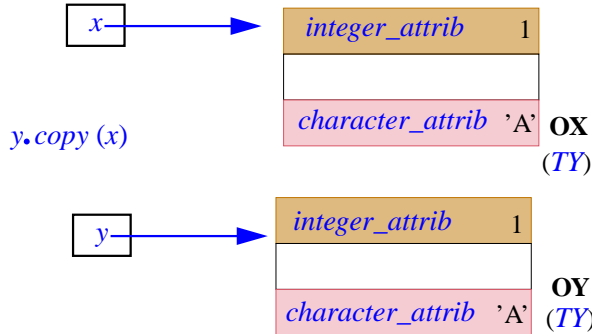


*Before a
reattachment*

Three possible kinds of operation may update *x* from *y*: copying, cloning and reference reattachment.

The first, copying, makes sense only if both x and y are attached (non-void). Its semantics, seen in the last chapter, is to copy every field of the source object onto the corresponding field of the target object. It does not create a new object, but only updates an existing one. We know how to achieve it: through procedure *copy* of the universal class *ANY* or, more precisely, its frozen version *identical_copy*, ensuring fixed semantics for all types (whereas *copy* may be redefined). The next figure illustrates the effect of a call $y.identical_copy(x)$ starting in the above situation.

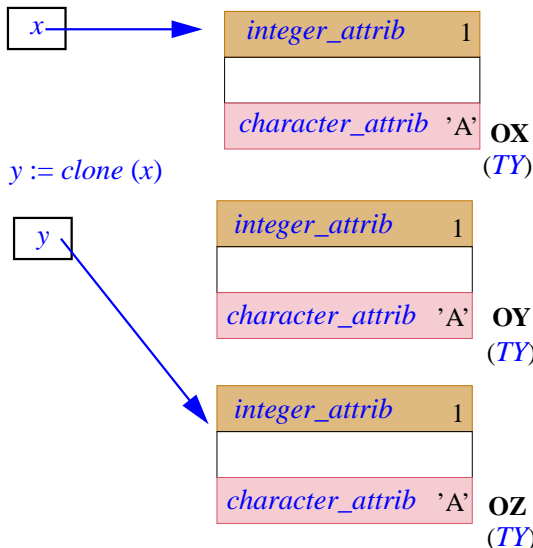
← See 21.2, page 565 on *copy* and its frozen version *identical_copy*.



Effect of standard copy

The second operation is a close variant of the first: cloning also has the semantics of field-by-field copy, but applied to a newly created object. No existing object is affected. Here too a general mechanism is available to achieve this: a call to function *clone* which (anticipating on this section) we have learned to use in an assignment $x := clone(y)$. To guard against redefinition we may use the frozen version *identical_clone*. The result is shown below; the cloning creates a new object, OZ, a carbon copy of OX.

← See 21.4, page 575, about *clone* and *identical_clone*.

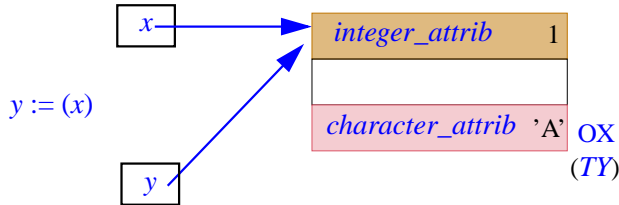


Effect of standard clone

Assuming y was previously attached to OY as a result of the preceding operation, it is natural to ask: “What happens to the object OY?”. This will be discussed in a [later section](#).

→ “[MEMORY MANAGEMENT](#)”, 22.15, page 616.

The third possible operation is reference reattachment. This does not affect any object, but simply reattaches the target reference to a different object. The result (already visible in the last figure) may be represented as follows:



**Effect of
reference
reattachment**



To devise the proper rule for semantics, we must study which of these operations make sense in every possible case. Since the source and target types may each be either expanded or reference, there will be four cases:

<i>SOURCE TYPE</i> →	Reference	Expanded
<i>TARGET TYPE</i> ↓		
Reference	[1] <ul style="list-style-type: none"> • Copy (if neither source nor target void) • Clone • Reference reattachment 	[2] <ul style="list-style-type: none"> • Copy (if target not void) • Clone
Expanded	[3] <ul style="list-style-type: none"> • Copy (will fail if source is void) 	[4] <ul style="list-style-type: none"> • Copy

Meaningful possibilities for the semantics of reference reattachment

This list only takes into account shallow operations. Deep variants were discussed in [21.5](#), page 579.

If all we were interested in was copying and cloning, we would not need any new mechanism: routines *identical_copy* and *identical_clone*, from *ANY*, are available for these purposes. The only operation we would miss is reference reattachment, corresponding to the last figure. This only makes sense for case **1**, when both target and source are of reference types: if the target is expanded, as in cases **3** and **4**, there is no reference to reattach; and if the source is expanded, as in cases **2** and **4**, a reattachment would introduce a **reference to a sub-object**, a case discussed and rejected in the discussion of the dynamic model.

← “*REFERENCE ATOMICITY*”, 19.7, page 510; the excluded case is illustrated by the figure on page 510.

In case **1**, however, we do need the ability to specify reference reattachment, not covered by *copy*, *clone* or their frozen variants. This will be the semantics of the *Assignment* $x := y$ and of the corresponding actual-formal association when both x and y are of reference types.

We now have notations for expressing meaningful operations in every possible case: reference assignment in case **1**, routines *identical_copy* and *identical_clone* in the other cases. At least two reasons, however, indicate that in addition to these case-specific operations we also need a single notation applicable to all four cases:

- In a generic class, TX and TY may be a *Formal generic name*; then the class text does not reveal whether x and y denote objects or references, since this depends on the actual parameter used in each generic derivation of the class. But it must be possible for this class text to include an *Assignment* $x := y$, or a call $r(\dots, y, \dots)$, with a clearly defined meaning in all possible cases.
- The availability of general-purpose copying and cloning mechanisms does not relieve us from the need to define a clear, universal semantics for actual-formal association.

← If the formal generic is TX , conformance requires TY to be identical to TX . If the formal is TY , TX is either TY or an ancestor of TY 's constraint (*ANY* if TY is unconstrained). See “*Direct conformance: formal generic*”, page 393.

Examination of the above table suggests a uniform notation addressing these requirements. What default semantics is most useful in each case?

- In case **1**, where both x and y denote references, the semantics should be reference reattachment, if only (as discussed above) because no other notation is available for that operation.
- In case **4**, with both x and y denoting objects, only one semantics makes sense for a reattachment operation: copying the fields of the source onto those of the target.
- In case **2**, with x denoting a reference and y an object, both copying and cloning are possible. But copying only works if x is not void (since there must be an object on which to copy the source's fields). If x is void, copying will fail, triggering an exception. It would be unpleasant to force class designers to test for void references before any such assignment. Cloning, much less likely to fail, is the preferable default semantics in this case.

Cloning may also fail, triggering an exception, if there is no more memory available (21.2). But this is a much less frequent situation than the target being a void reference.

- In case [3](#), as in case [1](#), the target x is an object, so copying is again the only possible operation. In this case it will fail if y is void (since there is no object to copy), but then no operation exists that would always work.

SEMANTICS

This analysis leads to the following definition of the semantics of unconditional reattachment in the case of a source conforming to its target.

← Remember that the **convertibility** case is distinct (“**CONVERSIONS**”, [22.6, page 591](#))

<i>SOURCE TYPE</i> →	Reference	Expanded
<i>TARGET TYPE</i> ↓		
Reference	[1] Reference reattachment	[2] Clone
Expanded	[3] Copy (Fails if source void)	[4] Copy

The semantics of conformance reattachment

NOT a semantic specification but only a list of available possibilities for such a specification. The actual semantics appears next.

In this semantic specification, “**Copy**” and “**Clone**” refer to the frozen features *identical_copy* and *identical_clone* that every class inherits from the universal class *ANY*.

→ The table giving equality semantics on [page 619](#) will be organized along similar lines.



Arguments could be found for using instead the redefinable version *copy*, and *clone* which is defined in terms of *copy*: after all, if the author of a class redefined these routines, there must have been a reason. But it is more prudent to stick to the frozen versions, so that the language defines a simple and uniform semantics for assignment and argument passing on entities of all types. If you do want to take advantage of redefinition, you can always use the call. *copy*.(y) instead of the assignment $x := y$, or pass *clone* (y) instead of y as an actual argument to a call. These alternatives to unconditional reattachment apply of course to reference types as well as expanded ones.

For the exception raised in case [3](#) if the value of y is void, the Kernel Library class *EXCEPTIONS* introduces the integer code *Void_assigned_to_expanded*.

See chapters [26](#) on exceptions and [37](#) on class *EXCEPTIONS*.

This semantic definition yields the most commonly needed effect in each case. This applies in particular to cases [1](#) and [4](#), which account for the vast majority of reattachments occurring in practice: for an integer variable (case [4](#)), it is pleasant to be able to write

```
n := 3
```

to produce the effect of

```
n.copy(3)
```

Here *copy* and *identical_copy* are the same.

but uses a commonly accepted notation and has the expected result. For a reference variable y , it is normal to expect the call

```
some_routine ( $y$ )
```

simply to pass to *some_routine* a reference to the object attached to y , if any, rather than to duplicate that object for the purposes of the call. If you do wish duplication – shallow or deep – to occur, you may make your exact intentions clear by using one of the calls

```
some_routine (clone ( $y$ ))
some_routine (identical_clone ( $y$ ))
some_routine (deep_clone( $y$ ))
```

An interesting application is the case of generic parameters and generically derived types. If the type of x and y is a formal generic parameter of the enclosing class, as in



```
class GENERIC_EXAMPLE [ $G$ ] feature
  example_routine
    local
       $x, y: G$ 
    do
       $x := y$ 
    end
end
```

the effect of the highlighted assignment may be reference reattachment or copying depending on the actual generic parameter used for G in the current generic derivation. (Cloning, which only occurs for reference target and expanded source, does not apply to this case since, by construction, x and y are of the same type.) We will shortly come back to the effect of reattachment semantics on generic programming.

→ [“EFFECT ON GENERIC PROGRAMMING”, 22.10, page 604.](#)

A consequence of the validity and semantics rules is the following semantic principle, which will be important to understand the run-time behavior of our systems:



Reattachment principle

After a reattachment to a target entity t of type TT , the object attached to t , if any, is of a type conforming to TT .

“If any” because the source of the attachment might have been void. If not, its value v is of a type VT that either conforms or converts to TT (but not both). If it conforms, the operation simply reattaches t to v , satisfying the principle. If it converts, the operation produces a new object of type TT ; this satisfies the principle too since TT conforms to itself.

Attaching an entity, attached entity

Attaching an entity e to an object O is the operation ensuring that the value of e becomes **attached to** O .

Although it may seem tautological at first, this definition simply relates the two terms “attach”, denoting an operation that can change an entity, and “attached to an object”, denoting the state of such an entity — as determined by such operations. These are key concepts of the language since:

- A reattachment operation (see next) may “*attach*” its target to a certain object as defined by the semantic rule; a creation operation creates an object and similarly “*attaches*” its creation target to that object.
- Evaluation of an entity, per the Entity Semantics rule, uses (partly directly, partly by depending on the Variable Semantics rule and through it on the definition of “value of a variable setting”) the object *attached* to that entity. This is only possible by ensuring, through other rules, that prior to any such attempt on a specific entity there will have been operations to “attach” the entity or make it void.

Reattachment Semantics

The effect of a reattachment of source expression *source* and target entity *target* is the effect of the first of the following steps whose condition applies:

- 1 • If *source* converts to *target*: perform a conversion attachment from *source* to *target*.
- 2 • If the value of *source* is a void reference: make *target*’s value void as well.
- 3 • If the value of *source* is attached to an object with copy semantics: create a clone of that object, if possible, and attach *target* to it.
- 4 • If the value of *source* is attached to an object with reference semantics: attach *target* to that object.



As with other semantic rules describing the “effect” of a sequence of steps, only that effect counts, not the exact means employed to achieve it. In particular, the creation of a clone in step [3](#) is — as also noted in the discussion of creation — often avoidable in practice if the target is expanded and already initialized, so that the instruction can reuse the memory of the previous object.

Case [1](#) indicates that a conversion, if applicable, overrides all other possibilities. In those other cases, if follows from the Assignment rule that \rightarrow . *source* must **conform** to *target*.

Case [2](#) is, from the validity rules, possible only if both *target* and *source* are declared of *detachable* types.

In case [3](#), a “clone” of an object is obtained by application of the \rightarrow . function *cloned* from *ANY*; expression conformance ensures that *cloned* is available (exported) to the type of *target*; otherwise, cloning could produce an inconsistent object.

The cloning might be impossible for lack of memory, in which case the semantics of the cloning operation specifies triggering an exception, of type *NO_MORE_MEMORY*. As usual with exceptions, the rest of case [3](#) does not then apply.

In case [4](#) we simply reattach a reference. Because of the validity rules (no reference type conforms to an expanded type), the target must indeed be of an reference type.

This rule defines the *effect* of a construct through a sequence of cases, looking for the first one that matches. As usual with semantic rules, this only specifies the result, but does not imply that the implementation must try all of them in order.

The semantics of assignment is just a special case of this rule:

Assignment Semantics

The effect of a reassignment $x := y$ is determined by the Reattachment Semantics rule, with source *y* and target *x*.

The other cases where Reattachment Semantics applies is actual-formal association, per step [5](#) of the General Call rule.

\rightarrow “*General Call Semantics*”, page 653.

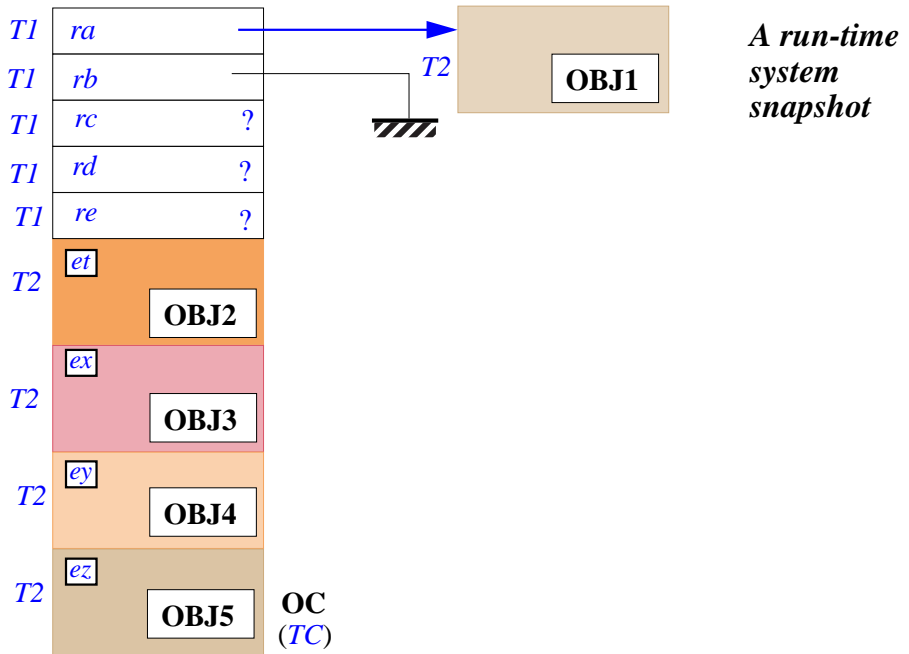
On the other hand, the semantics of *Object_test*, a construct which also allows a *Read_only* entity to denote the same value as an expression, is simple enough that it does not need to refer to reattachment.

22.8 AN EXAMPLE

---- WRONG (OLD SEMANTICS), TO BE REMOVED



To see the effect of reattachment in various cases, consider the run-time situation pictured below.



All the entities considered are attributes of a class C . OC, the complex object on the left, is a direct instance of type TC , of base class C . *OC is not only complex but composite.*

The first five attributes (ra , rb , rc , rd , re), whose names begin with r , are of a reference type $T1$. The corresponding fields of OC are references. The four others (et , ex , ey , ez), whose names begin with e , are expanded. The corresponding fields are sub-objects of OC, which have been given the names OBJ2 to OBJ5. The reference field ra is originally attached to another object OBJ1, also of type $T2$.

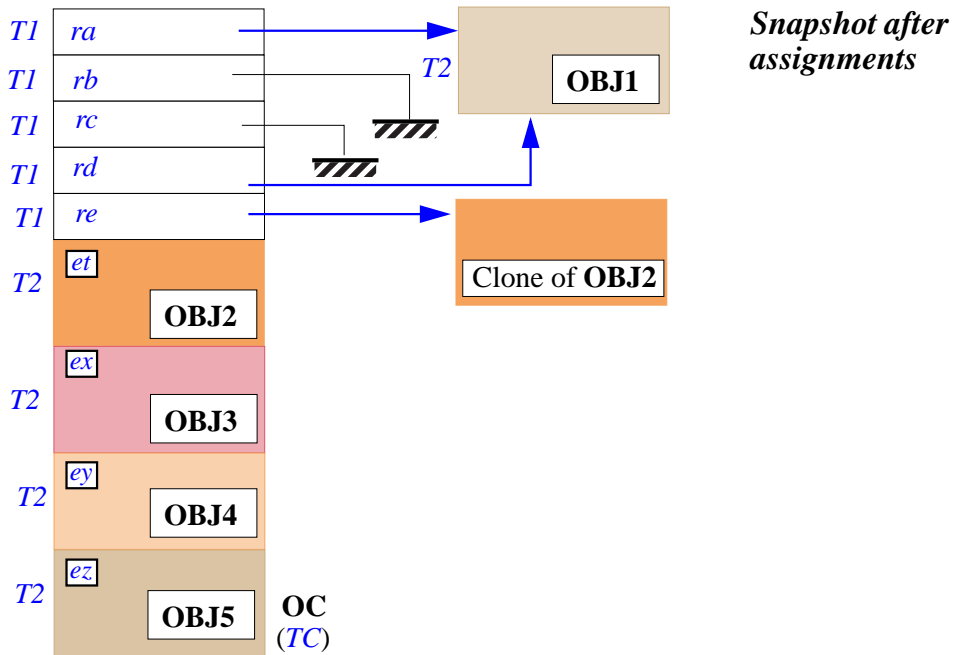
Assume that class C has the following routine, using **Assignment** instructions to perform a number of reattachments:

```

assignments is
  -- Change various fields.
  do
     $rc := rb$ 
     $rd := ra$ 
     $re := et$ 
     $ex := ey$ 
     $ez := ra$ 
  end

```


If applied to the above OC, this procedure will produce the following situation:



The assignment $re := et$, with reference target and expanded source, produces a duplicate of object OBJ2.

An attempt to execute $et := rb$, with an expanded target and a void source, would trigger an exception.

22.9 ABOUT REATTACHMENT



(This section brings no new Eiffel concept. It will only be of interest to readers who wish to relate the above concepts to the argument passing conventions of earlier programming languages.)

It may be useful to compare the semantics of unconditional reattachment to the mechanisms provided by other languages, in particular to traditional variants of argument passing semantics.

Consider a call of the form

$r(\dots, y, \dots)$

This causes an attachment as a result of actual-formal association between the expression y , of type TY , and the corresponding formal argument x , of type TX .

An examination of the semantics defined above in light of other argument passing conventions yields the following observations:

- If both *TX* and *TY* are reference types (case [1](#) of the [table of reattachment semantics](#)), the reattachment causes sharing of objects through references, also known as **aliasing**. For actual-formal association this achieves the effect of **call by reference**, with the target being protected against further reattachment for the duration of the call. ← Page [596](#).
- If both *TX* and *TY* are expanded types (case [4](#)), reattachment copies the content of *y*, an object, onto *x*. This achieves the effect of **call by value**.
- If *TX* is an expanded type and *TY* a reference type (case [3](#)), the operation copies onto *x* the content of the object attached to *y* (*y* must be non-void). This achieves what is often called **dereferencing**.
- If *TX* is a reference type and *TY* an expanded type (case [2](#)), the operation attaches to *x* a newly created copy of *y*. This case has no direct equivalent in traditional contexts; it may be viewed as a form of call by value combined with call by reference.

22.10 EFFECT ON GENERIC PROGRAMMING

The semantics of unconditional reattachment has a direct effect on both the production and the use of generic classes — a cornerstone of reusable software production.

For a generic class such as [GENERIC_EXAMPLE](#) above, it may seem surprising to see a given syntactical notation, the assignment symbol `:=`, denote different operations depending on the context, and similarly for argument passing. ← Page [599](#).

This convention corresponds, however, to the most common needs of generic programming. The container classes of EiffelBase, such as [LINKED_LIST](#), [TWO_WAY_LIST](#), [HASH_TABLE](#) and many others, used to store and retrieve values of various types, provide numerous examples. These classes are all generic and, depending on their generic derivations, the values they store may be references or objects. *The notion of container data structure was presented in [10.21, page 292](#), and [12.3, page 351](#).*

All of these classes have one or more procedures for adding an element to a data structure; for example, to insert an element to the left of the current cursor position in a linked list a client will execute

```
some_list.put_left (s)
```

Almost all of these procedures use assignment for fulfilling their task. Many do this not directly but through a call of the form

```
some_cell.put (x)
```

where *some_cell*, representing some individual entry of the data structure, is of a type based on some effective descendant of the deferred generic class *CELL*; for example, *LINKED_LIST* uses the descendant *LINKABLE*, describing cells of linked lists. Procedure *put* comes from *CELL*, where it appears (in effective form) as [



```
class CELL [G] feature
  item: G;
  put (new: G)
    -- Replace the cell value by new
  do
    item := new
  ensure
    item = new
  end
  ... Other features ...
```

This is a slight simplification; the type of the argument 'new' is actually like item, which has the same immediate effect since item is of type G.

Because the addition of an element *x* by *put* uses assignment, what will be added to the data structure is an object value if *x* is of expanded type, and otherwise a reference to an object.

This policy means that if you are a “generic programmer” (a developer or user of generic classes) you must exercise some care, when dealing with data structures having diverse possible generic derivations, to make sure you know what is involved in each case: objects or references to objects. But it provides the most commonly defaults: a call

```
some_list_of_integers.put_left (25)
```

inserts the value 25, whereas

```
some_list_of_integers.put_lift (her_bank_account)
```

does not duplicate the object representing the bank account. Storing a reference in this case is the most conservative default policy. As in earlier examples, you can always obtain a different policy by using such calls as

```
some_list_of_integers.put_left (clone (her_bank_account))
some_list_of_integers.put_left (deep_clone (her_bank_account))
```

which guarantee uniform semantics (duplication, shallow in the first case and deep in the second) across the spectrum of possible types.

The discussion also applies to the problem of **searching** a data structure, discussed below.

→ End of “*SEMANTICS OF EQUALITY*”.
22.16, page 618.

22.11 POLYMORPHISM

The only type constraint on unconditional reattachment is that (aside from the convertibility case) the type of the source must conform to the type of the target. ← “CONVERSIONS”, 22.6, page 591.

If the target is expanded, this means that the types must essentially be the same; the only permitted flexibility is that one may describe objects of a certain form and the other references to objects of exactly the same form. This follows directly from the rule defining conformance when an expanded type is involved. ← “General conformance”, page 388 and “Direct conformance: expanded types”, page 396.

If the target is a reference, however (cases 1 and 2 of the reattachment semantics table), the situation is more interesting. If the target’s base type is based on a class *C*, the validity rules mean that the base class of the source may be not just *C* but any proper descendant of *C*. This gives a remarkable flexibility to the type system, while preserving safety thanks to the conformance restrictions. ← Page 596.

As a consequence, an expression declared of type *TC* may at run time denote objects not just of type *TC* but of many other types, all based on descendants of the base class of *TC*.

So to study the run-time semantics of Eiffel systems we need to consider, along with the *type* of an expression (its type as deduced from declarations in the software text), its possible *dynamic types*:



Dynamic type

The **dynamic type** of an expression *x*, at some instant of execution, is the type of the object to which *x* is attached, or *NONE* if *x* is void.



This should not be confused with the **type** of *x* (called its **static type** if there is any ambiguity), which for an entity is the type with which it is declared, and for an expression is the type deduced from the types of its constituents. → “Type of an expression”, page 783.

An expression has, of course, only one (static) type. But, as a key property of Eiffel’s object-oriented style of computation, it may have more than one dynamic type. This is known as *polymorphism*.



Polymorphic expression; dynamic type and class sets

An expression that has two or more possible dynamic types is said to be **polymorphic**.

The set of possible dynamic types for an expression *x* is called the **dynamic type set** of *x*. The set of base classes of these types is called the **dynamic class set** of *x*.

Eiffel has a strongly typed form of polymorphism: the dynamic type set of an expression is not arbitrary. The type rules are organized to guarantee that the possible dynamic types for x all conform to the (static) type of x . This is how the type system keeps polymorphism under control.

It is possible to determine the dynamic type set of x through analysis of the classes in the system to which x belongs, by considering all the attachment and reattachment instructions involving x or its entities.

22.12 ASSIGNER CALL

You may have noted that the syntax for assignment

```
some_variable := some_expression
```

only supports assignment to a **Variable** entity; it does not allow assignment to a field of an object, as in

```
x.a := b [1]
```

Warning: invalid except as abbreviation for procedure call. See below.

Some programming languages permit such assignments, but — if viewed just as assignments — they violate fundamental rules of methodology (information hiding, data abstraction): clients of a class should not have the ability to modify class instances directly; they should only do so through the exported procedures of the class. A typical client call may be

```
x.set_a (b) [17]
```

assuming the author of the class — who is solely responsible for deciding what clients may and may not do — has provided a procedure `set_a` that sets the value of the `a` field. The procedure might have other properties, such as imposing requirements on the new values, or triggering a database update:

```
set_a (x: T)
  -- Update a to value x.
  require
    "Some condition on x, for example to ensure compliance
    with an invariant clause involving a"
  do
    a := x
    "Possibly some other action, for example updating a log
    or database to record that a has been updated"
  ensure
    set: a = x
  end
```

While [1] is not acceptable as a way to let clients modify fields directly, some programmers may find it more directly meaningful than [17] as a notation to represent the procedure call to *set_a*.

Assigner commands provide this syntactic simplification. When you declare a query in a class, you may associate with it an **assigner command**; in the example this means that the author of the supplier class must have declared *a* accordingly, as

```
a: SOME_TYPE assign set_a
```

which specifies *set_a* as the assigner command associated with the query *a*. The consequence of this declaration is to make form [17], *x.a := b*, valid, with the same semantics as form [1], *x.set_a (b)*.

Form [17] is known as an **Assigner_call**.

Remember that it is only a syntactical convenience: Eiffel doesn't permit violating principles of information hiding and data abstraction, as would be the case if clients could directly modify fields of objects. You have no choice but to go through the official interface as defined by the supplier class author. Assigner call— available only if that author has decided to provide it, by specifying an assigner command for the query — simply lets you call the procedure through assignment-like syntax. But the instruction is still a procedure call, not an assignment.

The instruction is in fact more general than a plain assignment since it allows you to use arguments. The target query may have any number of arguments; this is what allows you to write



```
your_array.item (i) := new_value [18]
```

as a shorthand for the procedure call

```
your_array.put (new_value, i) [19]
```

This shorthand is made possible by the declaration of *item* in class *ARRAY*, which specifies *put* as an assigner command:

```
item (i: INTEGER): G alias "[" assign put ...
```

In this case the **alias** "[" specification makes bracket syntax also possible, allowing the following form as a synonym for [19] and hence for [18]:

```
your_array [i] := new_value [20]
```

which makes traditional array assignment syntax available in a fully object-oriented context.

More generally, if q is a query with n arguments and has an associated assigner command p , which must have $n + 1$ arguments, you may use

$$x. q (a_1, a_2, \dots, a_n) := e$$

as an abbreviation for

$$x. p (e, a_1, a_2, \dots, a_n)$$

The syntax is straightforward:



Assigner calls

$$\text{Assigner_call} \triangleq \text{Expression} \text{ ":=" } \text{Expression}$$

The left-hand side is surprisingly general: any expression. The validity rule will constrain it to be of a form that can be interpreted as a qualified call to a query, such as $x.a$, or $x.f(i, j)$; but the syntactic form can be different, using for example bracket syntax as in $a [i, j] := x$.

You could even use operator syntax, as in

$$a + b := c$$

assuming that, in the type of a , the function *plus alias* "+" has been defined with an assigner command, maybe a procedure *subtract*. Then the left side $a + b$ is just an abbreviation for the query call

$$a.\textit{plus} (b)$$

and the *Assigner_call* is just an abbreviation for the procedure call

$$a.\textit{subtract} (c, b)$$

A *Call_chain* — the syntax appears in the study of calls — is a dot-separated sequence of two or more features, each possibly with arguments; examples of *Call_chain* are → Page 626.

$$\begin{aligned} &x.a \\ &\textit{your_array.item} (i) \\ &x.f(b).g(c, d) \end{aligned}$$

Both the validity and the semantics of an *Assigner_call* follow from this construct's role as a syntactic simplification for a call.

As implied by the rules on assigner commands, p must have one more argument than the associated query q . Here are a few examples of assigner calls and their unfolded forms:

Assigner_call	Unfolded form (assuming q has an assigner command p)
$x.q := e$	$x.p(e)$
$x.q(a) := e$	$x.p(e, a)$
$x.f(a, b).q(c, d) := e$	$x.f(a, b).p(e, c, d)$

From this notion we derive the validity rule for assigner calls:



Assigner Call rule	VBAC
<p>An Assigner_call of the form $target := source$, where $target$ and $source$ are expressions, is valid if and only if it satisfies the following conditions:</p> <ol style="list-style-type: none"> 1 • $source$ is <u>compatible with</u> $target$. 2 • The Equivalent Dot Form of $target$ is a qualified Object_call whose feature has an <u>assigner command</u>. 	

The first two clauses ensures the conditions of the definition of “unfolded form” above, so it’s indeed legitimate for the third clause to to rely on the unfolded form of the instruction.

The unfolded form also gives us the semantics:



Assigner Call semantics
<p>The effect of an Assigner_call $target := source$, where the <u>Equivalent Dot Form</u> of $target$ is $x.f$ or $x.f(args)$ and f has an <u>assigner command</u> p, is, respectively, $x.p(source)$ or $x.p(source, args)$.</p>

This confirms that the construct is just an abbreviation for a procedure call.

22.13 SEMI-STRICT OPERATORS



(This section is only for the benefit of readers with a taste for theory, and may be skipped. They bring new light on earlier concepts, but introduce no new language rules.)

*If skipping go to “CON-
DITIONAL REAT-
TACHMENT”, 22.14,
page 615.*

The application of reattachment semantics to argument passing has the interesting consequence of making *semi-strict* implementations possible. Let us see what this means.

The notion of strictness

We may use a definition from programming theory:



Strict, non-strict

An operation is **strict** on one of its operands if it is always necessary to know the value of the operand to perform the operation. It is **non-strict** on that operand if it may in some cases yield a result without having to evaluate the operand.

For a full discussion see the book [“Introduction to the Theory of Programming Languages”](#).

Many common operations are strict on all arguments: for example you cannot compute the sum of two integers m and n unless you know their values, so this operation is strict on both arguments.

Not all operations are strict on all arguments, however. Consider a conditional operation

```
test c yes m no n end
```

WARNING: this is a mathematical notation, not Eiffel syntax.

which yields m if the value of c (a boolean) is true, n otherwise. This is strict on c , but not on the other two arguments, since it does not need to evaluate m when it finds that c is false, or to evaluate n when c is true.

Detecting that an operation is non-strict on an argument may be interesting for performance reasons (since it may avoid unnecessary computations); more importantly, however, non-strict operations may be more broadly applicable than their strict counterparts. This is immediately visible on the previous example: a fully strict version of the **test** operation would always start by evaluating c , m and n ; but then it would fail to yield a result when c is true and n not defined, and when c is false and m not defined. A "semi-strict" version (strict on c but not on m and n) may, however, yield results in these cases, provided m is defined in the first and n in the second.

The need for semi-strict operators

How does this apply to Eiffel programming? Here the operations of interest are calls, of the general form

```
t.r(..., y, ...)
```

and the operands are the target t and the actual arguments such as y , if any. Such a call is always strict on its target (which must be attached to an object). In a literal sense, it is also strict on its actual arguments, since it will need to pass their values to the routine r .

When considering an actual argument such as y , however, it is more interesting to analyze strictness not for the value of y but for the attached object, if any. Then the specification of unconditional reattachment semantics yields two cases, depending on the types of y and of the corresponding formal argument in r :

- A • If both are reference types, the call passes to r a reference, not the attached object (which does not exist if the value of y is void).
- B • If either type is expanded, the call passes the attached object. (The value of y may not be void in this case.)

Case **A** corresponds to case **1** of reattachment semantics, page 596, and case **B** to **2**, **3** and **4**.

In other words, taking the object to be the operand, actual-formal association is non-strict on y in case **A**, and is strict in case **B**.

If the target is a reference and the source is expanded (case **2** of the table), actual-formal association results in reference reattachment, but the source must first be cloned, so that the operation is indeed strict on y .

The call as a whole will be said to be strict if it is strict on all arguments, and *semi-strict* otherwise:



Semi-strict

A call is **semi-strict** if it is non-strict on one or more arguments.

This case is called “semi-strict” rather than non-strict because an Eiffel call is always strict on at least one of its operands: the call’s target.

If a call may be semi-strict and you want to guarantee strictness on a particular argument without changing anything in the routine’s text, this is easy: just use cloning on the actual argument, passing *clone* (y) rather than y . Function *clone* is clearly strict. The reverse change is not always possible: if the routine has a formal argument of expanded type, it will always be strict on the corresponding actuals.

What does semi-strictness mean in practice? Essentially that if both an actual argument y and the corresponding formal argument are of reference types the implementation **may** choose a non-strict argument passing mechanism, which evaluates y when and only when the routine actually needs y ’s value.

The exception is semi-strict boolean operators, as explained below.



Such a semi-strict implementation is possible, but, except in one case, it is **not guaranteed**. Implementations are not required to use a non-strict argument passing mechanism even if the formal and actual arguments are both references. This means that when you write a call of the form

```
t.r (... , y, ...)
```

you must make sure that the value of y , which may be a complex expression, is always defined at the time of call execution — even in cases for which r does not actually need that value. The call may evaluate y anyway.

Consider for example a routine



```

too_strict_for_me
  (i: INTEGER; arr: ARRAY [REAL]; val: REAL): REAL
  do
    if i >= arr.lower and i <= arr.upper then
      Result := val
    end
  end
end

```

which returns the value of its last argument if its first argument, *i*, is within the bounds of the middle argument, an array, and returns 0.0 (the default value for *REAL*) otherwise. Then consider a call in the same class:

```

your_array: ARRAY [REAL]; a: REAL; n: INTEGER
...
a := too_strict_for_me (n, your_array @ n)

```

WARNING: potentially incorrect!

If the value of *n* may be outside of the bounds of *your_array*, then this call is not correct since *your_array* @ *n*, denoting the *n*-th element of *your_array*, is not defined in this case. Semi-strict implementation (non-strict on the last argument) would avoid evaluation of *some.array* @ *n* and hence ensure proper execution of the call, returning zero; but you may **not** assume that the implementation uses this policy.



There is, however, one exception. As will be seen in detail in the discussion of operator expressions, three functions of the Kernel Library class *BOOLEAN*, are required to be semi-strict (that is to say, non-strict on their single argument). These are functions representing a variant of the common boolean operations: and, or, implies. Their declarations in class *BOOLEAN* are

→ "[SEMISTRICT BOOLEAN OPERATORS](#)", 28.6, page 774.

```

conjunction_semistrict alias "and then"
  (other: BOOLEAN): BOOLEAN is do ... end;

disjunction_semistrict alias "or else"
  (other: BOOLEAN): BOOLEAN is do ... end;

implication alias "implies"
  (other: BOOLEAN): BOOLEAN is do ... end;

```

The semantics of these functions readily admits a semi-strict interpretation: ***a and then b*** should yield false whenever *a* is false, regardless of the value of *b*, and similarly for the others. To state this property concisely for all three operations, it is useful to express the value of each, as applied to arguments *a* and *b*, in terms of the above *ad hoc* **test** notation:

```
test not a yes false no b end
test a yes true no b end
test not a yes true no b end
```

*Remember that an operator expression such as **a and then b** stands for a call of target *a* and actual argument *b*. This explains why all the expressions considered here are strict on *a*, since a call is always strict on its target. See [“THE EQUIVALENT DOT FORM”](#), 28.8, page 780.*

This semi-strictness of these boolean operators is important in practice because it makes it possible to use them as conditional operators. As a typical example, again using arrays, it is often convenient to write instructions of the form



```
if
    i >= your_array.lower and then
    i <= your_array.upper and then
    (arr @ n).your_property
then
...

```

where the last condition is not defined unless the first two are true (because *i* would then be outside of the bounds of *arr*). In the absence of a semi-strict version of “and”, it would be much more cumbersome (as Pascal programmers know) to express such examples.

The discussion of boolean operators will show further uses of this semi-strict policy, especially for writing iterators on data structures, with examples from the EiffelBase library.

→ See for example [continue_until](#) from [LINEAR_ITERATION](#) on page =====

More on strictness



(This more theoretical section may be skipped on first reading.)

What about the ordinary boolean operators **and** and **or**? You may expect them to have a strict semantics, but this is not the case — at least not necessarily. Here the language definition is simply less tolerant: it makes it incorrect to evaluate expressions ***a and b*** and ***a or b*** when *b* is not defined, even if *a* has value false in the first case and if *b* has value true in the second case. There is nothing surprising in this convention, which has its counterpart in all other forms of expression except those involving semi-strict operators: no rule in this book will tell you how to compute the value of $m + n$ if the value of the integer expression *n* is not defined.

Because the language definition does not cover cases in which the second operand of **or** or **and** has no value, an implementation that uses **and then** to compute **and**, and **or else** to compute **or**, is legitimate; it may produce results in cases for which a strict implementation would not, but these cases are incorrect anyway.

The reverse is not true: a correct implementation of **and** and **or** does not necessarily provide a correct implementation of **and then** and **or else** since it may be strict. In other words: non-semi-strict does not necessarily mean strict! If you want to guarantee strictness, it does not suffice to rely on the operator **and** and the operator **or**; you should use cloning as suggested above. (For **implies**, which is semi-strict, there is no equivalent non-semi-strict operator, but you can use **not a or b**.)



It is legitimate to ask why the semi-strict property of three boolean operators — **and then**, **or else**, **implies** — is not expressed as part of the language syntax. One could indeed envision a special optional qualifier **nonstrict** applicable to formal arguments of reference type:

implication alias "and then"
(nonstrict other: BOOLEAN): BOOLEAN

WARNING: not legal Eiffel!

Such a facility was not, however, deemed worth the trouble, since the common practice of software development seldom requires semi-strictness outside of two special cases: the three boolean operators just studied; and, as we will see in the relevant chapter, concurrent computation. → *Chapter 33.*

22.14 CONDITIONAL REATTACHMENT

To complete the study of reattachment, there remains to see one mechanism which, like the operations examined so far, may reattach a reference to a different object. The semantics will in fact be reference reattachment; what differs is the validity constraint under which you may apply this mechanism, and also the conditional nature of its effect.

--- REPLACE WITH A SHORT PREVIEW OF Object_test

Limitations of unconditional reattachment



The need for a conditional form of reattachment arises when you must access an object of a certain type **TX**, but the only name you have to denote that object is an expression of type **TY**, for two different types with the “wrong” conformance (**TX** conforms to **TY** rather than the reverse), or even no conformance at all. Normally, you would use the assignment

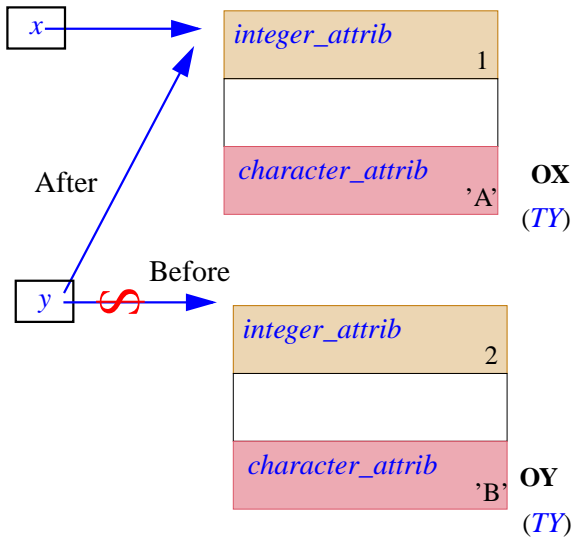
$x := y$

with x of type TX ; but this will not work because the fundamental constraint of unconditional reattachment, expressed in the **Assignment** rule, assumes conformance from y to x . Calling a routine with y as actual argument corresponding to a formal argument x of type TX would also be invalid for the same reason. This conformance property is essential to the soundness of the type system.

22.15 MEMORY MANAGEMENT

A practical consequence of the reference reattachment mechanism, both in the unconditional form (assignment, argument passing) and in the conditional form (assignment attempt), is that some objects may become useless. This raises the question of how, if in any way, the memory space they used may be reclaimed for later use by newly created objects.

For example, the reference reattachment illustrated by the figure below may make the object labeled OY unreachable from any useful object.



Effect of reference reattachment

*This is the same as the
second figure of page
596.*

In a similar way, the result of a cloning operation may make an object unreachable. This may be the case with the middle object (also labeled OY) in the earlier illustration of cloning. ← *First figure on page 595.*

What does it mean for an object to be “useful”? **Remember** that the execution of a system is the execution of a creation procedure (the root creation procedure) on an object (the root object, an instance of the system’s root class). The root object will remain in place for the entire duration of the system’s execution. An object is useful if it may be reached directly or indirectly, following references, from the either root object or any of the local variables of a currently executing routine. Because a non-useful object can have no effect on the remainder of the system’s execution, it is permissible to reclaim the memory space it uses. ← *“System execution”, page 114.*



Should a reattachment as illustrated above (or its clone variant) automatically result in freeing the associated storage? Of course not. The object labeled OY may still be reachable from the root through other reference paths.

It would indeed be both dangerous and unacceptably tedious to lay the burden of object memory reclamation on developers. Dangerous because it is easy for a developer to forget a reference, and to recycle an object’s storage space wrongly while the object is still reachable, resulting in disaster when a client later tries to access it; and unacceptably tedious because, even if you know for sure that an object is unreachable, you should not just recycle its own storage but also analyze all its references to other objects, to determine recursively whether other objects have also become unreachable as a result. This makes the prospect of manual reclamation formidable.

Authors of Eiffel implementation are encouraged to provide a **garbage collection** mechanism which will take care of detecting unreachable objects. Although many policies are possible for garbage collection, the following properties are often deemed desirable:

- **Efficiency:** the overhead on system execution should be low.
- **Incrementality:** it is desirable to have a collector which works in small bursts of activity, being triggered at specified intervals, rather than one which waits for memory to fill up and then takes over for a possibly long full collection cycle. Interactive applications require bursts to be (at least on average) of a short enough duration to make them undetectable at the human scale.
- **Tunability:** library facilities should allow systems to turn collection off (for example during a critical section of a real-time application) and on again, to request a full collection cycle, and to control the duration of the bursts if the collector is incremental.

The Kernel Library class “MEMORY”, A.6.25 CLASS, page 1006, provides such facilities.

22.16 SEMANTICS OF EQUALITY

The previous discussions have shown how to reattach values. A closely related problem, whose study will conclude this chapter, is to **compare** values, for example to see if they are attached to the same object. This raises the question of the semantics of the equality operator `=` and its alter ego the inequality operator `/=`.

If you remember how the study of object duplication (*copy*, *clone* and variants) led us to object comparison (*equal* and its variants), you will probably have anticipated the current section: just as the assignment operator `:=` has the semantics of reference attachment, copy or clone depending on the expansion status of its operands, so will the equality operator `=` have the semantics of reference or object equality. ← “*OBJECT EQUALITY*”, 21.6, page 580.



We can devote all our attention to equality since inequality follows: the effect of `x /= y` is defined in all cases to be that of

not (`x = y`)

Two meanings of equality are *a priori* possible: reference equality, true if and only if two references are either attached to the same object or both void; and object equality.

The previous chapter introduced a function to test object equality: *equal* from the universal class *ANY*, which in its original version will return true if and only if two objects are field-by-field equal. As with copying and cloning operations, it is more prudent to rely on the frozen version *identical*, guaranteeing uniform semantics. (By redefining *is_equal*, you may provide another version of *equal* for a specific class.) For convenience, *identical* (like *equal*) also applies to void values. In the present discussion, “object equality” denotes an operation that can only compare two objects, and so must be applied to non-void references. ← “*OBJECT EQUALITY*”, 21.6, page 580.

Here is the table of possibilities, which closely parallels the corresponding table for unconditional reattachment: ← Page 596.

<i>TYPE OF FIRST</i> →	Reference	Expanded
<i>TYPE OF SECOND</i> ↓		
Reference	[1] <ul style="list-style-type: none"> • Reference equality • Object equality (if neither void) 	[2] <ul style="list-style-type: none"> • Object equality

Possible semantics for shallow equality

NOT a semantic specification but only a list of available possibilities for such a specification. The actual semantics appears next.

Expanded	[3] • Object equality (if first not void)	[4] • Object equality
-----------------	---	---------------------------------



For each of the four cases, we must give a reasonable meaning to the equality operator =. The line of reasoning applied earlier to unconditional reattachment yields the following semantics, which again parallels the table for unconditional reattachment. ← Page 598.

<i>TYPE OF FIRST</i> →	Reference	Expanded
<i>TYPE OF SECOND</i> ↓		
Reference	[1] Reference equality	[2] <i>identical</i>
Expanded	[3] <i>identical</i>	<i>identical</i>

So if *x* and *y* are references the result of a test

```
x = y
```

is true if and only if *x* and *y* are either both void or both attached to the same object; if either or both of *x* and *y* are objects, then the test yields true if and only if they are attached to field-by-field equal objects, as indicated by function *identical_equal* from class *ANY*.

As with unconditional reattachment, the semantics given is the most frequently needed one for each case, and in particular is usually appropriate for operations on arguments of a *Formal_generic_name* type. For more specific semantics, you may use one of the calls

```
equal (x, y)
deep_equal (x, y)
identical_equal (x, y)
identical_deep_equal (x, y)
```

Many container classes of EiffelBase have routines that query a data structure such as a list, set, tree or hash table for occurrences of an object (or more generally a value). This may mean either of two things: does the structure contain a reference to the object of interest? Does it contain a reference to an object equal to it? You can switch between these two interpretations by applying the procedures *compare_objects* and *compare_references* to a certain container, as in *my_list.compare_objects*. This governs not only searching operations, such as the function *has*, but also certain insertion and replacement operations that will only add an element to a structure if it is not already present.

See [“Reusable Software”](#). The notion of container data structure was presented in [10.21, page 292](#), and [12.2, page 349](#).

For basic arithmetic types, which are expanded, the $=$ and \neq operators will always call *identical*. Thanks to the conversion mechanism studied [earlier in this chapter](#), you may use mixed-type equality expressions within the limits of the conversions specified in the corresponding classes. For example the expression $1.0 = 1$ is valid (and will return true) even though it has a *REAL* operand and the other is an *INTEGER*. This is because according to the above semantics the expression means *1.0.identical(1)*, and *INTEGER* converts to *REAL*. Thanks to the [target conversion mechanism](#), you may also write $1 = 1.0$, with the same result.

← [“CONVERSIONS”](#), [22.6, page 591](#).

← [“Accounting for target conversion”](#), [page 770](#)

Feature call

23.1 OVERVIEW

How does a software system perform its job — its computations?

It must first set the stage: create the needed objects and attach them to the appropriate entities. The preceding chapters discussed how to do this. But once it has the objects in place and knows how to access them, the system should do something useful with them.

In Eiffel’s model of computation, the fundamental way to do something with an object is to apply to it an operation which — because the model is class-based, and behind every run-time object lurks some class of the system’s text — must be a feature of the appropriate class.

This is feature call, one of the most important constructs in Eiffel’s object-oriented approach, and the topic of the following discussions.

One of the risks with calls in object-oriented languages is the *void call*: a run-time attempt to apply a feature to an object that doesn’t exist because a reference is void (or, in other terminology, a pointer is null). Eiffel distinguishes itself by making such a failure impossible thanks to the notion of *attached type* and associated constructs studied in previous chapters. Here we will reap the benefits of these mechanisms, which ensure statically — at compile time — that no Eiffel call can apply to a void target. This removes the principal source of run-time failure in object-oriented programming.

Three topics related to calls merit their own discussions in other chapters:

- The validity of calls raises the general question of **type checking**: how to make sure that the target of every call will be an object equipped with the appropriate feature. → [Chapter 25](#).
- A call has a **target**, which must be an object. If the target is known through a reference, we must be sure that the reference will never be void upon execution of the call. → [Chapter 24](#).
- **Operator expressions** are conceptually calls, but use traditional mathematical syntax. We’ll see them as part of the chapter on expressions, although there will be little new to learn about their validity and semantic properties, which are those of calls. → [Chapter 28](#).

23.2 PARTS OF A CALL

A call is the application of a certain feature to a certain object, possibly with arguments. As a consequence, it has three potential components:

- The *target* of the call, an expression whose value is attached to the object.
- The *feature* of the call, which must be a feature of the object's type.
- An *actual argument list*.

The target and argument list are optional; the feature is required.

Here is a typical example showing all three components:

```
remote_bank.transfer_by_wire (20000, Today)
```

This call uses **dot notation**. The target of the call is *remote_bank*; the feature of the call is *transfer_by_wire*; and the actual argument list contains the two elements *20000* and *Today*.

The target is separated from the feature of the call by a period, or *dot*, hence “dot notation”.

If the target is the predefined entity **Current**, representing the “current object” of system execution, as explained below, you may use, instead of the fully qualified form → “*Current object, current routine*”, page 649.



```
Current.print (message) [1]
```

a form which leaves the target implicit:

```
print (message) [2]
```

This is still considered to be a case of dot notation even though the dot is implicit. If the call does include an explicit target and dot, it is **qualified**; otherwise, as in the last example, it is **unqualified**.

In the presence of run-time assertion monitoring, there is a slight semantic difference between [1] and [2]: a qualified call causes invariant checking, an unqualified call doesn't.

A qualified call may have more than one level of qualification and is then said to be a **multidot** call, as in



```
paragraphs (2).line (3).second_word.set_font (Bold)
```

For a feature without arguments, the actual argument list will be absent, as in the source expression of the **Assignment**



```
code := remote_bank.authorization
```

where *authorization* is a query (attribute or function) without arguments.

In some cases we don't need a target object (as in a qualified call) but we still need a target type. If *T* is a type, the notation

$\{T\}.constant_or_external$

denotes a call to a feature *constant_or_external* from *T*. This only makes sense if the feature is either a constant attribute or an external (non-Eiffel) feature; anything else would require a target object.

Non_object_call is a shorthand for “non-object-oriented call”, as in $\{T\}.f(args)$ where *T* is a type. The usual object-oriented style of computation, $x.f(args)$, requires a target object denoted by *x*.

Calls may appear in syntactic forms other than dot notation:

- **Operator expressions**, have the semantics of calls: $a - b$ is, with a feature *minus alias* “-”, equivalent to the dot-notation call $a.minus(b)$. Similarly, with *item alias* “[]”, the expression $x [i]$ has the same semantics as the dot-notation call $x.item [i]$.
- You may also write a **non-object call** of the form $\{T\}.f$ where *T* is a type and *f* is either a constant attribute or an external feature of *T*. This is like a call in dot notation that would not need a target, but only a target type (to determine which *f* to use).

23.3 USES OF CALLS

A call may play either of two syntactic roles: instruction and expression.

A call is a specimen of construct *Call*, covering dot notation, qualified or unqualified, and non-object calls.

Operator_expression (in prefix or infix notation) and *Bracket_expression* are always used as expressions, but a *Call* in dot notation may be either an instruction or an expression. The syntax *productions* for both the *Instruction* and *Expression* constructs indeed include *Call* as one of the choices. To know which one applies, it suffices to look at the feature of the call:

Instruction: page 228;
Expression: page 761.



Call Use rule

VUCN

A *Call* of feature *f* denotes:

- 1 • If *f* is a query (attribute or a function): an expression.
- 2 • If *f* is a procedure: an instruction.

This rule has a validity code, so that compilers and other language processing tools may refer to it when detecting an error such as the use of a procedure call in an expression.

The above examples used calls to *transfer_by_wire*, *print* and *set_font* as instructions, and a call to *authorization* as an expression. The calls to *minus alias* “-” and *item alias* “[]” are also expressions. The non-object call $\{T\}.f$ is an instruction if *f* is a procedure of *T* and an expression otherwise.

23.4 UNIFORM ACCESS

An important property applies to dot-notation calls used as expressions: the notation is exactly the same whether the feature of a **Call** is a function with no arguments or an attribute. The expression



```
pl.age
```

where *pl* is of type *PERSON* is applicable both if the feature *age* of class *PERSON* is a feature of either kind.

If *age* is an attribute, every instance of *PERSON* has a field which gives the value of *age* for the instance. If *age* is a function, that value is obtained, when requested, through some computation, presumably of the difference between the current date and a "birth date" field. For a client containing the above call, however, this makes no difference.

This principle of **uniform access** facilitates smooth evolution of software projects by protecting classes from internal implementation changes in their suppliers. ← First discussed in ["UNIFORMACCESS"](#), 23.4, page 624.

23.5 OPERATOR AND BRACKET FORMS

A call serving as an expression may use, instead of dot notation, the **Operator_expression** form based on unary or binary operators. Both of the two operator expressions, respectively unary (prefix) and binary (infix)



```
- 1
4 - 3
```

are calls to functions of the Kernel Library class *INTEGER*: the first, to the function *negated alias* "-"; the second, to *minus alias* "-". The **Feature Declaration rule** requires a feature associated with a unary or binary operator to be an attribute or function without argument, like *negated*, or a function with one argument, like *minus*. Note that here although both are associated with the same operator – there is no ambiguity since the same rule guarantees that there is at most one feature for each of these signatures.

← Page 162, clause 7; see clause 1. of ["Alias Validity rule"](#), page 163.

The difference between such an operator expression and a **Call** is only syntactical. You may also write the above two expressions as:

```
(|1|).negated
(|4|).minus (3)
```

with exactly the same effect.

The syntax of **Call** requires putting in "target parentheses" (| ... |) around a **Manifest_constant**, such as 1 or 4, to use it as target of a call.

→ ["COMPLEX TARGETS"](#), 23.6, page 625 below.

Similarly, a bracket expression such as

```
your_array [some_index]
```

based on the feature *item alias* "[]" in class *ARRAY*, has the exact same semantics as

```
your_array .item (some_index)
```

The discussion of expressions will formalize the correspondence between the two syntactic forms by defining an **Equivalent Dot Form** for any operator expression.

→ "[THE EQUIVALENT DOT FORM](#)", 28.8, page 780.

23.6 COMPLEX TARGETS

In most cases the target x of a call $x.f(\dots)$ is just an entity: a local variable, an attribute, a formal argument. Sometimes you may want to use a non-elementary expression, such as $a + b$ (where a and b could be not just numbers but, for example, of some type *MATRIX*). Writing $a + b.f(c)$ would, according to precedence rules, denote a sum of two elements, a and the application of f to b . If that's not what you want, you may use a local variable to specify applying f instead to the sum of a and b :

```
local
  sum
do
  ... sum := a + b
  x := sum.f(c) ...
end
```

This technique works but forces the introduction of extra local variables. To avoid them you may use the **parenthesized target** notation (*Expression*):

```
x := (| a + b |).f(c)
```

The symbols use parentheses and a vertical bar. They remove any ambiguity by making clear that the feature, f in this example, is being applied to the whole expression.

You may also use a parenthesized target in connection with bracket notation, as in $(| a + b | [i])$, assuming the type of $a + b$ has a bracket feature.

→ "[BRACKET EXPRESSIONS](#)", 28.7, page 778; see the syntax on page 778.

Note that just using parentheses, as in $(a + b).f(x)$, would not be legal syntactically.

Why indeed not just use plain parentheses? The reason is syntactical. Eiffel **always** treats the semicolon separator as redundant, without making any difference between spaces, new lines and other break characters. If a parenthesized expression were permitted as target of a call, the assertion

```
require
  h
  (a + b).g
```



Would include two clauses. But syntactically the beginning could be parsed as $h(a + b)$, denoting the application of a function h to an argument $a + b$, even though the remainder, $.g$, doesn't have a proper syntactical interpretation.

This syntactical problem is typical of the confusion engendered by the dual use of parentheses, coming from mathematical conventions: as a *grouping* mechanism, as in $(a + b)$; and as a notation for *function application*, as in $f(c)$. The special symbols $(| \dots |)$ avoid any such ambiguity.

23.7 CALL SYNTAX

We'll now examine the syntax of the construct **Call**, describing calls in dot notation, qualified or not, and non-object calls.

Prefix

, infix and bracket forms are specimens of **Expression**; we'll see their syntax in the corresponding [chapter](#), which also defines their semantics in terms of the semantics of calls.

→ "[GENERAL FORM OF EXPRESSIONS](#)", [28.2, page 761](#) and rest of [chapter 28](#).



Feature calls	
Call	\triangleq Object_call Non_object_call
Object_call	\triangleq [Target "."] Unqualified_call
Unqualified_call	\triangleq Feature_name [Actuals]
Target	\triangleq Local Read_only Call Parenthesized_target
Parenthesized_target	\triangleq "(" Expression ")"
Non_object_call	\triangleq "{" Type } "." Unqualified_call

A call is most commonly of the form $a.b \dots$ where $a, b \dots$ are features, possibly with arguments. **Target** allows a **Call** to apply to an explicit target object (rather than the current object); it can itself be a **Call**, allowing multidot calls. Other possible targets are a local variable, a **Read_only** (including formal arguments and **Current**) a "non-object call" (studied below), or a complex expression written as a **Parenthesized_target** (\dots) .

When present, the optional **Actuals** part gives the list of actual arguments:



Actual arguments	
Actuals	\triangleq "(" Actual_list ")"
Actual_list	\triangleq {Expression "," ...}+



As the specification of `Actual_list` indicates, an `Actuals` argument list may not be empty: if f has no formal arguments, you must call it as f or $x.f$, not $f()$ or $x.f()$. This is for simplicity and clarity.

An object-oriented call is either *qualified* or not. It's qualified if it involves at least one dot:



Unqualified, qualified call

An `Object_call` is **qualified** if it has a `Target`, **unqualified** otherwise.

The Address form for Actual serves to pass the address of an Eiffel feature to a foreign (non-Eiffel) routine. See [31.8, page 833](#).

In equivalent terms, a call is “unqualified” if and only if it consists of just an `Unqualified_call` component.

The call $f(a)$ is unqualified, $x.f(a)$ is qualified.

Another equivalent definition, which does not explicitly refer to the syntax, is that a call is qualified if it contains one or more dots, unqualified if it has no dots — counting only dots at the dot level, not those that might appear in arguments; for example $f(a.b)$ is unqualified.



Of our earlier examples

```
print (message)
paragraph (2).line (3).second_word.set_font (Bold)
```

the first is unqualified and the second qualified. Both are instructions if we assume that `print` and `set_font` are procedures in their respective classes. The intermediate components of the second example

```
paragraph (2)
paragraph (2).line (3)
paragraph (2).line (3).second_word
```

are all specimens of construct ----- **FIX** . They may themselves be viewed as calls; any such intermediate call must be an expression (rather than an instruction) so that it may serve as the target of further calls.

The features of all examples so far have arguments. Here are two examples where the call has no argument:



```
paragraph (2).indent;
f := that_word.current_font
```

They assume that `indent` is a procedure with no arguments and that `current_font` is an attribute or function without arguments. As a result, the source the following assignment, in the last example, is a call expression.

For examples of calls using a **Parenthesized_target**, in addition to $(|1|).negated$ and $(|4|).minus(3)$ (more simply written as -4 and $4 - 3$), assume a class **VECTOR** with features *norm* and *plus*:



```
class VECTOR [G → X] feature
  norm: G is do ... end;
  plus alias "+" (other: like Current): like Current
    do ... end
  ... Other features ...
end
```

Then with a and b of type **VECTOR** [T] for some appropriate T you may use the expressions

```
(|u + v|).norm
(|u + v|).norm.f.h.
```

f must be a feature of class X , hence applicable to $(u + v).norm$ since the type G of this expression, a formal generic parameter of **VECTOR**, is constrained by X .

both of which apply function *norm* to the result of applying function *plus* to u with argument v . The syntax specification allows for at most one **Parenthesized_target**, at the beginning of the **Call**. In the second example the **Parenthesized_target** is followed by the **Call** $norm.f.h$.

Thanks to this mechanism, you may use any valid expression as qualifier by parenthesizing it. Without parentheses, the **Call** would be syntactically illegal, as in $3.negated$, or legal but with a different semantics, as with $u + v.norm$ which applies *norm* to v , not to the sum.

→ The dot has the highest precedence of all operators except parentheses, so in the second case it applies to v , not $u + v$. See “[SUBEXPRESSIONS](#)”, 28.3, [page 764](#)

23.8 COMPONENTS OF A CALL

DEFINITION

It is convenient to talk about “the target”, “the target type” and “the feature” of a call.

Target of a call

Any **Object_call** has a **target**, defined as follows:

- 1 • If it is qualified: its **Target component**.
- 2 • If it is unqualified: **Current**.

The target is an expression; in $a(b, c).d$ the target is $a(b, c)$ and in $(|a(b, c) + x|).d$ the target (case 1) is $a(b, c) + x$. In a multidot case the target includes the **Call** deprived of its last part, for example $x.f(args).g$ in $x.f(args).g.h(args1)$.

A `Non_object_call` does not have a target; this is what distinguishes it from an `Object_call`. In both cases, however, there is a target *type*:

Target type of a call

Any `Call` has a **target type**, defined as follows:

- 1 • For an `Object_call`: the type of its target. (In the case of an `Unqualified_call` this is the current type.)
- 2 • For a `Non_object_call` having a type *T* as its `Type` part: *T*.

A call of any kind also has a feature:

Feature of a call

For any `Call` the “**feature of the call**” is defined as follows:

- 1 • For an `Unqualified_call`: its `Feature_name`.
- 2 • For a qualified call or `Non_object_call`: (recursively) the feature of its `Unqualified_call` part.

Case 1 tells us that the feature of *f* (*args*) is *f* and the feature of *g*, an `Unqualified_call` to a feature without arguments, is *g*.

The term is a slight abuse of language, since *f* and *g* are feature names rather than features. The actual feature, deduced from the semantic rules given below and involving dynamic binding, is the **dynamic feature** of the call.

→ “*Dynamic feature of a call*”, page 639

It follows from case 2 that the feature of a qualified call *x.f* (*args*) is *f*. The recursive phrasing addresses the multidot case: the feature of *x.f* (*args*).*g*.*h* (*args1*) is *h*.

23.9 NON-OBJECT CALLS

The remaining sections of this chapter discuss the validity and semantics of calls. The most interesting cases are the object-oriented form of call, *x.f* (*args*), involving dynamic binding, and its unqualified variant *f* (*args*). They will occupy most of the discussion. Let us dispose first of a specific case, available mostly to facilitate interaction with non-object-oriented facilities: `Non_object_call`. In an example such as

```
{CHARACTER_CODES}.Underscore
```

we use a `Non_object_call` to access directly a constant attribute present in a “utility class”, `CHARACTER_CODES`. Were this mechanism not available in the language, you could still obtain the desired effect by either:

- Making the enclosing class inherit from `CHARACTER_CODES`, so that it can directly access its features such as `Underscore`.

- Declaring an entity *codes*: `CHARACTER_CODES` and using `codes.Underscore`.


Using inheritance as in the first solution is a bit heavy-handed for such a simple purpose. With the second solution, you must declare an entity that you won't use for anything else; in addition, if `CHARACTER_CODES` is not an expanded class, you'll have to perform a creation instruction `create codes` to obtain the corresponding object. All this is a diversion. With the `Non_object_call` you state, with no fuss, exactly what you need: feature `Underscore` from class `CHARACTER_CODES`.

The mechanism is applicable only in limited cases: we only allow `{T}.f...` if `f` is a constant, like `Underscore`, or an external (non-Eiffel) function, as in



```
{NETWORK_CONTROLLER}.open_channel(port_number, timeout)
```

The reason is that any feature other than a constant attribute or an external feature might need to work on the target, which a `Non_object_call` lacks. Even an external feature could be a problem through its assertions: consider a call



```
open_channel (pn: INTEGER; to: REAL)
  -- Open a channel on port number pn with timeout to.
  require
    valid_state
  external
    "C"
  end
```

Warning: makes above
`Non_object_call`
invalid.

where `open_channel`, in class `NETWORK_CONTROLLER`, is an external routine with two arguments. The precondition has an `Unqualified_call` to `valid_state`, a function that might use the current object. Or it might not; but this can be tricky to determine, so we should just ban such assertions.

To specify both the validity and the semantics it is convenient to treat a `Non_object_call` as a special case of an `Object_call`:

Imported form of a `Non_object_call`

The **imported form** of a `Non_object_call` of Type `T` and feature `f` appearing in a class `C` is the `Unqualified_call` built from the original `Actuals` if any and, as feature of the call, a fictitious new feature added to `C` and consisting of the following elements:

- 1 • A name different from those of other features of `C`.
- 2 • A `Declaration_body` obtained from the `Declaration_body` of `f` by replacing every type by its deanchored form, then applying the generic substitution of `T`.

This definition in “unfolded” style allows us to view $\{T\}.f(args)$ appearing in a class C as if it were just $f(args)$, an Unqualified_call, but appearing in C itself, assuming we had moved f over — “imported” it — to C .

← “TWO-TIER DEFINITION AND UNFOLDED FORMS”, 2.11, page 100.

In item 2 we use the “deanchored form” of the argument types and result, since a type like a that makes sense in T would be meaningless in C . As defined in the discussion of anchored types, the deanchored version precisely removes all such local dependencies, making the type understandable instead in any other context.

← “Deanchored form of a type”, page 344.

This notion helps us express the validity rule:



Non-Object Call rule *VUNO*

A Non_object_call of Type T and feature $fname$ in a class C is valid if and only if it satisfies the following conditions:

- 1 • $fname$ is the final name of a feature f of T .
- 2 • f is available to C .
- 3 • f is either a constant attribute or an external feature whose assertions, if any, use neither **Current** nor any unqualified calls.
- 4 • The call’s imported form is a valid Unqualified_call.

Condition 2 requires f to have a sufficient export status for use in C ; there will be a similar requirement for Object_call. Condition 3 is the restriction to constants and externals. Condition 4 takes care of the rest by relying on the rules for Unqualified_call.

→ Through the notion of export validity defined in the next section.

We also use the imported form to define the semantics:



Non-Object Call Semantics

The effect of a Non_object_call is that of its imported form.

23.10 CLASS VALIDITY

The rest of this chapter considers the most common — but also more delicate — case: object calls, involving dynamic binding. First, validity.

The basic idea is straightforward: in $x.f(args)$ appearing in a class C , the base class of x must have a feature f , that feature must be available (exported) to C , and the elements of $args$ must conform to the corresponding formal arguments as declared for f ; in addition, the type of x must be *strict* to avoid the possibility of calls on a void target. In the unqualified version $f(args)$, r must be a feature of the current class and the arguments must conform. For the overwhelming majority of cases this is all you need to remember.

The full story is more subtle; in fact the next two chapters are devoted to filling in the details. In the present discussion we will examine the **Class-Level validity** of a call, which it is convenient to define in four parts:

- **Export validity**, to ensure that f is exported to the client class.
- **Argument validity**, to ensure that the $args$ are of the right number and type.
- **Target validity**, to ensure that x is not void.

Target validity is defined in the next chapter; the following one will tackle the remaining notion of *System-Level validity*.



Elsewhere in this book, validity rules are of the form: “A specimen of construct C is valid if and only if ...”. The rules of this section appear instead as: “A **Call** is **X**-valid if and only if ...”, where **X** is one of Export, Argument, Target and Class-Level. The following chapter will define a **Call** as “valid”, without further qualification, if and only if it is System-Level-valid and Class-Level-valid. Since the three components of Class-Level validity address distinct aspects, it is convenient for compilers to produce error messages that refer to each of them; so you can view the rules below, as normal validity rules, except that they are “only if” but not “if”.

Export validity

The first of the three components of Class-Level validity, export validity, ensures that the caller is entitled to use the “feature of the call”:



Export rule

VUEX

An **Object_call** appearing in a class C , with $fname$ as the feature of the call, is **export-valid** for C if and only if it satisfies the following conditions.

- 1 • $fname$ is the final name of a feature of the target type of the call.
- 2 • If the call is qualified, that feature is available to C .



This defines export validity “for” a certain class C . Usually we consider a call appearing in a given class text, so we say just “export valid” to mean export-valid for the current class. In the discussion of type checking, we’ll need to consider the call, and its export validity, for an arbitrary descendant of the original class.



For an unqualified call f or $f(args)$, only condition 1 is applicable, requiring simply (since the target type of an unqualified class is the current type) that f be a feature, immediate or inherited, of the current class.

For a qualified call $x.f$ with x of type T , possibly with arguments, condition 2 requires that the base class of T make the feature available to C : export it either generally or selectively to C or one of its ancestors. (Through the Non-Object Call rule this also governs the validity of a `Non_object_call {T}.f`)

← As defined in “[Available for call, available](#)”, page 211.

As a consequence, $s(...)$ might be permitted and $x.s(...)$ invalid, even if x is **Current**. The semantics of qualified and unqualified calls is indeed slightly different; in particular, with invariant monitoring on, a qualified call will — even with **Current** as its target — check the class invariant, but an unqualified call won’t.

Clause 2 only applies to qualified calls. Clearly, a routine r of a class C can call another routine s of C on the current object unqualified, regardless of the export status of s . But in a qualified call $x.s(...)$ the routine s must always be exported to C , even if x is of type C .

Because this property sometimes surprises programmers accustomed to the conventions of other languages, it is useful to make it prominent:



Export Status principle

The export status of a feature f :

- Constrains all qualified calls $x.f(...)$, including those in which the type of x is the current type, or is **Current** itself.
- Does not constrain unqualified calls.

This is a validity property, but it has no code since it is not a separate rule, just a restatement for emphasis of condition 2 of the Export rule.

That clause also takes care of the multi-dot case: in $a.b.c$, the target, $a.b$, must itself satisfy the same condition. (This use of recursion is justified since the target has one more level of dot notation than the original **Call**, so the recursion cannot go on forever.)



In such multi-dot calls, all that counts is availability to the class C where the call appears; availability to intermediate classes is irrelevant. For example, if C contains the call

`next_paragraph.line (3).second_word.set_font (Bold)` [3]

where successive features are of types *PARAGRAPH*, *LINE* and *WORD*, export validity means that *PARAGRAPH* must make function *line* available to *C*, *LINE* must make *second_word* available to *C*, and *WORD* must make *set_font* available to *C*. It does not matter whether *second_word* is available to *PARAGRAPH*, or *set_font* is available to *LINE*. To understand why, note that any such call may be rephrased in single-dot form:

```
l: LINE; w: WORD
...
l := next_paragraph.line (3)
w := l.second_word
w.set_font (Bold)
```



This shows multi-dot notation as just a notational facility — although an important one, avoiding the need for intermediate variables such as *l* and *w*.

Argument validity

The second component of Class-Level validity ensures that the number and types of actual arguments match those of formals:



Argument rule

VUAR

An export-valid call of target type *ST* and feature *fname* appearing in a class *C* where it denotes a feature *sf* is **argument-valid** if and only if it satisfies the following conditions:

- 1 • The number of actual arguments is the same as the number of formal arguments declared for *sf*.
- 2 • Every actual argument of the call is compatible with the corresponding formal argument of *sf*.

→ The *S* in *ST* and *sf* is for “static”. See “[Descendant Argument rule](#)”, page 667.

For simplicity, the definition assumes export validity, ensuring that *f* exists.

Condition 2 is the fundamental type rule on argument passing, which allowed the discussion of direct reattachment to treat **Assignment** and actual-formal association in the same way. An expression is *compatible* with an entity if its type either conforms or converts to the entity’s type.

← Page “[ROLE OF REATTACHMENT OPERATIONS](#)”, 22.2, page 588.

In a generic context, condition 2 relies on the Generic Type Adaptation rule: in a call *a.sf(y)* where *a* is of type *C [T]* and *C [G]* has the routine *sf(x: G)*, the type to which *y* must conform is *T* — not *G*, which makes no sense outside of the text of *C*.

← Page 367.

A call to a feature with no arguments trivially satisfies the Argument rule if it doesn't include any **Actuals**. As noted at the beginning of this chapter, it's syntactically illegal to write a call as $f()$ or $x.f()$; either the feature has formal arguments and you must specify the corresponding **Actuals** in parentheses, or it doesn't and you just don't include any **Actuals** list.



A consequence of the Arguments rule is that Eiffel doesn't directly allow a routine to be called with a variable numbers of arguments. But there's an easy way to achieve this purpose: simply give the routine a formal argument of a tuple type. With



```
print_formatted (values: TUPLE [STRING])
```

a corresponding call may have any number of arguments greater than one as long as the first is a **STRING** (representing a format). Clients may call it as

```
print_formatted (some_format, int, re, str)
```

whatever the types of *int*, *re*, *str*, as long as the routine body handles them properly.

Target validity and Void-Safe Eiffel

The last component of Class-Level validity guarantees that a call $x.f(\dots)$ can never fail at run time because x turned out to be attached to a void reference:



Target rule

VUTA

An **Object_call** is **target-valid** if and only if either:

- 1 • It is unqualified.
- 2 • Its target is an attached expression.

Unqualified calls (case 1) are always target-valid since they are applied to the current object, which by construction is not void.

Another way of expressing this observation is to note that an unqualified call $g(\dots)$ is always the result of a qualified call $x.f(\dots)$ (or of an original root call to f , starting a system), where f , directly or indirectly, calls g unqualified on the same target x that was used for f ; that x cannot have been void since the call to f would then never have started in the first place. Put yet another way, the unqualified call is generally equivalent to **Current.g** (...) where **Current**, representing the current object, is never void.

← "System execution",
page 114.

For the target expression x to be “attached”, in case 2, means that the program text guarantees — statically, that is to say through rules enforced by compilers — that x will never be void at run time. This may be because x is an entity declared as attached (so that the validity rules ensure it can never be attached a void value) or because the context of the call precludes voidness, as in if $x \neq \text{Void then } x.f(\dots) \text{ end}$ for a local variable x . The precise definition will cover all these cases.

Combining the rules

Class-Level validity is the combination of the previous three constraints, and is the basic validity rule for calls:



Class-Level Call rule

VUCC

A call of target type ST is **class-valid** if and only if it is export-valid, argument-valid and target-valid.

The last requirement, target validity, may raise issues for older Eiffel systems not yet checked for this property. The Standard, for that reason, allows compilers to offer a special tolerance, with the associated risk of run-time failure, as a temporary measure to facilitate transition:



Void-Unsafe

A language processing tool may, as a temporary migration facility, provide an option that waives the target validity requirement in class validity. Systems processed under such an option are **void-unsafe**.

Bla bla bla =====

Void-unsafe systems are not valid Eiffel systems. Since void safety was not enforced by previous versions of Eiffel, compilers may need, all the same, to provide an option that temporarily lifts this requirement. Including the notion of “void-unsafe” in the language definition enforces a consistent way for various compilers to provide this transition facility.

23.11 INTRODUCTION TO CALL SEMANTICS

Let us now examine the semantics of calls. This section and the next few discuss the concepts; the formal rules are collected at the end.

It will suffice to consider as working example a qualified *Call*

$target.fname(y_1, \dots, y_n)$

→ “PRECISE CALL SEMANTICS”, 23.17, page 652.



where *target* is an expression, *fname* is a feature name of the appropriate class, and the y_i are expressions. We may further assume that *target* is either a **Parenthesized** expression or a single **Unqualified_call**, in other words that the **Call** is not a multi-dot of the form $a.b.c \dots .fname(\dots)$.

Concentrating on this example simplifies the discussion but doesn't lose any generality:

- By not considering multi-dot expressions we simply understand a multi-dot call as a succession of single-dot calls, as in the above call to *set_font*. The formal semantic definition will justify this equivalence. ← [3], page 633.
- We already noted that infix, prefix and bracket expressions always have an Equivalent Dot Form. ← “OPERATOR AND BRACKET FORMS”, 23.5, page 624.
- If there are no arguments, we simply consider that n is zero.
- Lastly, what of unqualified calls *fname* ($y_1 \dots, y_n$)? We'll also be able to handle them as a special case of qualified calls thanks to the notion of *current object* as discussed below.

We will also assume, on the basis of the preceding discussion of Void-Safe Eiffel, that at the time of execution *target* will not be void: either it is expanded, directly denoting an object, or it is a reference attached to an object. This is a universal requirement on call targets; if you want a feature to work on a void value for one of its operands x — definitely a useful possibility in some cases — you must treat x as an argument, not the target. You can only use x as target if its static type is an attached. Remember that this is not necessarily the declared type T of x : if T is not attached you can use the **Object_test**

```

if  $x \neq \text{Void}$  then
     $x.f(\text{args})$ 
    -- The static type of this occurrence of  $x$  is attached  $T$ 
else
    ... No calls with target  $x$  permitted here ...
end

```

This discussion leads to our first semantic definition for calls:

Target Object

The **target object** of an execution of an **Object_call** is:

- 1 • If the call is qualified: the object attached to its target.
- 2 • If it is unqualified: the current object.

→ “Current object, current routine”, page 649.

“Current object” has only been defined informally so far and its precise definition is forthcoming. The definitions, however, avoid circularity.

SEMANTICS





The notion of target object is used in all the semantic specifications for calls in the rest of this chapter.

In the qualified case (case [1](#)) you will use, to obtain the target's value, the rules of expression semantics. They yield the target object itself for an expanded type, and for a reference type a reference attached to that object.

The validity rules, as noted, prevent a void reference. For compilers that support an option that doesn't enforce void-safety requirements, we provide an exception type anyway:



Failed target evaluation of a void-unsafe system

In the execution of an (invalid) system compiled in `void-unsafe` mode through a language processing tool offering such a migration option, an attempt to execute a call triggers, if it evaluates the target to a void reference, an exception of type `VOID_TARGET`.

23.12 DYNAMIC BINDING

So we want to execute or evaluate `target.fname(args)` at a certain instant of system execution, on a non-void `target`.

"Execute" for an instruction, "evaluate" for an expression. The rest of the discussion uses the first of these terms for simplicity, except when the context implies an expression.

Assumed to be non-void, `target_value` is attached to a target object `OD`. `OD` is a direct instance of some type `DT`, of base class `D`. `D` (the generating class of `OD`) must be effective: otherwise `DT` could not have any direct instance.

The expression `target_value` has a certain type `ST`, of base class `S`. Recall that `ST` is also called — when we need more precision — the **static** type of `target`, and `DT` its **dynamic** type at the time the call is executed. The static type is obvious from the software text and is fixed for any occurrence of `target` in that text; polymorphism means that the dynamic type may change in successive executions of the call, as a result of reattachments.

The typing constraints imply that `DT` will always conform to `ST`, and hence that `D` is a descendant of `S`. The validity rules just seen imply that the feature of the call, `fname`, must be the final name in `S` of a feature of `S`, available to the class which includes the call. Let `sf` be that feature.

Actually, as you guessed, we couldn't care less about `sf`. Using `sf` for the call would be committing the gravest possible crime in object technology: **static binding**. What matters is not the type of `target` (what was declared in the software text) but the type of the object attached to `target_value` (what is actually found at run time). Using that type, `DT`, to determine the appropriate feature, yields the appropriate policy: **dynamic binding**.

The `D` in the type and class names stands for "dynamic", the `S` for "static".

"Generating class": [19.2, page 506](#); "Dynamic type", [page 606](#); "Type of an expression", [page 783](#); "POLYMORPHISM", [22.11, page 606](#)

← "Reattachment principle", [page 599](#).

The feature to be used, *df*, is the version of *sf* that applies to *D* and hence to *DT*. The two features will be different if *DT* or some intermediate class has redefined *sf*. The purpose of such a redefinition is precisely to ensure that the feature performs for instances of *DT* in a way that differs from its default behavior for instances of *ST*. Not using the redefined version would mean renouncing the power of the inheritance mechanism.

The word "version", as used here, has a precise meaning, defined as part of inheritance. Every feature of a class has a single "dynamic binding version" in any descendant of that class; that version is the result of applying any redefinition, undefinition or effecting that may have occurred since the original introduction of the feature. The definition takes into account the case of repeated inheritance, for which the Select subclause removes any ambiguity that could be caused by conflicting redefinitions on different inheritance paths, or by the replication of an attribute.

← "[Dynamic binding version](#)", page 468.

The following semantic definition captures dynamic binding:



Dynamic feature of a call

Consider an execution of a call of feature *fname* and target object *O*. Let *ST* be its target type and *DT* the type of *O*. The **dynamic feature** of the call is the dynamic binding version in *DT* of the feature of name *fname* in *ST*.

Behind the soundness of this definition stands a significant part of the validity machinery of the language:

- The rules on reattachment imply that *DT* conforms to *ST*.
- The Export rule imply that *fname* is the name of a feature of *ST* (meaning a feature of the base class of *ST*).
- As a consequence, this feature has a version in *DT*; it might have several, but the definition of "dynamic binding version" removes any ambiguity.

← "[Reattachment principle](#)", page 599.

← "[Export rule](#)", page 632.

Combining the last two semantic definitions enables the rest of the semantic discussion to take for granted, for any execution of a qualified call, that we know both the target object and the feature to execute. In other words, we've taken care of the two key parts of **Object_call** semantics, although we still have to integrate a few details and special cases.

23.13 THE IMPORTANCE OF BEING DYNAMIC



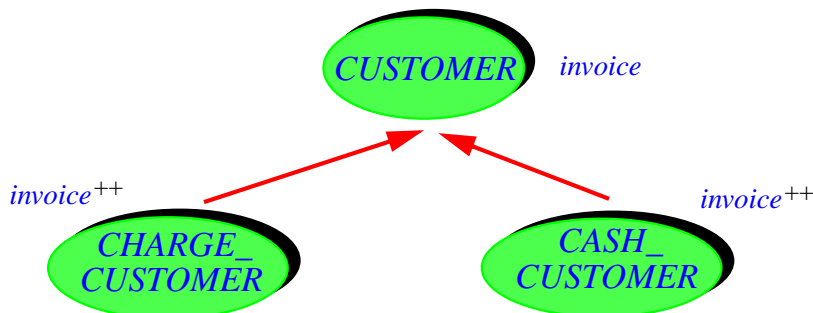
Dynamic binding is not just a useful convention but a condition of correctness. Every qualified call to an exported routine of a class must preserve its invariant, so as never to produce an inconsistent object — one that would not satisfy the invariant of its own generating class. This means that *sf* must preserve the invariant *SI* of *S*, and *df* the invariant *DI* of *D* (a possibly strengthened form of *SI*). But there is of course no requirement that *sf* preserve *DI*; in fact, the designer of *S* usually did not even know about class *D*, which may have been written much later by someone else. Static binding could then apply to an object, *OD*, a feature, *sf*, which does not preserve the invariant of the generating class — the ultimate disaster in the execution of a software system.

← As required by the definition of “Class consistency”, page 247.

← As implied by the definition of “Unfolded form of an assertion”, page 287.

Dynamic binding, then, is the only meaningful policy. In some cases, of course, *sf* and *df* are the same feature because no redefinition has occurred between *S* and *D*, or simply because *S* and *D* are the same class. Then static and dynamic binding trivially have the same semantics. A compiler or other language processing tool which is able to detect such situations through careful analysis of a system’s source text use this insight to generate slightly more efficient object code. This is perfectly acceptable as long as the system’s run-time behavior implements the semantics of dynamic binding.

Beyond its theoretical necessity, dynamic binding plays an essential role in the Eiffel approach to software structuring. It means that clients of a number of classes providing alternative implementations of a certain facility can let the mechanisms of Eiffel execution select the appropriate implementation automatically, based on the form of each polymorphic entity at the time of execution.



Kinds of customer



As a typical example, assume a class *CUSTOMER* with a procedure *invoice* used to bill customers. Heirs *CHARGE_CUSTOMER* and *CASH_CUSTOMER* may redefine this procedure in two different ways to account for different forms of invoicing. Then a Variable *c* of type *CUSTOMER* may be attached, at some run-time instant, to an instance of *CHARGE_CUSTOMER* or *CASH_CUSTOMER*. A call of the form

c.invoice

will, thanks to dynamic binding, be treated appropriately in each case.

This is a great advantage for the authors of client classes containing such calls, since they do not need to test explicitly for every possible case (charge customer, cash customer), and may integrate the introduction of a new case — such as check customers — at minimal change in their classes.

23.14 ONCE ROUTINES

We know the target of the call is not void, and we know (through dynamic binding) what feature was really meant. So the next thing to do is to execute the associated routine body, right? Wrong. The routine might be a **once routine**, designed to be executed only once, or once in a while.

Once basics

As you will remember, a **Routine_body** may start (other than **deferred** and **external** cases) not only with **do** but also with the keyword **once**, possibly followed by one or more “once keys” in parentheses as in **once** (“*THREAD*”).

← “*ROUTINE BODY*”, 8.5, page 222.

In the basic case without once keys, this means that you want the routine’s body to be executed at most once in the entire system execution. The first time — if at all — someone calls the routine, its body will be executed, with the actual arguments given if any; if it’s a function, it will return its result normally. Any subsequent call, however, will not cause any new execution of the routine body or initialization of local variables; it will return immediately to the caller, giving as result — if the routine is a function — the value recomputed by the first call, whether an object (if the result type is expanded) or an object reference.

A constraint on once functions was introduced as part of the Feature Declaration rule (condition 5): if the enclosing class is generic, the result type may not be one of the formal generic parameters. This is necessary for the function to provide a consistent result: since the first client that calls the function will determine the result of all later calls, the result type must be meaningful for all clients; but different clients may use different actual generic parameters for the class. The formal parameter, which stands for any possible actual generic parameter, would represent incompatible types.

← Page 162.

Once uses

The **Once** mechanism is a versatile tool allowing flexible initialization and access to shared information in an O-O environment. In particular:

- **Smart initialization:** to make sure that a library works on a properly initialized setup, write the initialization procedure as a **Once** and include a call to it at the beginning of every externally callable routine of the library.

The alternative would be to require clients to take care of the setup themselves by calling an initialization procedure;. Because this is error-prone, you'll want to check in the library itself that the initialization has been done; but then you might just as well take care of it silently and avoid bothering clients. In any case, you need a way to find out if initialization has indeed been done, typically through a flag — which must also have been initialized, only pushing the problem further. Once procedures provide a general solution.

- **Shared objects:** To let various components of a system share an object, represent it as a once function that creates the object. Clients will just “call” that function, although in all cases but the first such a call just returns a reference to the object created the first time around.

In this last case, the scheme is a common one in Eiffel programming:



```
shared_object: SOME_REFERENCE_TYPE
    -- A single object useful to several clients
    once
        ... ; create Result
    end
```

This declaration may for example appear in a service class inherited by the affected clients.

Predefined once keys

What exactly does “once” mean? By default, the semantics is to execute the routine body once over every execution of a system. By using once keys, however, you may exert finer control, specifying an execution every once in a specific while. For example by declaring a routine as



```
r: SOME_TYPE
    -- A single object useful to several clients
    once ("OBJECT")
        ...
    end
```


you specify that the body will be executed the first time it is called on **any specific instance** of the class. This provides welcome flexibility. Assume for example that some objects have associated information, much bigger than the object itself and needed only in certain cases. This could be (among many other examples) the list of all previous states of an object stored in a database. It's not something that you want to load by default into memory with every object that you retrieve from the database; but it should be easy to access when you need it. The following function does the job smoothly and (for the programmer) effortlessly:



```

history: ARRAY [like Current)
    -- A single object useful to several clients
    once ("OBJECT")
    create Result (...)
    ... Retrieve previous values and fill Result with them ...
end
  
```

Traditional programming techniques — using flags to check whether the function has been called — would be quite cumbersome here, especially if you have a need for several such functions.

The following once keys have a preset meaning:

```

"OBJECT"      -- Once for each instance
"THREAD"     -- Once per execution of a thread
"PROCESS"    -- Once per execution of a process
  
```

"PROCESS" is the default, equivalent to not specifying a once key.

Further once tuning

For even more flexibility, you may define your own meaning of “while” in “once in a while”. You’ll do this by choosing as once key an arbitrary string, beyond the three possibilities listed above. You can take advantage of this possibility in two ways.

First, you can control the meaning from outside of the Eiffel text, by defining it in the **once** clause of the Ace file. The recommended convention in this case is to use a once key of the form **\$KEYNAME**, using the dollar sign that serves in some scripting languages to denote the value of a variable. The Ace specification can set the key to mean, for example, **THREAD** in some executions and **PROCESS** in others, depending for example on the amount of multi-threading supported.

→ “*ONCE CONTROL*”, B.11, page 1033.

In the Eiffel text itself, you can go further by deciding when once is enough and when you want more of it. More precisely you may **refresh** a once key; this means that the next call of any once routine that lists it as one

of its once keys will execute its body. To refresh keys, class *ANY* has a feature *onces* of type *ONCE_MANAGER* (a Kernel Library class) which → *ONCE_MANAGER*, A.6.29, page 1010. you can use for such calls as

```
onces.refresh("SOME_KEY")
onces.refresh_some(["SOME_KEY", "OTHER_KEY"])
onces.refresh_all
onces.refresh_all_except(["SOME_KEY", "OTHER_KEY"])
```

You can also query *onces.nonfresh_keys*, returning an array of strings, to find out what keys have been exercised by at most one function.

A possible way to implement feature *onces* in class *ANY* is to make it a once function itself.

These are clearly advanced techniques, but they can help considerably in the building of sophisticated systems.

Once routine semantics

In defining the semantics of once routines we will rely on the following notion whose meaning follows directly from the preceding discussion:

SEMANTICS

Freshness of a once routine call

During execution, a call whose feature is a once routine *r* is **fresh** if and only if every feature call started so far satisfies any of the following conditions:

- 1 • It did not use *r* as dynamic feature.
- 2 • It was in a different thread, and *r* has the once key "*THREAD*" or no once key.
- 3 • Its target was not the current object, and *r* has the once key "*OBJECT*".
- 4 • After it was started, a call was executed to one of the refreshing features of *onces* from *ANY*, including among the keys to be refreshed at least one of the once keys of *r*.

Case 2 indicates that “once per thread” is the default in the absence of an explicit once key



Note that *every* call started so far has to satisfy *any* of the conditions listed. So *r* is fresh for example if:

- It hasn't been called at all.
- It has been called on different objects, and is declared **once** ("*OBJECT*").
- It's declared **once** ("*SOME_KEY*") and there has been, since the last applicable execution of *r*, a call *onces.refresh* ("*SOME_KEY*").

An applicable call — for example, with the once key "*OBJECT*", a call on the same object — makes *r* unfresh again, since the rule's conditions have to apply to every call started so far.

The call *onces.refresh_all* is understood to refresh all once routines, including those without an explicit once key.

Also note that the condition applies to calls *started* so far; so if a once routine is directly or indirectly recursive, its self-calls will not execute the body (in the absence of an intervening explicit refresh) and, for a function, they will return the **Result** as computed so far.

Latest applicable target and result of a non-fresh call

The **latest applicable target** of a non-fresh call to a once routine *df* to a target object *O* is the last value to which it was attached in the call to *df* most recently started on:

- 1 • If *df* has the once key "*OBJECT*": *O*.
- 2 • Otherwise, if *df* has the once key "*THREAD*" or no once key: any target in the current thread.
- 3 • Otherwise: any target in any thread.

If *df* is a function, the **latest applicable result** of the call is the last value returned by a fresh call using as target object its latest applicable target.

From these observations we may define the semantics of a call to a once routine. For fresh calls a once routine behaves like a non-once routine, and the rule correspondingly refers to the Non-Once Call Routine Execution rule appearing later in this chapter:

→ "*Non-Once Routine Execution Semantics*", page 652.



Once Routine Execution Semantics

The effect of executing a once routine df on a target object O is:

- 1 • If the call is fresh: that of a non-once call made of the same elements, as determined by Non-once Routine Execution Semantics.
- 2 • If the call is not fresh and the last execution of f on the latest applicable target triggered an exception: to trigger again an identical exception. The remaining cases do not then apply.
- 3 • If the call is not fresh and df is a procedure: no further effect.
- 4 • If the call is not fresh and df is a function: to attach the local variable **Result** to the latest applicable result of the call.

Case 2 is known as “*once an once exception, always a once exception*”. If a call to a once routine yields an exception, then all subsequent calls for the same applicable target, which would normally yield no further effect (for a procedure, case 3) or return the same value (for a function, case 4) should follow the same general idea and, by re-triggering the exception, repeatedly tell the client — if the client is repeatedly asking — that the requested effect or value is impossible to provide.



There is a little subtlety in the definition of “latest applicable target” as used in case 4. For a once function that has already been evaluated (is not fresh), the specification does not state that subsequent calls return the result of the first, but that they yield the value of the predefined entity **Result**. Usually this is the same, since the first call returned its value through **Result**. But if the function is **recursive**, a new call may start before the first one has terminated, so the “result of the first call” would not be a meaningful notion. The specification states that in this case the recursive call will return whatever value the first call has obtained so far for **Result** (starting with the default initialization). A recursive once function is a bit bizarre, and of little apparent use, but no validity constraint disallows it, and the semantics must cover all valid cases.

Had you detected this case?

The Once Routine Execution Semantics rule describes the effect of executing a **Call** once we know its run-time feature df , its target object O and its arguments arg_values . For the full context, we need the general semantics rule for calls, which comes at the end of this chapter and, in the once case, relies on the above rule to specify the effect of the call once its components have been determined.

23.15 ATTRIBUTES AND EXTERNALS

We may now concentrate on the case of a qualified `Object_call` whose feature is not a once routine. From the discussion of features and routines, ← *Chapters 5 and 8.* the dynamic feature of the call, if not a “once”, may be one of:

S1 • An attribute

S2 • An external routine (whose implementation is outside the system’s direct reach, being written in another language).

S3 • A non-once, non-external routine.

The syntax for `Routine_body` includes a fifth case: a routine with a `deferred` body. This case doesn’t apply here, however, since as noted above `D` has a direct instance and hence must be effective. ← *Page 222.*

In case S1, `df` is an attribute; the object `OD` has a field corresponding to `df`. Then the call is an expression, whose value is that field. The sole effect of the call is to return that value.

In case S2, `df` is an external routine; execution of the call will mean passing the values of the actual arguments to that external routine, waiting for it to complete its execution, and obtaining its result if it is a function. The semantics of argument passing and of routine execution — which may depend on the conventions of the routine’s native language — are examined in the chapter on interfaces with other languages. → *Chapter 31.*



Note that the target object is **not** passed by default to an external routine. If it’s needed for the computation, you should pass it as actual argument to the routine, which should include a corresponding formal.

These two cases will be integrated in the final call semantics rule. For the moment we may concentrate on the remaining one. → *“General Call Semantics”, page 653.*

23.16 THE MACHINERY OF EXECUTING CALLS

We’ll investigate the effect of a non-once, non-external routine (S3) of actual arguments `args`, target object `O` and dynamic feature `df`. This will also lead us to the semantic notions of current object and current routine.

Scheme for a routine call

The semantic rule will specify the effect of the call as the result of applying a sequence of steps. This doesn’t mean that the code must execute these exact steps, only that its effect must be the same as if it did. Somewhat informally and ignoring assertion monitoring, the steps are:

- 1 • Using the semantics of direct reattachment, attach every formal argument of *df* to the value of the corresponding actual from *args*. ← “SEMANTICS OF REATTACHMENT”, 22.7, page 593.
- 2 • If *df* has any local variables, save their current values if any call to *df* has been started but not yet terminated; then initialize each local variable to the default value of its type.
- 3 • If *df* is a function, initialize the predefined entity *Result* to the default value for the function’s return type.
- 4 • Execute the Compound of *df*’s Internal body, according to the conventions described next.
- 5 • If *df* is a function, the call is an expression. The value returned for that expression is the value of *Result* after the previous step.
- 6 • If the values of local variables have been saved under 2, restore the variables to these earlier values.

The Argument rule ensure that in step 1 the actual arguments (if any) match the formals in number, and that each actual is compatible with (conforms or converts to) the corresponding formal. ← Page 634.

In step 2, the default initialization values are the same as for the initialization of attributes in a Creation instruction.

The saving of local variables under 2, and their restoring under 6, are necessary because routines may be directly or indirectly recursive: the body of *df* may contain a call to another routine, and that routine may turn out to be *df*, or it may recursively call *df*. As a result, step 4 may start the whole process again on the same routine. The saving and restoring ensure that each incarnation of *df* recovers its local variables when it is resumed after a recursive call.

*The local variables include **Result**: “LOCAL VARIABLES AND RESULT”, 8.6, page 225.*

Current object and routine

To interpret the Compound of a routine’s Internal body in step 4, a little mystery remains. Assume the text of routine *df*, in class *D*, has the following simple form:

```

fname
do
    some_proc
    x.other_proc
end

```

*The feature name might be something other than *fname* as a result of renaming.*



where x is an attribute of D , *some_proc* a procedure of D , and *other_proc* is a procedure applicable to x . Step 4 — the core of the call’s execution — consists of executing the two instructions of the **Compound**.

But what exactly do they mean? What does x represent? To what object should the computation apply *some_proc*?

To answer these questions we must put ourselves in the global context of system execution and remember how anything ever gets executed. Quoting from a very early part of this book:

To execute (or “run”) a system on a machine means to cause the machine to apply a creation instruction to the system’s root class.

← “*System execution*”,
page 114.

In all but trivial cases, the root’s creation procedure will create more objects and execute more calls. This extremely simple semantic definition of system execution has as its immediate consequence to yield a precise definition of the *current object* and *current routine*. At any time during execution, the current object is the object to which the latest non-completed routine call applies, and the current routine cr is the feature of that call:



Current object, current routine

At any time during the execution of a system there is a **current object CO** and a **current routine cr** defined as follows:

- 1 • At the start of the execution: **CO** is the root object and cr is the root procedure.
- 2 • If cr executes a qualified call: the call’s target object becomes the new current object, and its dynamic feature becomes the new current routine. When the qualified call terminates, the earlier current object and routine resume their roles.
- 3 • If cr executes an unqualified call: the current object remains the same, and the dynamic feature of the call becomes the current routine for the duration of the call as in case 2.
- 4 • If cr starts executing any construct whose semantics does not involve a call: the current object and current routine remain the same.



Clause 4 addresses “*constructs whose semantics does not involve a call*” (rather than “constructs other than a call”). This is because the semantics of a construct that is not a call may involve a call; this is the case with an **Expression**, whose semantics is defined through an Equivalent Dot Form denoting a call.

Note the implicit recursion in case [2](#): to know the target object of a call *target.fname (args)*, we must evaluate *target*, which may itself be a call, whose evaluation requires using the above rule recursively.



There appears to be a cycle in the definitions since this definition of current object and current routine refers to “dynamic feature”, defined in terms of “target object”, itself defined in terms of “current object”. You will note on closer examination, however, that this is not a real problem: the definition of target object only refers to the current object in the case of an Unqualified_call, for which the relevant clause in the definition of current object retains an object already known from the context.

← Page [639](#).

← Page [637](#).

← Clause [2](#), page [637](#).

← Clause [3](#), page [649](#).

Naming the current object

Even though the current object is at the heart of the execution machinery, most calls in dot notation do not refer explicitly to the current object: if you need a **Call** with the current object as target, you may just write it as an Unqualified_call, which does not name its target.

For some other kinds of operation, however, you may need an explicit notation to refer to the current object. An example is equality comparison. Assume a function computing the distance between two points, which might be written in a class *POINT* as



```
distance alias "|-|" (other: POINT): REAL
    -- Distance of current point to other.
do
    ...
end
```

The routine’s implementation may need to determine whether the *other* point is in fact the same point as the current object:

```
if “other is not the same as the current point” then
    Result := “... Normal distance computation ...”
end
    -- Otherwise Result will be zero
```

To express the condition after **if** you may use the predefined entity **Current**:

```
if Current /= other then ...
```

As noted above, an Unqualified_call such as *some_proc* or *x* does not need to use **Current** explicitly as its target, although you may if you want to:

```
Current.some_proc
Current.x
```


with the only difference that, under assertion monitoring, qualified calls such as these cause evaluation of the invariant; unqualified calls don't.

It may also be convenient to use **Current** in connection with binary features. Thanks to the infix alias "`|-`", you may use the above *distance* function to express the distance of two points *p1* and *p2* as *p1* `|-` *p2*. To express in a similar form the distance to *p2* of the current point, you may write

Current `|-` *p2*

but even this use of **Current** is not strictly necessary, since there's always an identifier name, here *distance*, for such a feature, so that you may also use the plain **Unqualified_call**

distance (*p2*)

Similarly, if a class contains a unary function *negated alias* "`-`", you may express the negation of the current object as `- Current` as well as just *negated*.

Current, as indicated by its place in the syntax as one of the choices for the construct **Read_only**, is a **read-only entity**: you can't assign to it, or use it at the target of a creation instruction. A notation such as

← "*EXPRESSIONS AND ENTITIES*", 19.8, page 512.

Current.*q* := *v* [4]

is permitted only if *q* is a query of the enclosing class and it has an associated **assigner procedure**, say *p*. Then [1] is simply a shorthand for an unqualified call

← "*ASSIGNER CALL*", 22.12, page 607.

p (*v*) [5]

If *q* has arguments, **Current**.*q* (*a1*, *a2*) := *v* is an abbreviation for *p* (*a1*, *a2*, *v*). In either case, the instruction can't change **Current**.

The following rule gives the precise meaning of **Current**, distinguishing in particular between reference and expanded cases:



Current Semantics

The value of the predefined entity **Current** at any time during execution is the current object if the current routine belongs to an expanded class, and a reference to the current object otherwise.

23.17 PRECISE CALL SEMANTICS

We can now collect into precise rules the understanding of call semantics developed over the preceding sections. The rule for a `Non_object_call` appeared at the beginning of this chapter, so we only need to consider the case of an `Object_call`. For once routines we may refer to the earlier rule.

← [“Non-Object Call Semantics”, page 631](#); [“Once Routine Execution Semantics”, page 646](#)

Rule for non-once routines

Assume we have an `Object_call` and, at a particular stage of execution, we know the target object, the dynamic feature — which is not a “once” — and the argument values. Here then is the effect:



Non-Once Routine Execution Semantics

The effect of executing a non-once routine *df* on a target object *O* is the effect of the following sequence of steps:

- 1 • If *df* has any local variables, including **Result** if *df* is a function, save their current values if any call to *df* has been started but not yet terminated.
- 2 • Execute the body of *df*.
- 3 • If the values of any local variables have been saved in step 1, restore the variables to their earlier values.

General call semantics

We have semantics for executing routines, both once (the earlier rule) and non-once (the last rule). To have the full semantics of calls we need a more general rule, since:

← [“Once Routine Execution Semantics”, page 646](#).

- Both of the previous rules assumed that we know the target object, the dynamic feature, and argument values. But the form of a qualified call, *target.fname (args)*, doesn’t give us that information; the execution must obtain the target object from *target*, the dynamic feature from that object and *fname*, and the argument values from *args*. We’ve actually given ourselves the rules to do this; but to make the semantics precise we need to specify the **order** in which to apply these rules. We’ll require that the target be evaluated first, giving us the dynamic feature as a consequence, and then the arguments in the order listed.
- The rules covered non-external routines only; we must include the attributes and external routines, two cases discussed informally so far.
- Execution of the feature body (step 2 of the last rule) may use the formal arguments. We need to specify how to attach them to the actuals’ values.

← [“Target Object”, page 637](#); [“Dynamic feature of a call”, page 639](#).

← [“ATTRIBUTES AND EXTERNALS”, 23.15, page 647](#)

- Finally, the scheme does not yet include assertion monitoring.

The following rule fills these gaps:



General Call Semantics

The effect of an **Object_call** of feature *sf* is, in the absence of any exception, the effect of the following sequence of steps:

- 1 • Determine the target object **O** through the applicable definition.
- 2 • Attach **Current** to **O**.
- 3 • Determine the dynamic feature *df* of the call through the applicable definition.
- 4 • For every actual argument *a*, if any, in the order listed: obtain the value *v* of *a*; then if the type of *a* converts to the type of the corresponding formal in *sf*, replace *v* by the result of the applicable conversion. Let *arg_values* be the resulting sequence of all such *v*.
- 5 • Attach every formal argument of *df* to the corresponding element of *arg_values* by applying the Reattachment Semantics rule.
- 6 • If the call is qualified and class invariant monitoring is on, evaluate the class invariant of **O**'s base type on **O**.
- 7 • If precondition monitoring is on, evaluate the precondition of *df*.
- 8 • If *df* is not an attribute, not a once routine and not external, apply Non-Once Routine Execution Semantics to **O** and *df*.
- 9 • If *df* is a once routine, apply the Once Routine Execution Semantics to **O** and *df*.
- 10 • If *df* is an external routine, execute that routine on the actual arguments given, if any, according to the rules of the language in which it is written.
- 11 • If *df* is a self-initializing attribute and has not yet been initialized, initialize it through the Default Initialization rule.
- 12 • If the call is qualified and class invariant monitoring is on, evaluate the class invariant of **O**'s base type on **O**.
- 13 • If postcondition monitoring is on, evaluate the postcondition of *df*.

An exception occurring during any of these steps causes the execution to skip the remaining parts of this process and instead handle the exception according to the Exception Semantics rule.

← Page 637.

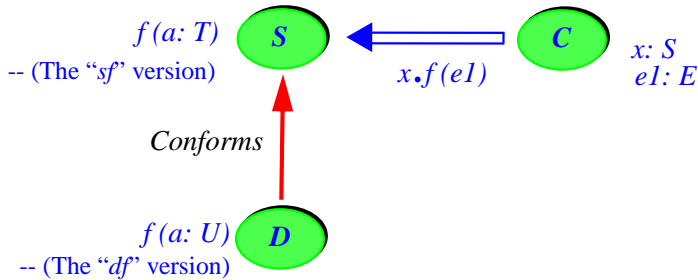
← “*Reattachment Semantics*”, page 600.

For steps **1** and **3**, the “applicable definitions” are those of Target Object and Dynamic Feature, as recalled above.

There is considerable implicit recursion in this definition: the target and the argument are expressions, and in many cases they will be calls, or operator expressions whose semantics is also defined as call semantics. So in steps [1](#), [3](#) and [4](#) we are potentially relying on the semantic rules of this chapter, including the above rule itself. The rule for once routines relies, for fresh calls, on the rule for non-once routines, so step [9](#) again causes recursion.



Step [4](#) specifies a somewhat subtle but important property: the precedence, statically, of convertibility over conformance. We know that every actual argument must be *compatible with* the corresponding formal: conform or convert to it. System validity will ensure that this requirement applies both to the “static” version of the feature df and to the “dynamic” version sf . Remember that sf is the feature named known from the text of the call: with $x.f(e1)$, if x is of type S , sf is the feature of name f in S ; as a result of dynamic binding, if x at execution time is attached to an object of a descendant type D , then df is the version in D .



Effect of redefinition on a client call

But while we want the type E of $e1$ to be compatible with the formal arguments to both the sf and df , we want it, for every one of them, **in the same variant**: either conformance in both cases, or convertibility in both cases. Assume E conforms to T ; then it cannot also convert to it. Now assume that E does not conform to U , the new formal argument type in D , but by some twist of fate E actually convert to U . Do we want to accept the call as descendant-argument-valid for D ? System validity tells us “no”. Accepting this would be confusing for the author of C , who does not realize that a conversion might be going on (since there’s none in the case of the original f).

← “Conversion principle”, page 408.

In addition, although this is not the main concern, the compiler writer would face the similar problem of not knowing whether to generate conversion code or not for the call.

So step [4](#) requires that we take care of any conversion on the basis of the argument types for the **static** feature sf ; only then, in step [5](#), do we attach the values of actuals to formals. Note that the types in these attachments may still be different, but no further conversion will be involved, only conformance.

23.18 CALLS AS EXPRESSIONS

The two uses of a **Call** are, as we know, as an **Instruction** or as an **Expression**, specifically the **Basic expression** variant. If f is a query (attribute or routine), a valid call

← “*Call Use rule*”, page 623. **Instruction**: page 228; **Expression**: page 761.

$x.f(args)$

or any of the other applicable variants — unqualified, non-object, multi-dot — is an expression, and can be included in a larger expression, such as $a + x.f(args) + b$.

For the instruction case we’ve seen all we need about calls. But to understand an expression we must also know its **type** and its **value**; these are defined for every kind of expression and we must now — as the final part of specifying calls — say what they are for a call used as expression.

First, the type. To make this concept useful in practice we must carry type analysis across class boundaries by defining the type of a call **with respect to** a certain type. Assume that x , in a class C , is of type $D[U]$, where $D[G]$ is a generic class with a query f of type G . The Call Expression Type definition given below will tell us that the type of $x.f$ is the type of f with respect to the type of x , that is to say with respect to $D[U]$. Now f , a query of D , is also a query of $D[U]$ thanks to the definition of “**feature of a type**” in the discussion of genericity. Its type as defined in D is G , which in the context of $D[U]$ we must understand, through the Generic Type Adaptation rule, as representing the associated actual generic parameter, U .

The following rule determines the type of a call:



Type of a Call used as expression

Consider a call denoting an expression. Its **type** with respect to a type CT of base class C is:

- 1 • For an unqualified call, its feature f being a query of CT : the result type of the version of f in C , adapted through the generic substitution of CT .
- 2 • For a qualified call $a.e$ of Target a : (recursively) the type of e with respect to the type of a .
- 3 • For a Non_object_call: (recursively) the type of its imported form.

In case 2, the recursion applies to a ; the type of the part after the dot, e , is determined through the general Expression Type definition — itself of course dependent, in several of its clauses, on the type of call expressions, causing more recursion.

→ “*Type of an expression*”, page 783 (see among others its clauses 6 and 11).

Finally the semantics. If a call is used as an expression its execution will, in addition to any other actions, return a result:



Call Result

Consider a **Call** c whose feature is a query. An execution of c according to the General Call Semantics yields a **call result** defined as follows, where O is the target object determined at step 1 of the rule and df the dynamic feature determined at step 3:

- 1 • If df is a non-external, non-once function: the value attached to the local variable **Result** of df at the end of step 2 of Non-Once Routine Execution Semantics.
- 2 • If df is a once function: the value attached to **Result** as a result of the application of Once Routine Execution Semantics.
- 3 • If df is an attribute: the corresponding field in O .
- 4 • If df is an external function: the result returned by the function according to the external language's rule.

For a Non_object_call, whose semantics is defined in terms of the imported form, this definition also applies, as a consequence, to the execution of the imported form.

← “*Non-Object Call Semantics*”, page 631



Functions should not produce any durable change to their environment; their sole role should be to return their result, and any computation they perform should be auxiliary to that goal. You may use **only** postcondition clauses to turn this methodological advice into an enforceable rule.

This book often refers, especially in the discussion of expressions, to the **value** of a call used as an expression. Here is what this precisely means: → Chapter 28.



Value of a call expression

The **value** of a **Call** c used as an expression is, at any run-time moment, the result of executing c .

Eradicating void calls

24.1 OVERVIEW

In the object-oriented style of programming the basic unit of computation is a qualified feature call

```
x.f(args)
```

which applies the feature f , with the given arguments $args$, to the object attached to x . But x can be a reference, and that reference can be void. Then there is *no* object attached to x . An attempt to execute the call would fail, triggering an exception.

If permitted to occur, void calls are a source of instability and crashes in object-oriented programs. For other potential run-time accidents such as type mismatches, the compilation process spots the errors and refuses to generate executable code until they've all been corrected. Can we do the same for void calls?

It has long been considered that this was too hard to do in practice. This chapter shows otherwise.

Eiffel indeed provides a coordinated set of techniques that guarantee the absence of void calls at execution time. The actual rules are specific conditions of more general validity constraints — in particular on attachment and qualified calls — appearing elsewhere; in the following discussion we look at them together from the viewpoint of ensuring their common goal: precluding void calls.

The language resulting from the combination of these rules may be called **Void-Safe Eiffel**; this name is mainly useful for comparison with earlier versions of Eiffel which did not enjoy void safety. The language as described in this book guarantees it, so Void-Safe Eiffel is just Eiffel.

24.2 OVERALL SCHEME

The basic idea is simple. Its the combination of three rules:

- A qualified call $x.f(args)$ is **target-valid** — a required part of being plain valid — if the type of x is **attached**, “Attached” is here a static property, deduced from the declaration of x (or, if it is a complex expression, of its constituents).
- A reference type with a name in the usual form, T , is attached. To obtain a **detachable** type — meaning that *Void* is a valid value — use $?T$.
- The validity rules ensure that attached types — those without a $?$ — deserve their name: an entity declared as $x: T$ can never take on a void value at execution time. In particular, you may not assign to x a detachable value, or if x is a formal argument to a routine you may not call it with a detachable actual. (With a detachable target, the other way around, you are free to use an attached or detachable source.)

The validity rules ensuring these three properties are: for calls,

24.3 THE OBJECT TEST

This leaves the more subtle case of `Object_test`. Such an expression doubles up as a declaration of a “very local” entity that is guaranteed, within its scope, to be attached at run time to an object of a specified type. A typical use of an object test (highlighted) is in the instruction

```
if {l: T} your_file.last_item then
    other_operations
    l.some_feature_of_T
end
```

The value of the expression is true if and only if the expression given, `your_file.last_item`, is attached to an object of type T (and hence not void). Evaluating the expression in addition makes it possible, within a small *scope*, to use l to denote that value. This is what makes it possible, in the example, to execute the feature call `l.some_feature_of_T`: since the object test, when true, guarantees that l is attached to an object of type T , we may apply the relevant features.

Why this temporary binding of an expression to a local entity -----



Object test

Object_test \triangleq "{" Identifier ":" Type "}" Expression

An **Object_test** of the form $\{x: T\} \text{exp}$, where *exp* is an expression, *T* is a type and *x* is a name different from those of all entities of the enclosing context, is a boolean-valued expression; its value is true if and only *exp* is attached to an instance of *T* (hence, non-void). In addition, evaluating the expression has the effect of letting *x* denote that value of *exp* over the execution of a neighboring part of the text known as the **scope** of the **Object_test**. For example, in **if** $\{x: T\} \text{exp then } c1 \text{ else } c2 \text{ end}$ the scope of the **Object_test** is the compound in the **then** part, *c1*. Within *c1*, you may use *x* as a **Read_only** entity, knowing that it has the value *exp* had on evaluation of the **Object_test**, that this value is of type *T*, and that it cannot be changed during the execution of *c1*.

The following rules define these notions precisely.

---- ADD EXPLANATIONS ----

Object-Test Local

The **Object-Test Local** of an **Object_test** is its **Identifier** component.



Object Test rule

VUOT

An **Object_test** *ot* of the form $\{x: T\} \text{exp}$ is valid if and only if it satisfies the following conditions:

- 1 • *x* does not have the same lower name as any feature of the enclosing class, or any formal argument or local variable of any enclosing feature or **Inline_agent**, or, if *ot* appears in the scope of any other **Object_test**, its Object-Test Local.
- 2 • *T* is an attached type.

Condition 2 reflects the intent of an **Object_test**: to test whether an expression is *attached* to an instance of a given type. It would make no sense then to use a detachable type.

----- SEE HERE -----

Conjunctive, disjunctive, implicative; Term, semistrict term

Consider an **Operator_expression** e of boolean type, which after resolution of any ambiguities through precedence rules can be expressed as $a_1 \S a_2 \S \dots \S a_n$ for $n \geq 1$, where \S represents boolean operators and every a_i , called a **term**, is itself a valid **Boolean_expression**. Then e is:

- **Conjunctive** if every \S is either **and** or **and then**.
- **Disjunctive** if every \S is either **or** or **or else**.
- **Implicative** if $n = 2$ and \S is **implies**.

A term a_i is **semistrict** if in the corresponding form it is followed by a semistrict operator.

In the implicative case the expression is of the form a_1 **implies** a_2 and a_1 is a semistrict term. Note that because we accept $n = 1$ a simple expression involving none of the given operators (but possibly involving **not**) is both conjunctive and disjunctive; this is convenient for the following definition.

----- MORE EXPLANATORY TEXT NEEDED -----

Scope of an Object-Test Local

The scope of the Object-Test Local of an Object_test ot includes any applicable program element from the following:

- 1 • If ot is a semistrict term of a conjunctive expression: any subsequent terms.
- 2 • If ot is a term of an implicative expression: the next term.
- 3 • If **not** ot is a semistrict term of a disjunctive expression e : any subsequent terms.
- 4 • If ot is a term of a conjunctive expression serving as the Boolean_expression in the Then_part in a Conditional: the corresponding Compound.
- 5 • If **not** ot is a term of a disjunctive expression serving as the Boolean_expression in the Then_part in a Conditional: any subsequent Then_part and Else_clause.
- 6 • If **not** ot is a term of a disjunctive expression serving as the Exit_condition in a Loop: the Loop_body.
- 7 • If ot is a term of a conjunctive expression used as Unlabeled_assertion_clause in a Precondition: the subsequent components of the Attribute_or_routine.
- 8 • If ot is a term of a conjunctive expression used as Unlabeled_assertion_clause in a Check: the subsequent components of its enclosing Compound.

The definition ensures that, for an Object_test $\{x: T\} \text{exp}$, we can rest assured that, throughout its scope, x will never at run time have a void value, and hence can be used as the target of a call.

Object Test semantics

The value of an Object_test $\{x: T\} \text{exp}$ is true if the value of exp is attached to an instance of T , false otherwise.

In particular, if x is void (which is possible only if T is a detachable type), the result will be false.

Object-Test Local semantics

For an **Object_test** $\{x: T\} \text{exp}$, the value of x , defined only over its scope, is the value of exp at the time of the **Object_test**'s evaluation.

Outside of the scope, the value is not defined. This poses no problem since the Entity rule makes any use of x invalid outside of its scope. Note that within the scope, the value of x will always — as achieved by the very definition of scope in its various cases — be attached to an instance of T . This is precisely what we want. This value can never change, since an Object-Test Local is (from the definition of entities) a **read-only** entity.

24.4 VOID TESTS

Read-only void test

A **read-only void test** is a **Boolean_expression** of one of the forms $e = \text{Void}$ and $e \neq \text{Void}$, where e is a read-only entity.

Scope of a read-only void test

The **scope** of a read-only void test appearing in a class text, for e of type T , is the scope that the Object-Test Local ot would have if the void test were replaced by:

- 1 • For $e = \text{Void}$: **not** ($\{ot: T\} e$).
- 2 • For $e \neq \text{Void}$: $\{ot: T\} e$.

This is useful if T is a detachable type, providing a simple way to generalize the notion of scope to common schemes such as **if** $e \neq \text{Void}$ **then** ..., where we know that e cannot be void in the **Then_part**. Note that it is essential to limit ourselves to read-only entities; for a variable, or an expression involving a variable, anything could happen to the value during the execution of the scope even if e is initially not void.

Of course one could always write an **Object_test** instead, but the void test is a common and convenient form, if only because it doesn't require repeating the type T of e , so it will be important to handle it as part of the Certified Attachment Patterns discussed next.

24.5 CERTIFIED ATTACHMENT PATTERNS

----- EXPLAIN

Certified Attachment Pattern

A **Certified Attachment Pattern** (or **CAP**) for an expression exp whose type is detachable is an occurrence of exp in one of the following contexts:

- 1 • exp is an Object-Test Local and the occurrence is in its scope.
- 2 • exp is a read-only entity and the occurrence is in the scope of a void test involving exp .

A CAP is a scheme that has been proved, or certified by sufficiently many competent people (or computerized proof tools), to ensure that exp will never have a void run-time value in the covered scope.

- The CAPs listed here are the most frequently useful and seem beyond doubt. Here too compilers could be “smart” and find other cases making $exp.f$ safe. The language specification explicitly refrains, however, from accepting such supposed compiler improvements: other than the risk of mistake in the absence of a public discussion, this would result in some Eiffel texts being accepted by certain compilers and rejected by others. Instead, a compiler that *accepts* a call to a detachable target that is not part of one of the official CAPs listed above is **non-conformant**.
- The list of CAPs may grow in the future, as more analysis is applied to actual systems, leading to the identification, and certification by human or automatic means, of safe patterns for using targets of detachable types.

24.6 ATTACHED EXPRESSIONS

Attached expression

An expression *exp* of type *T* is **attached** if it satisfies any of the following conditions:

- 1 • *T* is attached.
- 2 • *T* is expanded.
- 3 • *exp* appears in a Certified Attachment Pattern for *exp*.

This is the principal result of this discussion: the condition under which an expression is *target-valid*, that is to say, can be used as target of a call because its value is guaranteed never to be void at any time of evaluation. It is in an Expanded type's nature to abhor a void; attached types are devised to avoid void too; and Certified Attachment Patterns catch a detachable variable when it is provably not detached.

Typing-related properties

25.1 OVERVIEW

In discussing calls, the previous chapter covered syntax and semantics, but set aside any consideration of validity – even though its semantic definitions only apply to valid constructs. It is time now to come back to the second horse of our troika and examine what it takes to make a call meaningful.

This Part does not define any new rules, only a few definitions that facilitate discussion of type issues.

Catcall

A **catcall** is a run-time attempt to execute a **Call**, such that the feature of the call is not applicable to the target of the call.

The role of the type system is to ensure that a valid system can never, during its execution, produce a catcall.

“Cat” is an abbreviation for “Changed Availability or Type”, two language mechanisms that, if not properly controlled by the type system, could cause catcalls.

Calling features, it was already noted, is the principal means of performing computations in Eiffel. This is why the title of this chapter does not just read "validity of calls", but "type checking", since the type safety of a system is essentially defined by the validity of its calls. This is also why, in an approach that places so much emphasis on helping developers produce correct and robust software, it is crucial to ask what could go wrong at run time with a call – and see what we can do *before* run time to prevent it from going wrong.

Consider the basic form of a call in dot notation:

$target.fname(y1, \dots, yn)$

For this to be properly executed, *target* must be attached to an object DO, and DO must be equipped with a feature corresponding to *fname*; that feature must have a signature (types of arguments and result, if any) and a specification (precondition and postcondition) compatible with what the caller expects.

Not all of these requirements may be handled statically by mere analysis of the software text. To ascertain statically that DO will always exist (that is to say, that *target* will never have a void value) and that the assertions will always be satisfied, we would need theorem and program provers beyond the reach of current software technology. For these properties, the presentation has reluctantly settled for run-time checks which, if not satisfied, may trigger exceptions.

For the remaining properties, however, the picture is brighter. Assuming the object DO exists, it is a direct instance of a certain class *D*, and if we have enough information about the possible *D* we will know statically what features they have. Determining the possible classes and checking that their features match the corresponding calls will enable us to perform static type checking. This chapter explains how to achieve this goal.

As usual, we will take an informal look first, then examine the precise rules.

25.2 SYNTAX VARIANTS



As noted in the previous chapter, dot notation is only one possible form of call. You may also use operator and bracket expressions, resembling traditional mathematical and programming language notation. The difference is just syntactical; an *Operator_expression*



$a + b$

is semantically a call having *a* as its target, *plus alias* "+" as its feature, and *b* as its single argument. The chapter on expressions formalizes this notion by defining, for every kind of expression, a call in dot notation with the same validity properties and semantics: the Equivalent Dot Form.s

→ "THE EQUIVALENT DOT FORM",
28.8, page 780.

For simplicity, this chapter will assume that all expressions are in Equivalent Dot Form.

25.3 BASIC CONCEPTS

The original notion of argument validity considered, for a call $x.fname$ with x of type ST , the feature of name $fname$ in ST . But with polymorphism and dynamic binding we run into the possibility that we'll call a feature with different argument types, from a descendant. As a consequence we must generalize the concept, making it *relative* to a descendant:



Descendant Argument rule VUDA

Consider a call of target type ST and feature $fname$ appearing in a class C . Let sf be the feature of final name $fname$ in ST . Let DT be a type conforming to ST , and df the version of sf in DT . The call is **descendant-argument-valid** for DT if and only if it satisfies the following conditions:

- 1 • The call is argument-valid.
- 2 • Every actual argument conforms, after conversion to the corresponding formal argument of sf if applicable, to the corresponding formal argument of df .

ST stands for Static Type
and DT for Dynamic Type.

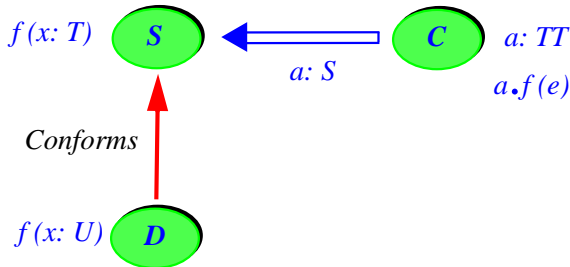
DT can, as a special case, be ST itself. In that case the rule is automatically satisfied as a consequence of the Argument rule invoked by clause 1; clause 2 is then redundant since it follows directly from the second clause of the Argument rule, which told us that ← Page 634.

“Every actual argument of the call is compatible with the corresponding formal argument of sf .”

where compatibility is conformance or convertibility. But if we now consider a proper descendant, where sf has been redefined into a new feature df , the argument types might be different; with polymorphism and dynamic binding, we want the call to be argument-valid not only with respect to ST but also for any applicable descendant DT . Hence clause 2.



Why doesn't that clause transpose its Argument rule counterpart from *sf* to *df* and just tell us that every actual argument should be *compatible with* the corresponding formal for *df*? The reason is that while we want compatibility with both the *sf* and *df* arguments we want it, for every one of them, **in the same variant**: either conformance in both cases, or convertibility in both cases. In a case like this:



*Effect of
redefinition on
a client call*

the call $a.f(e)$ in the client C of S is argument-valid only if TT conforms or converts to T , the type declared for the corresponding formal. Assumes TT conforms to T ; then it cannot also convert to it. Now assume that TT does not conform to U , the new formal argument type in D , but by some twist of fate TT actually convert to U . Do we want to accept the call as descendant-argument-valid for D ? The rule tells us “no”. Accepting this would be confusing for the author of C , who does not realize that a conversion might be going on (since there's none in the case of the original f).

← “*Conversion principle*”, page 408.

In addition, although this is not the main concern, the compiler writer would face the similar problem of not knowing whether to generate conversion code or not for the call.

So clause 2 requires more than compability: it wants *conformance* after possible conversion to the type of the original formal argument in *sf*. Once a conversion, always a conversion.

25.4

Like the above validity constraint, the definition relies on further notions to be defined next: export validity, argument validity and the dynamic class set.



Single-level Call rule

VUSC

A call of target x is **system-valid** if for any element D of the dynamic class set of x it is export-valid for D and descendant-argument-valid for D .

Export validity will require suppliers to make the needed features available to D ; argument validity will require every actual argument to conform to the corresponding formal argument.

The validity of a call at either level – class or system – will require both export and argument validity. The only difference is that for class validity you need only apply these criteria to S , the type declared for the target x of the call, whereas for system-level checking you will need to consider all possible dynamic classes of x . S

25.5 SYSTEM-LEVEL VALIDITY

Although class-level validity may at first appear sufficient, the typing problem is in fact less trivial than the above would suggest. The reason is polymorphism and dynamic binding, which forces us to take into account not just the declared types of entities, but also their possible dynamic types (their dynamic type set).

Polymorphism means that the type used to declare the *target* (S above) is not the only possible type for the object DO to which the call will apply. To see this, let us extend the context introduced above:

```

class C feature
  target: S; other D -- D must be a descendant of S
  y: SOME_TYPE
  ...
  routine
    do
      if some_test then target := other end;...
      target.fname (y);
    end
  ...
end

```

where the Assignment rule requires D to be a descendant of S . Because of the possible polymorphism resulting from the assignment of *other* to *target*, the type of the object DO may now be not just S but T as well.

In other words, we need to consider not only the type of *target* as it results from the declarations, called the **static type** if there is any ambiguity, but also the set of all the types that *target* may assume at run-time as a result of polymorphic attachments. This set was defined in the discussion of polymorphism as the **dynamic type set** of *target*; a member of that set is said to be a possible **dynamic type** for *target*.

The Assignment rule is on page ==. Since the classes are non-generic, conformance is the same relation as "descendant of".

The base classes of the possible dynamic types constitute the **dynamic class set** of *target*.

In this example all types are classes, so that the dynamic type set and dynamic class set are the same, but with generic derivation, expansion and anchored types we will need to reintroduce the distinction between types and classes.

Dynamic type sets and dynamic class sets were defined in [22.11, page 606](#), and are covered more extensively below: first in [25.9, page 677](#), and then in [25.10, page 681](#), for the precise rules.

What then is the actual type constraint? It still applies to a class *D* the conditions defined above for class-level validity:

- *D* must have a feature corresponding to *fname*, available to *C*.
- That feature must have the required signature.

The common goal of the type mechanisms and rules of the language is to ensure that every call is both class-valid and system-valid.

In a simple world we would expect any class-valid call to be system-valid. Unfortunately this is not always the case because of two important properties of the inheritance mechanism:

On P1, see "adapting the export status of inherited features", page====, On P2, see the informal discussion on "typing and redeclaration", page====, and the Redeclaration rule, page====, clause 2.

P1 • A class may override the export policies of its parents; it may for example make secret its version of a feature which the parent exported.

P2 • A routine redefinition may replace the type of a formal argument by a type conforming to the original. This is known as the **covariant** argument typing policy.

Although they may seem surprising at first, properties P1 and P2 are important aspects of the typing policy. The rationale is discussed in detail below. First, we must understand why they may have unpleasant consequences if we limit ourselves to class-level validity checking.

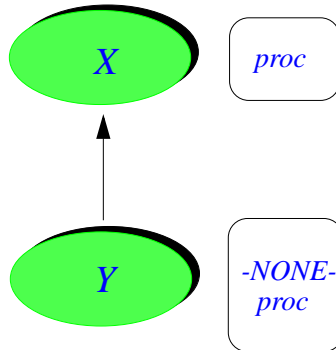
25.7, page 672 below, explains why these properties are essential for realistic uses of object-oriented concepts.

25.6 VIOLATING SYSTEM VALIDITY



(If this is your first reading, you may be content with the realization that type checking is less trivial than it appears at first, and that a systemwide type checker will detect the non-trivial errors. You may then want to skip the rest of this chapter.)

It is indeed not hard to put together an example where P1 prevents a class-valid system from working properly. Similar examples relying on P2 would be almost as immediate.



Hiding an inherited feature

Consider a class *X* which exports a procedure *proc* without arguments, and its heir *Y* which makes *proc* secret, as shown by the above figure. To hide *proc*, class *Y* will use the `New_exports` clause:

The New_exports clause of a Feature_adaptation part enables a class to override the export policies of its parents. See 7.11, page =====.

```
class Y inherit
  X
  export {NONE} proc end;
  ... Rest of class omitted ...
```

Then consider a class *C* which calls *proc* on a polymorphic entity which at run time may become attached to an object of type *Y*. This will be the case if *C* contains the following declarations and instructions, in some order:



```
a: X
b: Y           -- Y is an heir of X
create b      -- Instruction β
a := b        -- Instruction δ
a.proc
```

WARNING: *this is not a contiguous extract, just some lines which may appear anywhere in a class text in any order satisfying the constraints on declarations and instructions.*

The call *a.proc* is class-valid since *X* exports *proc*. But it is not system-valid: the instruction labeled β may attach to *b* an object of type *Y* (the static type of *b*); the instruction labeled δ may attach *a* to the same object as *b*; then the last instruction may call *proc* on that object, even though it is an instance of *Y*, and *proc* is secret in *Y*.

This example – or any similar one using P1 or P2 to violate system-level validity in the presence of polymorphism – immediately brings four important comments.

First, the example is not affected by the order of the offending instructions (β , δ and the call). As long as they appear in the same system and may all potentially be executed, the call is system-invalid. For obvious reasons of simplicity, system-level validity does **not** involve any flow analysis; even in the extreme case in which the polymorphic assignment δ would be replaced by

if *False* then $a := b$ end

we would still consider the call to be invalid.

The second comment is that system-level invalidity in such an example is a serious problem, not just a matter of style. If the author of Y did not export *proc*, we must presume that this was for a good reason. Remember in particular that an exported routine must preserve the class invariant. So *proc* preserved the invariant of X , but perhaps the invariant of Y is stronger and *proc* does not preserve it any more. In this case, applying *proc* to an object of type Y may produce an inconsistent object – one which does not satisfy the fundamental consistency constraints expressed by the invariant. This is a potential disaster.

The invariant preservation requirement is part of class consistency, page =====.

Third, neither the call $a.proc$ nor the polymorphic assignment $a := b$ is wrong by itself. The call applies an exported procedure of class X to an entity a of type X ; the assignment satisfies the type conformance rule. What is wrong is the possibility for these two individually legitimate constructs to be executed as part of an execution of the same system. To be more precise, even that combination would be harmless were it not for the presence of a third accomplice, the Creation instruction labeled β , which raises the possibility for b , and hence for a as well, to become attached to an object of type Y .

This brings the last comment, addressing a question that may well have been troubling you for some time now: isn't the type policy *wrong*? Why do we allow a class to hide some of its parent's exported features, or to replace an argument type by a more specific (conforming) one? Shouldn't we have a stricter policy, guaranteeing that class-level validity implies system-level validity?

As it turns out, however, the type policy, although perhaps surprising at first, is essential to support the practice of object-oriented software development. Let us take a closer look at the underlying issues.

25.7 NOTES ON THE TYPE POLICY



You may indeed have wondered what all the fuss was about. Shouldn't class-level validity imply system-level validity? Then type checking would be trivial, involving only local properties of classes.

The culprits were identified above: the two properties P1 and P2, which free heirs from some of the export and typing decisions made by their parents. We should really ask ourselves whether these properties are appropriate.

Here they are again:

- A class may override the export policies of its parents; it may for example make secret its version of a feature which the parent exported.
- A routine redefinition may replace the type of a formal argument by a type conforming to the original (covariant argument policy).

A third related property is that a creation procedure of a class may not enjoy the status of creation procedure any more in a proper descendant. See the Creation rule, page =====.

Then if S has an exported routine sf of name $fname$ with a formal argument of type $SOME_TYPE$ the call used earlier as example will be class-valid. Here it is again, with some of the enclosing class text omitted:

```
target: S; other: B; -- D must be a descendant of S
y: SOME_TYPE; ...
routine
  do
    if some_test then target := other end; ...
    target.fname (y); ... Rest of routine omitted ...
```

But that call is not necessarily system-valid. D may redefine $target$ to be of some type D ; or it may make its version of sf secret; or it may redefine this routine to take an argument of type B , a proper descendant of $SOME_TYPE$. Any of these cases makes the above call system-invalid since the dynamic class set of $target$, as a result of the polymorphic assignment $target := other$, includes D .

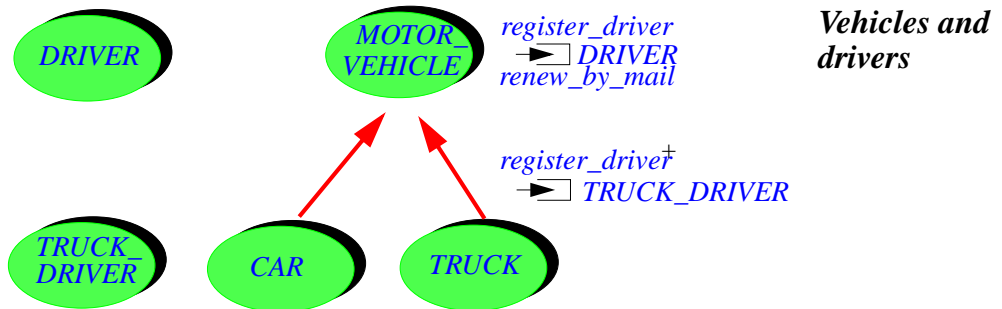
System-level checking will detect the problem and flag the system as invalid.

The above two properties (P1 and P2) often seem surprising at first. Why make type checking more difficult, and introduce the distinction between class-level and system-level validity by allowing classes to choose export and argument typing policies different from those of its parents?

The answer is that this flexibility is indispensable to the practice of object-oriented design. Without it, designers would constantly have to reshuffle inheritance hierarchies, and would have much difficulty observing the constraints of a typed object-oriented language. P1 and P2 serve to acknowledge the inescapable difficulty of reconciling the goals of orderly classification (as implemented through inheritance) and safety (as implemented through typing) with the irregularities and instability of the real-world situations which our software systems attempt to model through their inheritance hierarchies.



Although a full discussion of this question falls beyond the scope of this book, a simple example will serve to illustrate the need for properties P1 and P2.



Assume the two inheritance hierarchies represented above, with *MOTOR_VEHICLE* having heirs *CAR* and *TRUCK*, and *DRIVER* having *TRUCK_DRIVER* as heir. These classes could be part of the system used by a company to manage its fleet of vehicles, or by a Department of Motor Vehicles to keep track of driver registration.

To begin, this raises an obvious case of P2 (covariant argument type redefinition). Class *MOTOR_VEHICLE* has a procedure

```
register_driver (d: DRIVER) is...
```

which naturally takes an argument of type *DRIVER*. For trucks, however, the driver must be approved for truck driving; accordingly, class *TRUCK* redefines *register_driver* to take an argument of type *TRUCK_DRIVER*.

The type constraints in such a case permit the above inheritance structure and the redefinition of *register_driver* – a case of possibility P2. They even permit such polymorphic assignments as the one in

```
a: MOTOR_VEHICLE; t: TRUCK;
...
a := t
```

or Creation instructions such as *!TRUCK !a ...*, with the same declarations. What system-level validity will reject is the only case that could lead to an erroneous call at run time: the presence in the same system of a polymorphic reattachment such as the above and a Call such as

```
a.register_driver (dr1)
```


where *drv* is of type *DRIVER* (not *TRUCK_DRIVER*). Clearly, the presence of this Call in a system that may also attach an instance of *TRUCK* to *a* is erroneous, and will be flagged as invalid. This, however, does not affect the need for P2-like covariant argument redefinition; in fact, the system-level validity rule is what makes P2 possible.

Examples of this kind, with two parallel inheritance hierarchies, are a constant occurrence in the development of systems and their class hierarchies. Many appear in the Data Structure Library. For example, to describe doubly linked lists, *TWO_WAY_LIST* inherits from *LINKED_LIST*; to describe two-way chained linked cells, *BI_LINKABLE* inherits from *LINKABLE*. The list classes have procedures manipulating list cells, such as *put_linkable_left*, which quite naturally take arguments of type *LINKABLE* in *LINKED_LIST* and *BI_LINKABLE* in *TWO_WAY_LIST*. *LINKED_LIST* and its use of *LINKABLE* and 'put_linkable_left' are sketched in 5.5, page 134.

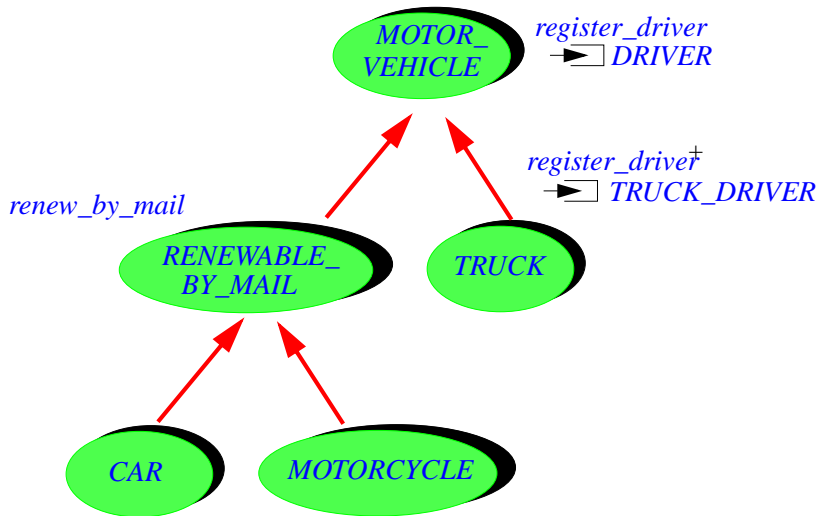
In this case, however, there is no explicit redefinition such as that of *register_driver* in *TRUCK*. The reason is the presence of the Anchored form of type declaration. Class *LINKED_LIST* contains the declarations

first_element: *LINKABLE* [**like first**]
put_linkable_left (*new*: **like first_element**) **is..**

so that *TWO_WAY_LIST* only needs to redefine *first_element* (to be of a type based on *BI_LINKABLE*); the argument *new* of *put_linkable_left*, and all the other entities declared **like first_element** in *LINKED_LIST*, follow automatically. As this example shows, the whole idea of anchored declarations is based on the principle of covariant argument redefinition.

The example of motor vehicles, trucks and drivers may also provide an example of the need for policy P1 (independence of heirs' and parents' export policies. Assume the permits for motor vehicles can normally be renewed by mail, hence the presence in *MOTOR_VEHICLE* of an exported procedure *renew_by_mail*. For trucks, however, this does not apply (the truck must be inspected for safety, every year at the time of re-registration). So *TRUCK* does not export *renew_by_mail* (which might violate an invariant of this class, although it preserves the invariant of *MOTOR_VEHICLE*).

In a case like this, one may always argue that the inheritance hierarchy was improperly designed, and should have separated renewable-by-mail vehicles from others, with *renew_by_mail* introduced not in class *MOTOR_VEHICLE* but one level down:



Not all registrations may be renewed by mail

But forcing this as the only acceptable choice would make the practice of object-oriented software development almost impossible.

In any practical problem, there will be many possible criteria for classification; what will happen if, after you have taken apart the original hierarchy because of the registration-by-mail problem, you must take into account other, independent criteria? For example, some vehicles will be for personal use and others for professional use; some will have two wheels and others will have more; some will pay a road tax and some will not; some will require smog inspections every three years; and so on. Since the original designers could not, without perfect foresight, have come up with the ideal inheritance hierarchy, the developers will find themselves constantly redoing the structure. The conflicting criteria may in fact make it impossible to obtain any acceptable inheritance structure at all.

The flexibility of policy P1 makes it possible to handle this problem by allowing a class to be selective about its inheritance – exporting or hiding inherited features to its own clients according to its own local properties. As before, this is an example of transferring part of the burden from developers (in the form of constant architecture redesign) to the supporting implementation (in the form of the more sophisticated form of type checking required by system-level validity).



It should be clear from this discussion that a well-designed inheritance hierarchy will include few occurrences of classes hiding some of their parent's features. If you find yourself constantly at odds with the parent designers' decisions, then you should probably consider improving the inheritance structure (assuming you are permitted to do so). This is why the default policy for inherited features is to retain the parents' export status; to override it, you must include an explicit `New_exports` clause. But the ability to do this in the minority of cases which call for it is a key component in the effort to make object-oriented software construction not just a pleasant theoretical idea but a practical way to produce real systems.

25.8 WHY DISTINGUISH?

If we accept that system-level validity is the appropriate notion of validity, it is fair to ask why one should bother at all with class-level validity. Why not have a single validity condition, as for the other constructs studied in this book?

The reason is pragmatic, and involves two complementary observations on possible violations of the validity constraints:

- First, it is easier (for a language processing tool as well as a human reader) to detect violations of class-level validity, since they only involve a local analysis of the features one class – S , the base class of *target*'s type. In contrast, checking system-level validity may involve systemwide analysis. The names "class-level" and "system-level" reflect this difference.
- Second, a study of errors as they occur in actual system development reveals that system-level validity violations which are not also class-level violations occur very rarely.

For these reasons, implementors may choose to design class-level and system-level checking as separate facilities. Class-level checking will detect a vast majority of errors; the remaining ones will be found by applying system-level checking.

Of course, to guarantee fully the type safety of a system, you must check both kinds.

Class-level checking is straightforward. The only non-trivial part of system-level validity is to determine the dynamic class set. Let us see concretely how this can be done.

25.9 A LOOK AT THE DYNAMIC CLASS SET

The dynamic class set of an entity is the set of base classes of all types that the entity may take on at run-time, as a result of polymorphic reattachments and creation instructions.

The call validity rule, appearing in the precise discussion at the end of this chapter, will give a full definition. It is important, however, to get first an intuitive view of what the base class represents. (Although "intuitive" this view is not incorrect; it simply misses some details and does not cover all cases.)

The idea will be to determine in a single process the dynamic class sets of *all* entities. The process is iterative; if you have a background in numerical mathematics, it will remind you of algorithms which compute the solution to a vector or matrix equation by successive iteration (for example over a grid); if you are familiar with the theory of programming languages, it will remind you of fixpoint methods for approximating the high-level functions and domains of denotational semantics.

Fixpoint methods for denotational semantics are covered in "Introduction to the Theory of Programming Languages". See bibliography.

An example will serve to illustrate the process. Consider a class extract containing the following four instructions, in some order:



create {X} <i>a</i>	-- α
create {Y} <i>b</i>	-- β
<i>c</i> := <i>a</i>	-- γ
<i>a</i> := <i>b</i>	-- δ
<i>a.proc</i>	

This is a variant of the example on page ====.

Each instruction has been identified by a Greek letter. The context is missing, in particular the declarations; the Creation instructions have an explicit creation type for clarity, although α , for example, could appear as just **create** *a* if *a* is of type *X*.

As before, the order of these instructions is irrelevant. If they appear in the same context, then *a*, as discussed above, may become attached to an object of type *Y*; this means that, for system validity, any routine *rou*t appearing in a call *a.rou*t using *a* must meet the appropriate conditions not just for *X* but also for *Y*.

System-level validity analysis will need to determine the base class sets of *a*, *b* and *c*. The result, obtained through a process explained below, will be the following:

The complete form of this result, given page ====, will include more information, in particular references to iteration steps.

a $X[\alpha] ; Y[\delta]$

b $Y[\beta]$

c $X[\gamma] ; Y[\gamma]$

This shows a vector of dynamic class sets, one for each entity. Each class set contains a list of types. Any type which appears in one of these lists is there because of one of the instructions; to make this justification clear, the instruction's identifying greek letter appears in brackets next to the type.

For b , the class set includes just Y , resulting from the Creation instruction β . For a , it includes X , resulting from the Creation instruction α , and Y , resulting from the polymorphic assignment δ which adds all of b 's class set to the class set of a . For c , assignment γ means that the class set is the same as that of a .

In the rest of this section "class set" is an abbreviation for "dynamic class set".

How do we determine this result? An iterative process will provide the solution. Starting from the types given by Creation instructions, we may repeatedly extend the current sets by adding to the class set of every entity e the class set of any other entity f such that there is a reattachment of f to e somewhere. We stop when we have reached a "fixpoint", that is to say when our vector of class sets is stable.

Here is this process applied to the above example. To obtain the initial vector $v0$ of class sets, just look at the creation instructions:

$$\begin{array}{l}
 a0 \quad \boxed{X [\alpha]} \\
 b0 \quad \boxed{Y [\beta]} \\
 c0 \quad \boxed{}
 \end{array}$$

For each type appearing in a class set, a comment in brackets identifies the instruction which causes the class to be there: the Creation instruction α puts X in the class set of a , and β puts Y in the class set of b . Entity c is not the target of any Creation, so its class set is empty for the moment.

On each iteration, we will look at every reattachment and extend the target's class set with all the classes obtained so far in the source's class set. For the first iteration, this gives the new class set vector $v1$:

$$a1 \quad \boxed{X [\alpha] ; Y [\delta : b0]}$$

$$b1 \quad \boxed{Y [\beta]}$$

$$c1 \quad \boxed{X [\gamma : a0]}$$

The class set of a now contains Y , again identified, for clarity, by its origin: the comment $[\delta : \sim b0]$ means that Y comes from $b0$, the earlier class set for b , and has been added to $a1$, the new class set of a , because of the polymorphic assignment $a := b (\delta)$. In the same way, X now appears in the class set of c because of its presence in $a0$ and of the assignment $c := a (\gamma)$. You may be tempted to add Y , which appears in $a1$, but this would be cheating: to update a vector at any step, we may only use vector elements from the previous step.

The next step, producing vector $v2$, will indeed use $a1$ and γ to add Y to the class set of c :

$$a2 \quad \boxed{X [\alpha] ; Y [\delta : b0]}$$

$$b2 \quad \boxed{Y [\beta]}$$

$$c2 \quad \boxed{X [\gamma : a0] ; Y [\gamma : a1]}$$

If you apply the mechanism once more, you will find that it does not bring anything new: $v3$ is the same as $v2$. We have put all the available type information to good use; $v2$ gives the complete class sets for all entities involved. (In technical terms $v2$ is a fixpoint.)

The process as illustrated on this example is not hard to generalize to the full language. The extension must integrate expressions which are function calls rather than simple entities; it must also account for the two other forms of reattachment beside Assignment: actual-formal association, which raises no particular problem, and Assignment_attempt. For this last case, the effect of

$$b: Y,$$

$$\dots$$

$$b \text{ ?} = a$$

to extend b 's class set not with all elements of a 's class set (as with normal Assignment), but only with those which are descendants of Y .



This discussion has outlined a way of obtaining the dynamic class sets of the entities in a system. Two words of warning will serve as its conclusion:

The precise specification of dynamic class sets appears as part of the call validity rule below. The iterative process that we have just discussed is only one concrete interpretation of that specification, although of course it satisfies that specification.

Although you may view the description of that process as an abstract algorithm for language processing tools that perform type checking, its purpose is explanatory only. Implementors of compilers and other type checking tools may well rely on totally different methods.

By now you should have a good understanding of the practical implications of type checking. All that remains is to give the rules in their full and precise form.

25.10 THE CALL VALIDITY RULE



(This last section formalizes the previous discussion of validity, but does not introduce any new concepts, so that you may safely skip to the next chapter. In fact this section will be mostly of interest to implementors of language processing tools.)



General Call rule

VUGC

A call is **valid** if and only if it is both class-valid and system-valid.

This is very general and means that we must now define class-level and system-level validity.

To remove any ambiguity, we must provide an equally precise definition of the **dynamic class set** of an expression. This is the set of base classes of all elements in the **dynamic type set** of the expression; the dynamic type set was itself defined as the set of all possible dynamic types of the expression, where a possible dynamic type for an expression is the type of any object to which it may become attached at run time. This definition is correct, but it does not enable us to determine easily the dynamic type set, or the dynamic class set, from the software text.

Since this general validity rule is very abstract, the successive definitions of this section, introducing the different aspects of call validity, are in 22.11; see page 681.

The above informal illustration constructed dynamic class sets through successive vector approximations, until it reached a fixpoint. Since it assumed all classes to be non-generic, its results were both dynamic classes and dynamic types. The full definition, which covers anchored and generically derived types, will yield the the dynamic *type* sets; to obtain the corresponding class sets, just replace every type by its base class.

The iterative process was described in [25.9](#), page 677.

The definition needs the following two notions to deal with genericity. Let T be a Class_type based on a class C . If C is a generic class $C [GI, \dots]$, T is $C .[AI, \dots]$ for some types AI, \dots ; if C is not generic, T is just C . Then:

- If e is an entity or expression appearing in a feature of C , the "dynamic type set of e for T " is the set of dynamic types of objects that may become attached to e as a result of calls to C 's features on direct instances of T . The "dynamic type set of e ", with no further qualification, is the dynamic type set of e for $C .[GI, \dots]$, or just C if C is not generic.
- If U is a type, the notation UT will stand for the type obtained from U by substituting Ai for any occurrence of Gi – or just U if C is not generic. For example, if U is $X .[G, , INTEGER]$ appearing in a feature of class $D .[G]$, and T is $D .[REAL]$, then UT is $X .[REAL, , INTEGER]$.

Here is the full definition of dynamic type sets:



Dynamic type set

The dynamic type set of an expression e is the set of types of all objects that can become attached to e during execution.

- 1 • The dynamic type sets of the expressions, entities and functions of a system result from performing all possible applications of the following rules to every **Class_type** T , of base class C , used in the system.
- 2 • If a routine of C contains a creation instruction, with target x and creation type U , the dynamic type set of x for T is $\{U_T\}$.
- 3 • The dynamic type set for T of an occurrence of **Current** in the text of a routine of C is $\{T\}$.
- 4 • For any entity or expression e of expanded type appearing in the text of C , if the type ET of e is expanded, the dynamic type set of e for T is $\{ET_T\}$. (Rules 4 to 7, when used to determine elements of the dynamic type set of some e , assume that e 's type is not expanded.)
- 5 • If a routine of C contains an **Assignment** of target x and source e , the dynamic type set of x for T includes (recursively) every member of the dynamic type set of e for T .
- 6 • If a routine of C contains an **Assignment_attempt** of target x , with type U , and source e , the dynamic type set of x for T includes (recursively) every type conforming to U_T which is also a member of the dynamic type set of e for T .
- 7 • If a routine of C contains a call h of target ta , U is (recursively) a member of the dynamic type set of ta for T , and tf is the version of the call's feature in the base class of U , then the dynamic type set for U of any formal argument of tf includes every member of the dynamic type set for U_T of the corresponding actual argument in h .
- 8 • If h , tf and U are as in case 6 and tf is an attribute or function, the dynamic type set of h for T includes (recursively) every member of the dynamic type set for U_T of the **Result** entity in tf .



Each of the seven cases of this definition, explained in detail below, is a rule which you may use to bring new elements to the possible dynamic type sets of the expressions, entities and functions of a system. More precisely:

- Rules 1, 2 and 3 are non-recursive: they yield elements of the dynamic type sets without further ado.

- Rules 3 to 7 are recursive: given known elements of the dynamic type sets of the expressions of a system, they may add new elements.

In other words, you may view the definition as describing an iterative process, generalized from the earlier discussion, for building the dynamic type sets: first apply rules 1 to 3 to every possible case, obtaining v_0 , the initial vector of dynamic type sets; then, at each successive step i , apply rules 3 to 7 to every possible case, obtaining new elements of the type sets in vi from elements of the type sets in $vi - 1$. The process terminates if the resulting vi is the same as $vi - 1$ – that is to say, the last iteration has brought nothing new.

This process is finite since the set of types in the system is finite. To get an upper bound to the number of iterations, call DEPTH be the maximum depth of a call (number of dots in Call form, number of operators in Operator_expression form) and ATTACH the maximum length of a non-cyclic sequence ei such that there is a reattachment from $ei + 1$ to ei ; then the process will terminate in at most DEPTH + ATTACH steps.

Let us now make sure we understand the seven rules. Rule 1 addresses creation instructions. It considers that an instruction of the form



$! U ! x$

adds the creation type, here U , to the dynamic type set of x . (If U is absent, the creation type is x 's base type.) If C is generic, the rule pertains to the dynamic type sets relative to some generic derivation T of C ; then we must perform the corresponding substitution of actual for formal generics, so the rule uses UT rather than just U .

Creation instructions were discussed in 20.11, with the definition of "creation type" appearing on page =====.

Rule 2 indicates that *Current*, when used in C , represents an object of type T – that is to say C , with the requested generic derivation if applicable.

Rules 1 and 2 reflect the pragmatism of system-level validity checking. Class-level checking considers the developer's intentions (the declarations); but system-level checking only considers deeds: the types of the objects that Creation and reattachment instructions may actually attach to entities.

Rule 3 takes care of expressions of expanded types, which are never polymorphic. In this case we just take the declarations at face value.

See 14.9, page =====, about conformance for basic types. Assignment semantics in this case ([4] on the table page 317) is copy, implying conversion to the "heavier" type (case 2 of copy semantics, page =====).

As you may remember, the conformance rule for expanded types allows for some tolerance in the case of basic arithmetic types. For example you may attach the integer value 3 to an entity r of type *REAL*. But this has no effect on the dynamic type set of r : such an assignment causes a conversion, and attaches to r a real value, here 3.0. So there is no polymorphism in this case.

Rules 4, 5 and 6 covers the three forms of possibly polymorphic reattachment: Assignment, Assignment_attempt, and actual-formal association in a call. For a reattachment of a value e to an entity x , we must add all of e 's possible dynamic types to those of x . In addition:

- For an Assignment_attempt of the form $x \text{ ?} = e$ (rule 5), we must only consider those possible types for e which conform to the type of x : any other one would result, in accordance with the semantics of Assignment_attempt, in no object attachment for ta .
- For a call (rule 6), e is an actual argument and x is the corresponding formal argument in the appropriate version of the routine.

In rule 6, a member U of the dynamic type set of ta , the call's target, may be generically derived; then when need to perform the corresponding type substitutions in adding the members of the actual argument's dynamic type set to the dynamic type set of the formal argument. This is why the rule considers the dynamic type set of e (the actual argument) for UT .

A call whose feature is a function is itself an expression, with its own dynamic type set. Since the expression's value is the final value of *Result* as used in the function, rule 7 defines the dynamic type set of the expression as that of *Result*. As with rule 6, we must perform the appropriate substitution if the type of the call's target is generic, hence the use of UT .

We need one more convention to make the above rule fully applicable in practice: how to handle the dynamic type sets of array elements. The relevant features in the Kernel Library class *ARRAY* are *put* and *force*, which set an element's value, and *item*, which returns an element's value, as in

```
x, some_entity: T; i, j: INTEGER; a: ARRAY [T];
...
    -- Assign the value of some_entity to the i-th element of a:
a.put (some_entity, i)
...
    -- Assign to x the value of the j-th element of a:
x := a.item (j)
```

*Arrays require a specific convention since the Kernel Library specification covers the interface of class *ARRAY* (A.6.19, page 996) but not its implementation.*

'put' assumes that the index, 'i' in the example, is within bounds; 'force' resizes the array if necessary. Details in chapter 28.

To find out the dynamic type set of *a.item* (*j*) (and hence of *x*), note that the software text usually does not suffice to determine whether *i* and *j* will have the same value at execution time. So we must treat every *put* or *force* operation as affecting potentially every array element. Hence the rule:



Array type rule

To study the effect of array manipulations on dynamic type sets, assume that in class *ARRAY* feature *item* is an attribute, and that *put* (*v*, *i*) and *force* (*v*, *i*) are both implemented as

```
item := v
```

The rule also applies to manifest arrays. A manifest array is an expression of the form `<<a1 ,, ... ,, an>>`, denoting an array of *n* elements, containing the values given. For typing purposes, it will be treated as it had been initialized explicitly by *n* calls to *put*, each of the form

See [29.9, page 809](#), about manifest arrays.

```
a.put (ai, i)
```

25.11 CREATION VALIDITY (SYSTEM-LEVEL)



(This section explores a specialized type-checking issue and may be skipped at first reading. Even if you want all the details, you will probably have to come back after you have read the chapter on type checking, which is necessary for a full understanding of this discussion.)

Chapter [25](#) explains the type checking policy, with special emphasis on calls.

Although class-level validity generally suffices to determine the validity of a **Creation**, the complete definition will require system-level validity as well.

For our immediate purposes it suffices to note that some invalid cases may escape class-level validity checks. The reason is polymorphism. As a result of assignments of the form

Polymorphism is studied in [22.11, page 606](#).

```
x := y
```

a **Variable** entity *x* of type *T* may become attached to objects of *y*'s type (which the Assignment rule requires to be a descendant of *T*). These types, for all possible *y*, make up the set of all **possible dynamic types** of *x*, also called its **dynamic type set**. The dynamic type set may contain types other than *x*'s static type, *T*.

The rigorous definition of the dynamic type set is on page [683](#).

But a **Creation_instruction** involving *x*

```
create x.make (...)
```

may be invalid even if it is class-valid. This will be the case, for example, if in the above assignment y is of a type U based on a class D (a proper descendant of T 's base class), and D fails to list its version of *make* as a creation procedure.

System-level validity avoids any such problem:



Creation System-Validity rule VGCS

A **Creation_instruction** is **system-valid** if and only if it satisfies one of the following two conditions:

- C1•The creation type is explicit (in other words, the instruction begins with **create** $\{ET\}$... for some type T).
- C2•The creation type is implicit (in other words, the instruction begins with **create**...) and every possible dynamic type T for x , with base class C , satisfies conditions 1 to 6 of the Creation Instruction rule (page \n(9g). In applying conditions 5 and 6, the feature of the call, f , must be replaced by its version in C .

In other words, system-level validity is the same property as its class-level counterpart, but applied to all possible dynamic types of the target x . In interpreting conditions on the creation procedure f , we must take into account the dynamic binding version of that procedure in a descendant class, which may be different from the original because of redeclaration, and may have a different name because of renaming.

← “*Dynamic binding version*”, page 468.



As condition 1 of the definition indicates, the problem of system-level validity only arises for **Creation** instructions with an implicit type. If the type is explicit, as in **create** $\{T\}$ x ..., the possible dynamic types of x do not affect the validity of the instruction, which in this case is entirely covered by class-level validity.

System-level validity, as all other validity properties, is a **static** requirement, which a human reader or language processing tool may ascertain simply by looking at the software text. Checking it does not require any control flow analysis: whenever a given context contains both an assignment $x := y$ and a **Creation** with target x which would be invalid for y 's type, the **Creation** will be system-invalid – even if clever control flow analysis would in fact show that no control flow path will ever execute the assignment and the **Creation** in sequence. Static validity checking doesn't need to be clever; it needs to be safe. This discussion will be generalized to calls in the discussion of type checking.

To be valid, a **Creation** must satisfy the requirements at both levels:



Creation Instruction rule

VGCI

A **Creation_instruction** is valid if and only if it is both class-valid and system-valid.

Exception handling

26.1 OVERVIEW

During the execution of an Eiffel system, various abnormal events may occur. A hardware or operating system component may be unable to do its job; an arithmetic operation may result in overflow; an improperly written software element may produce an unacceptable outcome.

Such events will usually trigger a signal, or **exception**, which interrupts the normal flow of execution. If the system’s text does not include any provision for the exception, execution will terminate. The system may, however, be programmed so as to *handle* exceptions, which means that it will respond by executing specified actions and, if possible, resuming execution after correcting the cause of the exception.

This chapter presents the exception mechanism by explaining what conditions lead to exceptions, and how systems can be written so as to handle exceptions.

---- REWRITE It also introduces the *EXCEPTION* Kernel Library class and some of its descendants, which provides tools for fine-tuning the exception mechanism.



When using the exception facility, remember to take its name literally. The constructs discussed in this chapter — Rescue clause, Retry instruction — are not control structures on a par with those of the previous chapter; they should be reserved for those unexpected cases which cannot be detected a priori. Complex algorithmic structures, if any, should appear in *Feature_body* parts, not in exception handlers. If your system has many sophisticated exception handling clauses, it is probably misusing the mechanism.

26.2 WHAT IS AN EXCEPTION?

DEFINITION

Failure, exception, trigger

Under certain circumstances, the execution or evaluation of a construct specimen may be unable to proceed as defined by the construct's semantics. It is then said to result in a **failure**.

If, during the execution of a feature, the execution of one of its components fails, this prevents continuing its execution normally; such an event is said to **trigger** an **exception**.

"Failure" is in fact the more primitive notion; an exception is the consequence of a failure.

See chapter 9 about assertions.

See below about routine failure.

Examples of exception causes include:

- Assertion violation (in an assertion monitoring mode).
- Failure of a called routine.
- Impossible operation, such as a **Creation** instruction attempted when not enough memory is available, or an arithmetic operation which would cause an overflow or underflow in the platform's number system.
- Interruption signal sent by the machine for example after a user has hit the "break" key or the window of the current process has been resized.
- An exception explicitly raised by the software itself.

"Machine" means hardware combined with operating system. See 2.12, page====.

Common exception types that do *not* arise in Eiffel, other than through mistakes in the definition of the language as specified by the Standard, are "void calls" (attempts to execute a feature on a void target) and "catcalls" (attempt to execute a feature on an unsuitable object).

The software may raise an exception through procedure 'raise' in class EXCEPTIONS. See 5.11, page====below.

These categories distinguish the manifestation of the exception, not its real cause. Causes of exceptions essentially boil down to two possibilities: an error (a bug) in the software, or the inability of the underlying machine to carry out a certain operation. Assertion violations are a clear example of the first cause – a correct program always satisfies its assertions at run time – whereas running out of memory for a **Creation** is an example of the second.

In a way, the second of these types of cause is a variant of the first: if systems never executed an operation without checking first that it is feasible, then a correct system would never run into an exception. But it would be hardly practical to have every **Creation** instruction preceded by a check for available space, or every addition preceded by a check that the result will fit in the machine's number system – assuming such checks were possible.

*In an environment supporting virtual memory and a garbage collector, unsuccessful **Creation** only occurs when the system has exhausted virtual memory and the collector is unable to reclaim any space. See 20.16, page====.*

In cases like these, a priori checking is expensive, and only a small percentage of executions are likely not to pass the checks. These are the cases requiring exceptions – ways to detect an abnormal situation, and possibly recover from it, *after* it has occurred.

26.3 EXCEPTION HANDLING POLICY

What can happen after an exception? In other words, what can we do when the unexpected occurs?



To answer this question properly, we must remember that a routine or other software component is not just the description of some computation. What transcends that particular computation is the goal that it is meant to achieve – what in the Eiffel theory is called the **contract**. The component provides just one way to achieve the contract; often, other implementations are possible. For simple components the contract is defined by the language: for example the contract of a Creation instruction is to create an object, initialize its fields and attach it to an entity. For more complex components you may express the contract through assertions: for example, a routine's contract may be defined by a precondition, a postcondition, and the class invariant. Even if there are no explicit assertions, the contract implicitly exists, perhaps expressed informally by the routine's Header_comment.

See "Object-Oriented Software Construction" for more in-depth discussions of exception handling principles. References in appendix C.

If we want to remain in control of what our software does, we must concentrate on the notion of contract to define possible responses to an exception. The contract of a software component defines the observable aspects of its behavior, those which its clients expect. Any exception handling policy must be compatible with that expectation.

An exception is the occurrence of an event which prevents a component from fulfilling the current execution of its contract. An unacceptable reaction would be to terminate the component's execution and to return silently to the client, which would then proceed on the wrong assumption that everything is normal. Since things are *not* normal – the client's expectations were not fulfilled – such a policy would almost inevitably lead to disaster in the client's execution.

What then is an acceptable reaction? Depending on the context, only three possibilities make sense for handling an exception:

- A favorable albeit unlikely case is one in which the exception was in fact not justified. This is called the **false alarm**.
- When writing the component, you may have anticipated the possibility of an exception, and provided for an alternative way to fulfil the contract. Then the execution will try that alternative. This case is called **resumption**.
- If you have no way of fulfilling the contract, then you should try to return the objects involved into an acceptable state, and signal your failure to the client. This is called **organized panic**.

The language mechanism described below – Rescue clauses and Retry instructions – directly supports resumption and organized panic. The rather infrequent case of false alarm is handled through features of the Kernel Library class *EXCEPTIONS*.

These mechanisms are defined at the routine level. For components at a lower level, such as an instruction or a call, you have no language mechanism to specify potential recovery. This means that for an unsuccessful attempt at executing such a component (for example an attempt at object creation when there is not enough memory, or at feature call on a void target) only policy E3 is possible: the component's execution will fail immediately, causing an exception. The exception interrupts the last started routine, called the *recipient* of the exception--- NOT TRUE, SEE FOLLOWING RULES, REMOVE

Depending on the recipient routine and its class, the exception will be handled through one of the three techniques listed

: above. --- NOT TRUE, SEE FOLLOWING RULES, REMOVE



Recipient of an exception

The **recipient** of an exception is the current routine at the time of the exception.

For the rest of this chapter, then, the unit of discourse is the routine. Any exception has a recipient, which is a routine. By writing an appropriate Rescue clause, you may specify the routine's response as resumption or organized panic; through the appropriate calls to library features, you may in some cases proceed with the routine's execution after a false alarm.

The next sections explain how to specify one of these three possibilities as your choice for exception handling.

26.4 RESCUE CLAUSES AND ORGANIZED PANIC

The construct which specifies a routine's response to exceptions that may occur during an execution of the routine is the Rescue clause.

This is an optional part of a Routine declaration, introduced by the keyword **rescue**.

Here is a sketch of a routine with a Rescue clause:



```

attempt_transaction (arg: CONTEXT)
  --Try transaction with arg; if impossible,
  --reset current object
require
  ...
do
  ...
ensure
  ...
rescue
  reset (arg)
end

```

Any exception triggered during the execution of the **Feature_body** (**do...** clause) will cause execution of the Rescue clause. Here this clause calls procedure *reset*, meant to restore the current object to a stable state; such a state should satisfy the class invariant.

Termination of the Rescue clause also terminates the routine execution; in this case, however, as opposed to what would happen if the **do..** clause was executed to the end with no exception, the call to *attempt_transaction* will fail. This is indeed the only way for a routine call to fail: being the recipient of an exception and executing its Rescue clause to the end, not ending with a Retry instruction (described below).

In other words, the routine illustrates the policy defined above as organized panic – put back the object in an acceptable state (satisfying the invariant) and terminate, notifying your caller, if any, of the failure. The technique used for this notification is to trigger a new exception, with the caller as recipient.

As noted, organized panic should restore the invariant. The formal version of this requirement, given below as part of the definition of exception correctness, is that any branch of a Rescue clause not terminating with a Retry should yield a state satisfying the invariant, independently of the state in which it is triggered.



As you may remember from the definition of class consistency, this requirement of ensuring the invariant also applies in another context: creation procedures of a class. This suggests that it is sometimes possible to write a Rescue clause as a call to a creation procedure, which will reset the object to a state which it could have reached just after creation. Of course, other situations may require more specific Rescue clauses, taking into account the routine that failed and the context of the failure.

Class consistency is defined on page =====.

26.5 THE DEFAULT RESCUE

In most systems, the vast majority of routines will not have an explicit Rescue clause. What happens if an exception is triggered during the execution of such a routine?

The convention a routine of a class *C* is considered, if it has no explicit Rescue clause, to have an implicit Rescue of the form ‘

```
rescue
  def_sec
```

where *def_resc* is the version of *default_rescue* in the enclosing class. Procedure *default_rescue* is introduced in the universal class ANY, where it is defined so as to have no effect:

```
default_rescue
do
end
```

Any developer-defined class, which is automatically a descendant of *ANY*, may redefine this routine to serve in case of organized panic. The redefined version will be called by any routine of the class which does not have a specific Rescue clause. Like any other routine, the redefined version is passed on to every heir, which will use it as default Rescue clause unless there is a new redefinition in the heir.

The reason for using the name *def_resc* rather than *default_rescue* in expressing the above equivalence is that in the process of inheriting from *ANY*, directly or indirectly, classes may rename features. For clarity, however, it is recommended to keep the original name *default_rescue*.

← “ANY”, 6.5, page 172; see also chapter 35 for more details.

The “version” of a routine in a descendant of its class of origin is the result of any redefinition and renaming that may have occurred along the inheritance path; see 11.12, page

====

If, following the possibility suggested above, you use a creation procedure as default Rescue, you may rely on the following scheme, where *default_rescue* and the creation procedure are declared as synonym features:

It is also possible to undefine 'default_rescue' and rename it as 'make'. This, however, would lose the original name. the end of 6.12.



```

class C create
  make, ... other creation procedures if any ...
inherit
  ANY
  redefine default_rescue end
...
feature
  make, default_rescue
  -- No precondition
  do
    ... Appropriate implementation;
    ... must ensure the invariant.
  end
. Other features ...
end

```

With this scheme, since *default_rescue* has no argument, there must also be no argument for the creation procedure chosen as synonym, here *make*.

The *default_rescue* convention explains what happens if a routine such as *attempt_transaction* above fails and its caller had no explicit Rescue. The caller will simply execute its version of *default_rescue* – which means doing nothing at all if it still has the original version inherited from *ANY*. Then it will fail and trigger an exception in its client, which will itself be faced with the same situation. The effect of executing this Rescue chain all the way to the original root call will be described below.

'attempt_transaction' was on page =====.

26.6 RETRY INSTRUCTIONS AND RESUMPTION

Sometimes you can do better than just conceding defeat and cutting your losses. This is where the Retry instruction is useful.

This instruction, which supports the resumption policy, may only appear in a Rescue clause. It has a very simple form, being just the keyword

```

retry

```

The effect of a Retry is to execute again the body of the routine. A Rescue clause which executes a Retry escapes failure – perhaps only temporarily, of course, since the body may again cause an exception.

Here is a general scheme that covers many uses of Retry. To solve a problem, you normally use method 1; if that method does not work, however, it may trigger an exception, and method 2 may yield the desired result.



```

try_once_or_twice
  -- Solve problem using method 1 or, if unsuccessful, method 2
  local
    already_tried: BOOLEAN
  do
    if not already_tried then
      method_1
    else
      method_2
    end
  rescue
    if not already_tried then
      already_tried := true
    retry
  end
end

```

This example relies on the default initialization rules for local variables: *already_tried*, being of type *BOOLEAN*, is initialized to **false** on routine entry. This initialization is not repeated if the rescue block executes a Retry.

If *method_2* triggers an exception, that is to say if both methods have failed, the Rescue clause will execute an empty Compound (since the Conditional has no Else_part). So the routine execution will fail, triggering an exception in the caller. This is because *try_once_or_twice* had two methods to reach a goal, and neither succeeded.

Local variable initialization is specified in the discussion of call semantics, "PRECISE CALL SEMANTICS", 23.17, page 652.

You may of course prefer a routine that behaves less dramatically when it cannot produce a result. Rather than sending an exception to the caller, it will just record the result in a boolean attribute *impossible* of the enclosing class: .



```

try_once_or_twice
-- Solve problem using method 1 or, if
  unsuccessful, method 2
  --if unsuccessful, method 2. Set
  impossible to true
  --if neither method succeeded,
  false otherwise.
local
already_tried: BOOLEAN
do
if not already_tried then
  method_1
elseif not impossible then
  method_2
end
rescue
if not already_tried then
  impossible := true;
  end;
  already_tried := true;
retry
end

```



This routine will never fail, since its Rescue clause always terminates with a Retry. This is not a paradox: the contract here is simply broader. As opposed to the contract for *try_once_or_twice*, it does not require the routine to solve the problem, but, more tolerantly, either to solve the problem (and set attribute *impossible* to true) or to set *impossible* to false if it is unable to solve the problem. Clearly, it is always possible to satisfy such a requirement; so there is no cause for failure.

You may easily generalize either version – *try_once_or_twice*, which may fail, and *try_and_record*, which never fails but sets a boolean success indicator – to try more than two alternative methods: just replace *already_tried* by a local variable *attempts* of type *INTEGER*, which will be initialized to zero.

As a special case, the resumption may in some situations simply amount to trying the same policy again. This applies when the exception was caused by an intermittent malfunction in external device, for example a busy communication line, or by an erroneous human input; by trying the line again, or outputting an error message asking the user to correct his input, you may hope to succeed. Here is the general scheme:



```

try_repeatedly_and_record
    -- Attempt to solve problem in at most Maximum trials.
local
    attempts: INTEGER
do
    if attempts <= Maximum then
        attempt_to_solve
    else
        impossible := true
    end
rescue
    attempts := attempts + 1
    ... Other corrective actions, such as outputting
    an error message ...
retry
end

```

Maximum is a constant attribute with a positive value. The integer attribute *attempts* will be initialized to zero on routine entry.

The strategy used by *try_repeatedly_and_record* derives from *try_and_record* rather than *try_once_or_more*: if unable to perform its duty, it does not fail but simply sets attribute *impossible* to true. Adapting to the other style, which causes the routine to fail and trigger an exception in its caller, is easy and is left as an exercise.

26.7 SYSTEM FAILURE AND THE EXCEPTION HISTORY TABLE

In the organized panic case, a failed execution of a routine *r* triggers an exception in the caller. But what if there is no caller?



This can occur only if the execution that fails is the "original call": the execution of the root's creation procedure which started system execution. Remember that executing a system means creating an instance of its root class and applying a creation procedure to that instance. The creation procedure usually calls other routines, which themselves execute further calls. This means that any routine execution except the original call has a caller.

The semantics of system execution was defined on page =====.

A failure of the original call produces a **system failure**. Execution of the system terminates, producing an appropriate diagnostic about the system's inability to fulfil its task.

This rule does not just apply to exceptions triggered directly by the original call – an infrequent case since root creation procedures tend in practice to perform only simple actions before creating other objects or calling other routines. The more interesting case is the failure of a routine execution deep down in the call sequence, for which all direct and indirect callers eventually fail because they are not able to apply resumption. Then the failure bubbles up the call chain until it finally causes system failure.

This scenario in fact applies to the simplest case, in which no routine of the system has a Rescue clause, and no class redefines *default_rescue*: then any exception occurring during execution will propagate to the root's creation procedure, and result in a system failure.

What happens after a system failure? As noted, the tool that handles execution (the run-time system) should produce a diagnostic. The exact form of that diagnostic is not part of the language specification. Here is the format used in one particular implementation. After a system failure, that implementation prints an error message and an **exception history table** such as the following:

This is the format used by ISE's implementation. Others may use different conventions.

ObjectClassRoutine exceptionEffect	Nature	of
<i>2FB44INTERFACEm_creation</i> <i>Feature "quasi_inverse":</i>		
<i>reference.Retry</i>	<i>Called on void</i>	
<i>2F188MATHquasi_inverse</i> <i>"positive_or_null":</i>		
<i>(from BASIC_MATH)</i> <i>Precondition violated.Fail</i>		

ObjectClassRoutine	Nature	of
<i>2F188MATHraise</i> "Negative_value": (from EXCEPTIONS)Developer exception.Fail		
<i>2F188MATHfilter</i> "Negative_value": Developer exception.Retry		
<i>2F32MATHnew_matrix</i> "enough_memory": (from BASIC_MATH)Check violated.Fail		
<i>2FB44INTERFACEsetRoutine</i> failureFail		

For an exception whose recipient was a routine *r*, during a call on an object OBJ, the first column identifies OBJ (through an internal object identifier), the second column identifies the generating class of OBJ (the base class of its type), and the "Routine" column identifies *r*. The next column indicates the nature of the exception; for developer-defined exceptions and assertion violations this includes a tag (the Assertion_tag for assertions clauses). The last column indicates the effect of the exception: resumption (appearing as *Retry*) or organized panic (appearing as *Fail*).

The table contains not just a trace of the calls that led to the final failure but also the entire history of recent exceptions. Some exceptions may have been caught and handled through resumption, only to lead to further exceptions. This is why the exception history is divided into periods, each terminated by a *Retry*. The table shows these periods separated by a double line; exceptions appear in the order in which they occurred, which is the reverse of the order of the calls.



The case illustrated on the table, which resulted from a specially contrived system meant to illustrate the various possibilities – involving exceptions of many kinds, and resumptions that trigger new exceptions – is unusual. Exception handling in well-written systems should remain simple, and as much effort as possible should go into avoiding exceptions rather than handling them a posteriori. Exception handling does play a crucial role, however, for those hard to prevent cases which, in the absence of an appropriate exception mechanism, would leave defenseless the system, its users and its developers.

26.8 SYNTAX AND VALIDITY OF THE EXCEPTION CONSTRUCTS

It is time now to look at the precise properties of the two constructs associated with exceptions: rescue clauses of routines and retry instructions.

The grammar is straightforward: I



Rescue clauses	
Rescue \triangleq	rescue Compound
Retry \triangleq	retry

A Rescue clause is part of a Routine. A Retry instruction is one of the choices for the Instruction construct.

See page ===== for the syntax of Routine and page ===== for the syntax of Instruction.

A constraint applies to Rescue clauses:



Rescue clause rule	VXRC
It is valid for an Attribute_or_routine to include a Rescue clause if and only if its Feature_body is an Attribute or an Effective_routine of the Internal form.	

VXRC

An **Internal** body is one which begins with either **do** or **once**. The other possibilities are **Deferred**, for which it would be improper to define an exception handler since the body does not specify an algorithm, and an **External** body, where the algorithm is specified outside of the Eiffel system, which then lacks the information it would need to handle exceptions.

*The various kinds of **Feature_body** are discussed in [8.5, page 222](#).*

The constraint on Retry instructions has already been mentioned:



Retry rule	VXRT
A Retry instruction is valid if and only if it appears in a Rescue clause.	

VXRT



Because this constraint requires the **Retry** physically to appear within the **Rescue** clause, it is not possible for a **Rescue** to call a procedure containing a **Retry**. In particular, a redefined version of *default_rescue* (see next) may not contain a **Retry**.

26.9 EXCEPTION CORRECTNESS



As described in a later chapter, every routine has a **rescue block**, syntactically a **Compound**, which takes over whenever an execution of the routine triggers an abnormal condition (an exception). The rescue block is the contents of the routine's **Rescue** clause, if any; otherwise it consists of a call to the routine *default_rescue*, which has a null effect in its default version (from class *ANY*) but may be redefined by any class. *Chapter 15.*

The execution of the rescue block may end in either of two ways:

- 1 • A rescue block which executes a **Retry** instruction causes the *Routine_body* to be executed again.
- 2 • If it terminates without executing a **Retry**, the rescue block causes the routine execution to fail, triggering an exception in the routine's caller (which will handle it in one of the same two ways).

To be correct, the rescue block must be such that any branch terminating with --- FiX --- a **Retry** (case 1) ensures the precondition and the invariant, and that any other branch (case 2) ensures the invariant. This provides for a first definition of exception correctness:



Exception-correct

A routine is **exception-correct** if any branch of the **Rescue** clause not terminating with a **Retry** ensures the invariant.

See “EXCEPTION CORRECTNESS”, §26.11, page 707, for a more precise definition of exception correctness.

26.10 SEMANTICS OF EXCEPTION HANDLING

Default Rescue Original Semantics

Class *ANY* introduces a non-frozen procedure *default_rescue* with no argument and a null effect.

As the following semantic rules indicate, an exception not handled by an explicit **Rescue** clause will cause a call to *default_rescue*. Any class can redefine this procedure to implement a default exception handling policy for routines of the class that do not have their own **Rescue** clauses.

To define the semantics of exception handling, it is convenient to consider that every feature has an implicit or explicit "rescue block":

DEFINITION

Rescue block

Any **Internal** or **Attribute** feature f of a class C has a **rescue block**, a **Compound** defined as follows, where rc is C 's version of ANY 's *default_rescue*:

- 1 • If f has a **Rescue** clause: the **Compound** contained in that clause.
- 2 • If r is not rc and has no **Rescue** clause: a **Compound** made of a single instruction: an **Unqualified_call** to rc .
- 3 • If r is rc and has no **Rescue** clause: an empty **Compound**.

The semantic rules rely on this definition to define the effect of an exception as if every routine had a **Rescue** clause: either one written explicitly, or an implicit one calling *default_rescue*. To this effect they refer not to **rescue** clauses but to rescue blocks.

Condition 3 avoids endless recursion in the case of *default_rescue* itself.

The procedure *default_rescue* in class ANY has, by default, no effect. Any class can redefine

SEMANTICS

Exception Semantics

An exception triggered during an execution of a feature *f* causes, if it is neither ignored nor continued, the effect of the following sequence of events.

- 1 • Attach the value of *last_exception* from *ANY* to a direct instance of a descendant of the Kernel Library class *EXCEPTION* corresponding to the type of the exception.
- 2 • Unlike in the non-exception semantics of *Compound*, do not execute the remaining instructions of *f*.
- 3 • If the recipient of the exception is *f*, execute the rescue block of *f*.
- 4 • If case 3 applies and the rescue block executes a *Retry*, this terminates the processing of the exception. Execution continues with a new execution of the *Compound* in the *Feature_body* of *f*.
- 5 • If neither case 3 nor case 4 applies (in particular in case 3 if the rescue block executes to the end without executing a *Retry*), this terminates the processing of the current exception and the current execution of *f*, causing a failure of that execution. If the execution of *f* was caused by a call to *f* from another feature, trigger an exception of type *ROUTINE_FAILURE* in the calling routine, to be handled (recursively) according to the present rule. If there is no such calling feature, *f* is the root procedure; terminate its execution as having failed.

After failure and termination, the run-time should normally produce a diagnostic similar to the exception history table of page =====.

False alarm was response E1 introduced on page ===== as part of the exception handling policy discussed in 255.

As usual in rules specifying the “effect” of an event in terms of a sequence of steps, all that counts is that effect; it is not required that the execution carry out these exact steps, or carry them in this exact order.

In step 1, the **Retry** will only re-execute the **Feature_body** of *r*, with all entities set to their current value; it does **not** repeat argument passing and local variable initialization. This may be used to ensure that the execution takes a different path on a new attempt.

In most cases, the “recipient” of the exception (case 3) is the current routine, *f*. For exception occurring in special places, such as when evaluating an assertion, the next rule, Exception Cases, tells us whether *f* or its caller is the “recipient”.

In the case of a **Feature_body** of the **Once** form, the above semantics only applies to the first call to every applicable target, where a **Retry** may execute the body two or more times. If that first call fails, triggering a routine failure exception, the applicable rule for subsequent calls is not the above Exception Semantics (since the routine will not execute again) but the Once Routine Execution Semantics, which specifies that any such calls must trigger the exception again.

Type of an exception

The **type** of a triggered exception is the generating type of the object to which the value of *last_exception* is attached per step 1 of the Expression Semantics rule.

Exception Cases

The triggering of an exception in a feature *f* called by a feature *caller* results in the setting of the following properties, accessible through features of the exception class instance to which the value of *last_exception* is attached, as per the following table, where:

- The **Recipient** is either *f* or *caller*.
- “**Type**” indicates the type of the exception (a descendant of *EXCEPTION*).
- If *f* is the root procedure, executed during the original system creation call, the value of *caller* as given below does not apply.

	Recipient	Type
Exception during evaluation of invariant on entry	<i>caller</i>	[Type of exception as triggered]
Invariant violation on entry	<i>caller</i>	<i>INVARIANT_ENTRY_VIOLATION</i>
Exception during evaluation of precondition	<i>caller</i>	[Type of exception as triggered]
Exception during evaluation of Old expression on entry	See <u>Old Expression Semantics</u>	
Precondition violation	<i>caller</i>	<i>PRECONDITION_VIOLATION</i>
Exception in body	<i>f</i>	[Type of exception as triggered]
Exception during evaluation of invariant on exit	<i>f</i>	[Type of exception as triggered]
Invariant violation on exit	<i>f</i>	<i>INVARIANT_EXIT_VIOLATION</i>
Exception during evaluation of postcondition on exit	<i>f</i>	[Type of exception as triggered]
Postcondition violation	<i>f</i>	<i>POSTCONDITION_VIOLATION</i>

This rule specifies the precise effect of an exception occurring anywhere during execution (including some rather extreme cases, such as the occurrence of an exception in the evaluation of an assertion). Whether the “recipient” is *f* or *caller* determines whether the execution of the current routine can be “retried”: per case 3 of the Exception Semantics rule, a **Retry** is applicable only if the recipient is itself. Otherwise a **ROUTINE_FAILURE** will be triggered in the *caller*.

In the case of an **Old** expression, a special rule, given earlier, requires the exception to be remembered, during evaluation of the expression on entry to the routine, for re-triggering during evaluation of the postcondition on exit, but only if the expression turns out to be needed then.

Exception Properties

The value of the query *original* of class **EXCEPTION**, applicable to *last_exception*, is an **EXCEPTION** reference determined as follows after the triggering of an exception of type **TEX**:

- 1 • If **TEX** does not conform to **ROUTINE_FAILURE**: a reference to the current **EXCEPTION** object.
- 2 • If **TEX** conforms to **ROUTINE_FAILURE**: the previous value of *original*.

The reason for this query is that when a routine fails, because execution of a routine *f* has triggered an exception and has not been able to handle it through a **Retry**, the consequence, per case 5 of the Exception Semantics rule, is to trigger a new exception of type **ROUTINE_FAILURE**, to which *last_exception* now becomes attached. Without a provision for *original*, the “real” source of the exception would be lost, as **ROUTINE_FAILURE** exceptions get passed up the call chain. Querying *original* makes it possible, for any other routine up that chain, to find out the *Ur*-exception that truly started the full process.

26.11 EXCEPTION CORRECTNESS

The role of Rescue clauses is to cope with unexpected events. Although in a well-designed system Rescue clauses will only be executed in rare, special conditions, they still have an obligation to maintain the consistency of objects.

In particular, a routine failure should leave the current object (corresponding to the target of the latest call) in a consistent state, satisfying the invariant, so as not to hamper further attempts to use the object if another routine is able, through resumption, to recover from the failure. Also, a `Retry` instruction, which will restart the `Feature_body`, should re-establish the routine's precondition, if any, since the precondition is required for the `Feature_body` to operate properly.

These two requirements yield the notion of **exception correctness**, one of the conditions which make up class correctness. As you may recall, a class is correct if it is consistent (every `Feature_body`, started in a state satisfying the precondition and the invariant, terminates in a state satisfying the postcondition and the invariant), loop-correct (loops maintain their invariant and every iteration decreases the variant), check-correct (the conditions of `Check` instructions are satisfied) and exception correct, a notion which was sketched in the general discussion of correctness but may now be made more precise.

Chapter 9 addressed correctness, with the full definition on page ==, as part of 9.16.

DEFINITION

Exception-correct

A routine r of a class C is **exception-correct** if and only if, for every branch b of its rescue block:

- 1 • If b ends with a `Retry`: `{true} b {INVC and then prer}`
- 2 • If b does not end with a `Retry`: `{true} b {INVC}`

In this rule, INV_C is the invariant of class C and pre_r is the precondition of r .

Here INV_C is the class invariant and pre_r is the precondition of r .

The definition involves the rescue block of a routine. Remember that the rescue block always exists: if the routine has a `Rescue` clause, then its `Compound` is the rescue block; otherwise the rescue block is the local version of `ANY`'s procedure `default_rescue`.

As with other correctness conditions, exception correctness should ideally be provable automatically, but in practice you will most likely have to ascertain it through informal means.

26.12 FINE-TUNING THE MECHANISM

Ignoring, continuing an exception

It is possible, through routines of the Kernel Library class *EXCEPTION*, to ensure that exceptions of certain types be:

- **Ignored:** lead to no change of non-exception semantics.
- **Continued:** lead to execution of a programmer-specified routine, then to continuation of the execution according to non-exception semantics.

The details of what types of exceptions can be ignored and continued, and how to achieve these effects, belong to the specification of class *EXCEPTION* and its descendants.

---- REWRITE In some cases it is useful to have finer control over the handling of exceptions. Features from the Kernel Library class *EXCEPTIONS* address this need. These features will be available to any descendant of *EXCEPTIONS*: to use them in a class *C* which is not already a descendant, just add a Parent part for *EXCEPTIONS* to the Inheritance clause of *C*.

Let us take a look at the facilities offered. A later chapter gives the full short-flat form of *EXCEPTIONS*.

See chapter 37 about the details of class EXCEPTIONS.

First, *EXCEPTIONS* introduces an integer code for every possible type of exception. Examples include

Precondition (code for a violated precondition)

Routine_failure

Incorrect_inspect_value

Void_call_target

No_more_memory

The integer-valued feature *exception* is then guaranteed, after an exception occurs, to have the value of the code for that exception. This makes it possible to write Rescue clauses such as

```

rescue
  if exception = No_more_memory
then
  ... Specific treatment...
else
  default_rescue
end

```



The call to *default_rescue* in the Else_part is not required, of course, but as a general guideline if you do need to treat certain categories of exception in a special way then such treatment should remain simple and apply to a small number of categories. You should handle the remaining categories through *default_rescue* or another general-purpose mechanism.

Another integer-valued feature, *original_exception*, complements the information given by *exception*. It yields the "real" cause of an exception, disregarding any resulting failures of intermediate routines. Consider for an example a Postcondition violation which causes a routine *t* to fail, triggering an exception whose recipient is *t*'s caller, *s*; the Rescue clause of *s*, if any, does not execute a Retry, so *s* in turn fails, sending an exception to its own caller, *r*. If the Rescue clause of *r* looks at *exception* to determine what happened, it will find as exception code the value of *Routine_failure*. This gives the immediate cause of *r*'s exception (the failure of *s*) but not the real source of the problem – *t*'s Postcondition violation. Feature *original_exception* provides more precise information in such cases. Its value is the code of the oldest exception not handled by a Retry. In the example, this will be the value of *Precondition*.

Features which provide further information about the original exception include *routine_name* (name of the original recipient) and *tag_name* (tag of the violated Assertion_clause, for an assertion violation).

Class *EXCEPTION* also provides a way to raise an exception on purpose. This is called a **developer exception** and is triggered by the procedure call

```
raise (code, name)
```

whose arguments are an exception code *code*, which must be a negative integer (non-negative values are reserved for predefined exceptions), and a string *name* describing the nature of the exception. To obtain that string when handling the exception, use feature *developer_exception_name*.

To know the general category of an exception given of a given *code* (usually *exception* or *original_exception*), use one of the boolean functions

```
is_assertion_violation (code)  
is_developer_exception (code)  
is_signal (code)
```

To prescribe a **false alarm** response you may use one of the procedure calls

```
ignore (code)  
continue (code)
```

A call to *ignore* simply prescribes that later occurrences of the event with the given *code* must not cause an exception.

After a call to *continue* with *code* as argument, any occurrence of the corresponding signal will cause execution of the appropriate version of procedure *continue_action*, followed by continuation of the **Feature_body** which was the signal's recipient. Procedure *continue_action* is introduced in class **EXCEPTIONS** with an empty body, but (like *default_rescue*) may be redefined in any class to yield specific behavior. The procedure has an integer argument *code* to which the exception handling mechanism, when calling *continue_action* as a response to an exception for which "continue" status has been prescribed, will attach the code of that exception.

The false alarm policy would not make sense for exceptions which cause irrecoverable damage to the current routine execution. For example, an assertion violation indicates a breach of some consistency condition, making it impossible to continue normal execution. For this reason, *continue* has the precondition *is_signal(code)*. No such precondition has been imposed on *ignore* in deference to the potential needs of developers of low-level systems software; except in very special cases, however, *ignore* must only be applied to signals.

To restore the default behavior after a call to *ignore* or *continue*, use the call

```
catch (code)
```

To know the behavior specified for *code*, use

status (code)

whose result, an integer, is given by one of the constants *Caught* (the default), *Continued* and *Ignored*.



As a final comment, it is useful to note once again that the best exception handling is simple and modest. The facilities of class *EXCEPTIONS* are there to give you full access to the context of exceptions if you need it; remember, however, that if you are trying to do something complicated in a Rescue clause, you are probably misusing the mechanism.

Non-trivial algorithms belong in the *Feature_body*; the Rescue clause is there to recover in a simple and non-committing way from abnormal situations which it was absolutely impossible to avoid.

26.13 OVERVIEW

As explained in the discussion of exceptions, it is sometimes useful to control the details of the exception handling mechanism. Applications include: *Chapter 26 discussed exceptions.*

- Finding out the nature of the latest exception (such as assertion violation, running out of memory or platform signal) and any other property, such as the *Assertion_tag* of a violated assertion clause.
- Handling certain kinds of exception in a special way.
- Raising special developer-defined exceptions.
- Prescribing that certain exceptions must be ignored at run-time, or must let execution continue after a call to a specified procedure.

"Platform" means hardware plus operating system. See 2.12, page 101.

All these facilities for fine-tuning the exception mechanism are available through features of class *EXCEPTIONS* from the Kernel Library, which is the subject of this chapter. To use these facilities in a class *C*, it suffices to ensure that *C* is a descendant of *EXCEPTIONS*.

Since the key concepts were introduced in the general discussion of exceptions, the rest of this chapter will simply give the flat-short form of class *EXCEPTIONS*, after a few comments about platform-dependent signal codes. *See the explanations in 26.12, page 709.*

26.14 PLATFORM-DEPENDENT SIGNAL CODES

The exception codes introduced in class *EXCEPTIONS*, such as *No_more_memory* or *Precondition*, cover exceptions which will arise in the same manner on every platform.

Some platform-dependent machine signals, however, will also trigger exceptions. Unix systems, for example, may raise signals such as "change of child process status" or "writing on a pipe with no one to read it".

To enable systems to specify platform-specific exception handling when appropriate, Eiffel implementations may include library classes extending the features of *EXCEPTIONS* with platform-specific exception handling features, in particular exception codes. The recommended names for such classes are of the form *platform_EXCEPTIONS*, for example *UNIX_EXCEPTIONS*, *MS_DOS_EXCEPTIONS* and so on. Classes which need the specific features should have one of these classes, in addition to *EXCEPTIONS*, as one of their ancestors.

No platform-dependent exception class appears in this book.



A system which uses one of these platform-specific classes will of course require some adaptation to be ported to other platforms. If you do need platform-specific exception handling, you should severely restrict the number of classes that inherit from the appropriate *platform_EXCEPTIONS* class, so as to facilitate any eventual porting effort.

26.15 CLASS EXCEPTIONS

Here is the short form of class *EXCEPTIONS*.

```

-- Facilities for controlling exception handling.
class interface EXCEPTIONS exported features
assertion_violation: BOOLEAN
-- Was last exception due to a violated assertion or
-- non-decreasing variant?
catch (code: INTEGER)
-- Make sure that any exception of code code will be caught.
-- This is the default. (See continue, ignore.)
ensure
status (code) = Caught
Check_instruction: INTEGER
-- Exception code for violated Check
Class_invariant: INTEGER
-- Exception code for violated class invariant
class_name: STRING
-- Name of the class containing the routine which
-- was the recipient of oldest exception not leading to a Retry.
continue (code: INTEGER)
-- Make sure that any exception of code code will cause
-- execution to resume after a call to continue_action (code).
-- This is not the default. (See catch, ignore.)
require
must_be_a_signal: is_signal (code)
ensure
status (code) = Continued
continue_action (code: INTEGER)
-- Action to be executed before resuming normal execution for an
-- exception of code code, resulting from a signal,
-- on which continue has been called.
-- By default, does nothing; redefine it to get specific behavior.

```


require

must_be_continued: status (code) = Continued
developer_exception_name: STRING
-- Name of last developer-raised exception (see raise)
exception: INTEGER
-- Code of last exception that occurred
ignore (code: INTEGER)
-- Make sure that any exception of code code will be ignored.
-- This is not the default. (See catch, continue.)

ensure

status (code) = Ignored
Incorrect_inspect_value: INTEGER
-- Exception code for inspect value which is not one of the
-- inspect constants, if there is no Else_part
is_developer_exception (code: INTEGER): BOOLEAN
-- Is the code of a developer-defined exception (see raise)?
is_assertion_violation (code: INTEGER): BOOLEAN
-- Is the code of an exception resulting from the violation
-- of an assertion (precondition, postcondition, invariant, check)?
is_signal (code: INTEGER): BOOLEAN
-- Is code the code of an exception due to a hardware
-- or operating system signal?
Loop_invariant: INTEGER
-- Exception code for violated loop invariant
Loop_variant: INTEGER
-- Exception code for non-decreased loop variant
meaning (code): STRING
-- Nature of exception of code code, expressed in plain English
message_on_failure
-- Print an Exception History Table in case of failure.
-- (This is the default; see no_message_on_failure.)

```

no_message_on_failure
-- Do not print an Exception History Table in case of failure.
-- (This is not the default; see message_on_failure.)
No_more_memory: INTEGER
-- Exception code for failed memory allocation
original_exception: INTEGER
-- Code of oldest exception not leading to a Retry
Postcondition: INTEGER
-- Exception code for violated postcondition
Precondition: INTEGER
-- Exception code for violated precondition
raise (code: INTEGER; name: STRING)
-- Raise a developer exception of code code and name name.
require
negative_code: code < 0
reset_all_default
-- Make sure that all exceptions will lead to their default handling.
reset_default (code: INTEGER)
-- Make sure that exception of code code will lead
-- to its default action.
Routine_failure: INTEGER
-- Exception code for failed routine
routine_name: STRING
-- Name of routine that was recipient of oldest exception
-- not leading to a Retry
status (code: INTEGER): INTEGER
-- Status currently set for exception of code code (default: Caught)
ensure
Result = Caught or Result = Continued or Result = Ignored
Caught, Continued, Ignored: INTEGER is unique
-- Possible status for exception codes
tag_name: STRING
-- Tag of last violated assertion clause

```

Void_assigned_to_expanded: INTEGER

-- Exception code for assignment of
void value to expanded entity

Void_call_target: INTEGER

-- Exception code for feature called on
void reference

void_call_feature: STRING

-- Name of feature that was called on a
void reference

end interface -- class *EXCEPTIONS*

Agents, iteration and introspection

27.1 OVERVIEW

Objects represent information equipped with operations. These are clearly defined concepts; no one would mistake an operation for an object.

For some applications — graphics, numerical computation, iteration, writing contracts, building development environments, “reflection” (a system’s ability to explore its own properties) — you may find the *operations* so interesting that you will want to define *objects* to represent them, and pass these objects around to software elements, which can use these objects to execute the operations whenever they want. Because this separates the place of an operation’s *definition* from the place of its *execution*, the definition can be incomplete, since you can provide any missing details at the time of any particular execution.

You can create **agent** objects to describe such partially or completely specified computations. Agents combine the power of higher-level functionals — operations acting on other operations — with the safety of Eiffel’s static typing system.

27.2 A QUICK PREVIEW



Why do we need agents? Here are a few examples. This preview skips many details but will give you an idea of the power of the mechanism; any apparent mystery will soon be cleared as you read further into the chapter.

Let’s start with a typical need of graphical user interface (GUI) programming. Using EiffelVision, the multi-platform graphical library of Eiffel Software, you may write


```
your_button.click_actions.extend(agent your_routine)
```



to add *your_routine* — a routine of your application, executing appropriate operations — to the list of actions triggered by a mouse click on *your_button*. This is all you need to set up the application’s response to such an event.

The argument to *extend*, **agent** *your_routine*, is an **agent expression**. The keyword **agent** avoids confusion with an actual routine call: when calling *extend*, you don't want to call *your_routine* yet! Instead you pass to *extend* an "agent", which *extend* adds to the *click_actions* list for *your_button*, enabling EiffelVision to call *your_routine* for every subsequent occurrence of a click event on the button. The agent includes any context information that *your_routine* may need: cursor position, button number, pressure.


Now a numerical example. Over the interval [0, 1], you want to integrate a function $g(x: REAL): REAL$. With *your_integrator* of a suitable type *INTEGRATOR* (detailed later), just use the expression



```
your_integrator.integral (agent g (?), 0.0, 1.0)
```

Again this doesn't call the routine *g*, but enables *integral* to call *g* when it pleases, as often as it pleases, on whatever values it pleases. We must tell *integral* where to substitute such values for *x* at the places where its algorithm needs to evaluate *g* to approximate the integral. This is the role of the question mark *?*, replacing the argument to *g*.

You may use the same scheme in




```
your_integrator.integral (agent h (u, ?, v), 0.0, 1.0)
```

to compute the integral $\int_0^1 h(u, x, v) dx$, where *h* is a three-argument function $h(a: T1; x: REAL; b: T2): REAL$ and *u* and *v* are arbitrary values. As before you will use a question mark at the "open" position, corresponding to the integration variable *x*. Two "closed" positions show actual values *u* and *v*.

Note the flexibility of the mechanism: it allows you to use the same routine, *integral*, to integrate a one-argument function such as *f* as well as functions such as *h* involving extra values.

You can rely on a similar structure to provide **iteration** mechanisms on data structures such as lists. Assume a class *CC* with an attribute



```
intlist: LINKED_LIST [INTEGER]
```

and a function



```
integer_property (i: INTEGER): BOOLEAN
```

returning true or false depending on a property involving *i*. You may write

```
intlist.for_all (agent integer_property (?))
```

to denote a boolean value, true if and only if every integer in the list *intlist* satisfies *integer_property*. This expression might be useful, for example, in a class invariant. It will work for any kind of *integer_property*, even if this function involves arbitrary features of the current object.

Now assume that in *CC* you also have a list of employees:



```
emplist: LINKED_LIST [EMPLOYEE]
```

and that class *EMPLOYEE* has a function *is_married: BOOLEAN* with no argument, telling us about the current employee's marital status. Then you may also write in *CC* the boolean expression

```
emplist.for_all ( agent {EMPLOYEE}.is_married )
```

to find out whether all employees in the list are married. The argument to *for_all* is imitated from a normal feature call *some_employee.is_married*, but instead of specifying a particular employee we just give the type *{EMPLOYEE}*, to indicate where *for_all* must evaluate *is_married* for successive targets taken from the the list.

The *{EMPLOYEE}* notation replaces the question mark of the previous examples. Those examples used an argument as the open operand — the place where the routine will be evaluated — as in *integer_property(?)*, where the argument type is clear from the declaration of *integer_property*. But with *is_married* the open operand is the target, so we need to specify the type: many classes may have a function called *is_married*.



Note again the flexibility of the iteration mechanism and its adaptation to the object-oriented form of computation: you can use the same iteration routine, here *for_all* from *LINKED_LIST*, to iterate actions applying to either:

- The **target** of a feature, as with *is_married*, a feature of class *EMPLOYEE*, to be applied to its *EMPLOYEE* target.
- The **actual argument** of a feature, as with *integer_property* which evaluates a property of its argument *i* — and may or may not, in addition, involve properties of its target, an object of type *CC*.



It seems mysterious that a single iterator mechanism can handle both cases equally well. We will see how to write *for_all* and other iterators accordingly. The trick is that they simply work on their open operands; when calling them, you choose what to leave open: either the argument as with *integer_property* and *integral*, or the target as with *is_married*.

Now assume that you want to pass to some object the mechanisms needed to execute the cursor resetting and advance operations, *start* and *forth*, on a particular list. Here nothing is left open: you fix the list, and the operations have no arguments. You may write

This is the iterator style of the C++ STL (Standard Template Library).

```
object.operation ( agent your_list.start, agent your_list.forth )
```

All operands — target and arguments — of the agents passed to *object* are “closed”, so *object* can execute call operations on such objects without providing any further information.

At the other extreme, you might leave an agent expression fully open, as in

```
object.operation ( agent {LINKED_LIST [T]}.extend (?) )
```

so that *object*, when it desires to apply a call operation, will have to provide both a linked list and an actual argument to execute *extend*. When as here all the arguments are open, you may omit the argument list, writing just *agent {LINKED_LIST [T]}.extend*. Such an agent is a “**routine object**”: an object representing the routine *extend* from *LINKED_LIST*, such as could be used by browsing tools or other *reflection* facilities.



To use an agent, a routine such as *operation* can apply to it the procedure *call*, passing a tuple of values for the open operands. This will have the same effect as an execution of the original feature — *f*, *h*, *integer_property*, *is_married*, *start*, *forth*, *extend* ... — on all the operands, closed and open.

The notation provides an extra degree of flexibility by letting you define **inline agents**, which instead of referring to a feature of the class define a routine text as part of the agent declaration. Inline agents have the same form as a **Routine** body, as in



```
(agent (i: INTEGER): BOOLEAN do Result := integer_property (i)
-- Means the same as: agent integer_property (?)
(agent (e: EMPLOYEE): BOOLEAN do Result := e.is_married end)
-- Means the same as: agent {EMPLOYEE}.is_married
```

In these examples the previous forms were simpler and shorter, but inline agents are useful when you want to express the computation just for the agent, without making it a routine of the enclosing class. For example you may define the inline agent



```
(agent (i: INTEGER): BOOLEAN
do Result := (item (i) = a.item (i) + b.item (i)) end)
```

which could be useful in a postcondition

```
summed: (lower |..| upper).for_all
((agent (i: INTEGER): BOOLEAN
do Result := (item (i) = a.item (i) + b.item (i)) end))
```

This states that for every element *i* of the interval *lower* |..| *upper* the value of the item at position *i* (in a structure such as an array or list) is the sum of the corresponding values in *a* and *b*. To obtain the same semantics without agent arguments, you would need to express the agent as **agent is_sum_of** (?, *a*, *b*) and define a function *is_sum_of* such that *is_sum_of* (*i*, *x*, *y*) is true if and only if *item* (*i*) = *x.item* (*i*) + *y.item* (*i*). The semantics is the same, but if you have many properties of this kind — for example in contracts — the inline form avoids introducing many specialized functions such as *is_sum_of*.

In this example the agent represents a function, with an expression as its body: *item* (*i*) = *a.item* (*i*) + *b.item* (*i*). It is also possible to use an inline form for a procedure agent, as in



```
emplist.do_all( (agent (e: EMPLOYEE) do sum:=sum + e.salary end) )
```

where *do_all* applies its agent argument to all successive elements in a list; this increases *sum* by the total of all employees' salaries.



For an agent involving a single routine such as *integer_property*, *integral*, *is_married*, *extend* and the other previous examples, the original non-inline form is shorter, more abstract, and usually preferable.

You may wonder how this can all work in a type-safe fashion. So it is time to stop this preview and cut to the movie.

27.3 FROM CALLS TO AGENTS

Feature calls and their operands



First we should remind ourselves of the basic properties of **feature calls**. When programming with Eiffel we rely all the time on this fundamental mechanism of object-oriented computation. We write things like

← *Feature calls were studied in chapter 23 and their type properties in chapter 25.*

[Q] $a0.f(a1, a2, a3)$

to mean: call feature f on the object attached to $a0$, with actual arguments $a1$, $a2$, $a3$. In Eiffel this is all governed by type rules, checkable statically: f must be a feature of the base class of the type $a0$; and the types of $a1$ and the other actuals of the call must all conform to the types specified for the corresponding formals in the declaration of f .

In a frequent special case $a0$, the **target** of the call, is just *Current*, denoting the current object. Then we may omit the dot and the target altogether, writing the call as just

[U] $f(a1, a2, a3)$

which assumes that f is a feature of the class in which this call appears. The first form, with the dot, is a *qualified* call; the second form is *unqualified* (hence the names [Q] and [U] given to our two examples).

In either form the call is syntactically an expression if f is a function or an attribute, and an instruction if f is a procedure. If f has been declared with no formals (as in the case of a function without arguments, or an attribute) we omit the list of actuals, $(a1, a2, a3)$.

The effect of executing such a call is to apply feature f to the target object, with the actuals given if any. If f is a function or an attribute, the value of the call expression is the result returned by this application.

To execute properly, the call needs the value of the target and the actuals, for which this chapter needs a collective name:



Operands of a call

The **operands** of a call include its target (explicit in a qualified call, implicit in an unqualified call), and its arguments if any.

In the examples the operands are $a0$ (or *Current* in the unqualified version [U]), $a1$, $a2$ and $a3$. Also convenient is the notion of *position* of an operand:

Operand position

The target of a call has **position 0**. The i -th actual argument, for any applicable i , has **position i** .

← *Every Object_call has a target, as defined on page 628.*

Positions, then, range from 0 to the number of arguments declared for the feature. Position 0, the target position, is always applicable.

Delaying calls



For a call such as the above, we expect the effect just discussed to occur as a direct result of executing the call instruction or expression: the computation is immediate. In some cases, however, we might want to write an expression that only *describes* the calls intended computation, and to *execute* that description later on, at a time of our own choosing, or someone else's. This is the purpose of agent expressions, which may be described as **delayed calls**.

Why would we delay a call in this way? Here are some typical cases:

- A • We might want the call to be applied to all the elements of a certain structure, such as a list. In such a case we will specify the agent expression once, and then execute it many times without having to re-specify it in the software text. The software element that will repeatedly execute the same call on different objects is known as an **iterator**. Function *for_all*, used earlier, was an example of iterator.
- B • In an iterator-like scheme for numerical computation, we might use a mechanism that applies a call to various values in a certain interval, for example to approximate the integral of a function over that interval. The first example in this chapter relied on such an *integral* function.
- C • We might want the call to be executed by another software element: passing an agent object to that element is a way to give it the right to operate on some of our own data structures, at a time of its own choosing. This was illustrated with the calls passing to *object* some agent expressions representing operations applicable to *your_list*. GUI examples also belong to that category: to state that a certain action must be executed whenever a certain event (such as mouse click) occurs on a certain graphical object (such as a button), we add an agent representing the action to a list of agents associated with the object and the event.
- D • We might want to ensure that the call is executed only when and if needed, and then only once for any particular object. This would give us a “once per object” mechanism along the lines of “once functions” (which are executed once per system).
- E • Finally, we may be interested in the agent as a way to gain information about the feature itself, whether or not we ever intend to execute the call. This may be part of the more general goal of providing **introspective** capabilities: ways to enable a software system to explore and manipulate information about its own properties.

*Once functions see “ROUTINE BODY”, 8.5, page 222. The once per object mechanism using agents is described below. Introspection is also called **reflection**, but the first term appears more appropriate.*

These examples illustrate one of the differences between an agent expression and a plain feature call: to execute a feature call we need the value of all its operands (target and actuals); but for an agent expression we may want to leave some of the operands open for later filling-in. This is clearly necessary for cases **A** and **B**, in which the iteration or integration mechanism will need to apply the feature repeatedly, using different operands each time. In an integration

$$\int_{x=a}^{x=b} g(x) dx$$

we will need to apply g to successive values of the interval $[a, b]$.

Agents and their operands

For an agent we need to distinguish between two moments:



Construction time, call time

The **construction time** of an agent object is the time of evaluation of the agent expression defining it.

Its **call time** is when a call to its associated operation is executed.

Since the only way to obtain an agent initially is through *agent expressions*, as specified next, it is meaningful to talk about the “agent expression defining it”.

For a normal call the two moments are the same. For an agent we will have one construction time (zero if the expression is never evaluated), and zero or more call times. At construction time, we may leave some operands unspecified; they they will be called the *open* operands. At call time, however, the execution needs all operands, so the call will need to specify values for the open operands. These values may be different for different executions (different call times) of the same agent expression (with a single construction time).

→ A precise definition of “open” and “closed” operands appears on page 758.

Readers familiar with lambda calculus may think of open as “free” and closed as “bound”.

There is no requirement to make **all** operands open at construction time: you may provide some operands, which will be closed, and leave some others open. In the example of computing, for some values u and v , the integral

where h is a three-argument function, we pass to the integration mechanism an agent that is closed on its first and last operands, u and v , but open on x .

$$\int_{x=a}^{x=b} h(u, x, v) dx$$

Nothing forces you, on the other hand, to leave **any** operand open. An agent with all operands closed corresponds to the kind of application called **C** above, in which we don't want to execute the call ourselves but let another software element carry it out when it is ready. We choose the construction time, and package the call completely, including all the information needed to carry it out; the other software element chooses the call time. This style is used by iterators in the C++ STL library.

At the other extreme, an agent with **all operands open** has no information about the target and actuals, but includes all the relevant information about the feature. This is useful in application **E**: passing around information about a feature for introspection purposes, enabling a system to deliver information about its own components.

27.4 AGENT TYPES

A normal call is a syntactical component — instruction or expression — meant only for one thing: immediate execution. If it is an expression (because the feature is a function), it has a value, computed by the execution, and so it denotes an object.



An agent expression has a different status. Since construction time is separate from call time, the agent expression can only **denote an object**. That object (an agent) contains all the information needed to execute the call later, at various call times. This includes in particular:

- Information about the routine itself and its base type.
- The values of all the closed operands.

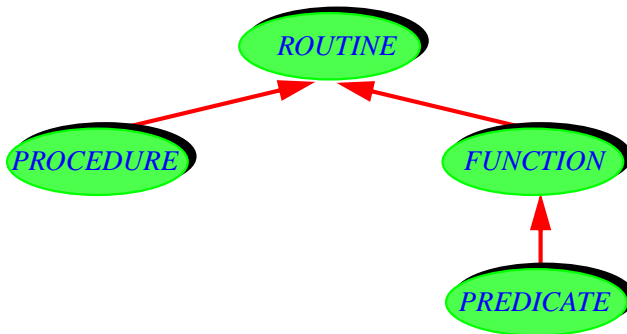
What is the type of an agent expression? Four Kernel Library classes are used to describe such types: *ROUTINE*, *PROCEDURE*, *FUNCTION* and *PREDICATE*. Their class headers start as follows:

```
deferred class ROUTINE [BASE, OPEN -> TUPLE]
class PROCEDURE [BASE, OPEN -> TUPLE] inherit
  ROUTINE [BASE, OPEN]
class FUNCTION [BASE, OPEN -> TUPLE, RES] inherit
  ROUTINE [BASE, OPEN]
class PREDICATE [BASE, OPEN -> TUPLE] inherit
  FUNCTION [BASE, OPEN, BOOLEAN]
```

In the actual class texts, the formal generic matters have names *BASE_TYPE*, *OPEN_ARGS* and *RESULT_TYPE* to avoid conflicts with programmer-chosen class names. This chapter uses shorter names for simplicity.

→ [A.6.30](#) to [A.6.32](#) in the *ELKS* chapter, starting on page [1011](#).

If the associated feature is a procedure the agent will be an instance of *PROCEDURE*; for a function or attribute, we get an instance of *PREDICATE* when the result is boolean, of *FUNCTION* with any other type. Here for ease of reference is a picture of the inheritance hierarchy:



Agent classes

The role of the formal generic parameters is:

- *BASE*: type (class + generics if any) to which the feature belongs.
- *OPEN*: tuple of the types of open operands, if any.
- *RES*: result type for a function.

One of the fundamental features of class *ROUTINE* is

```
call (v: OPEN)
  -- Call feature with all its operands, using v for the open operands.
```

In addition, *FUNCTION* and *PREDICATE* have the feature

```
last_result: RES
  -- Function result returned by last call to call, if any
```

and, for convenience, the function *item* combining *call* and *last_result*, with the following specification:

```
item (v: like open_operands): RES
  -- Result of calling feature with all its operands,
  -- using v for the open operands.
  -- (Uses call for the call.)
ensure
  set_by_call: Result = last_result
```

The formal generic parameters for *ROUTINE*, *PROCEDURE*, *FUNCTION* and *PREDICATE* provide what we need to make the agent mechanism statically type-safe. *OPEN*, a tuple type, gives the exact list of open operand types; since the argument to *call* and *item* is of type *OPEN*, it is possible from the software text to check that the actual arguments to *call* will at call time be of the proper types, conforming to the original feature's formal argument types at the open positions. The actuals at closed positions are set at construction time, again with type checking. So the combination of open and closed actuals will be type-valid for the feature.



ROUTINE, *PROCEDURE*, *FUNCTION* and *PREDICATE* have more features than listed above; in particular, they provide introspection facilities, describing properties of the associated routines and discussed below. For a complete interface specification, see the corresponding sections in the presentation of Kernel Library classes.

→ Sections [A.6.30](#) to [A.6.32](#), starting on page [1011](#).

27.5 CALL AGENTS

How do we obtain agent objects? The most common construct is a *call agent* expression. (We will see the other case, *inline agents*, in a later section.)

→ “[USING INLINE AGENTS](#)”, 27.8, page [746](#).

The basic form of a call agent is very simple: just add the keyword **agent** at the beginning of a normal feature call. This yields an agent with operands all closed. To specify open operands, you may:

- Use a question mark **?** in lieu of an argument.
- Use a type in braces, **{TYPE}**, in lieu of the target.
- Omit the argument list altogether, to make all arguments open.

Let's examine these variants and the associated semantics.

All-closed agents

If you start from a valid call, either qualified or unqualified



```
[Q]    a0.f(a1, a2, a3)
[U]    f(a1, a2, a3)
```

you get an agent expression in each case by adding the keyword **agent**:

```
agent a0.f(a1, a2, a3)
agent f(a1, a2, a3)
```

Such an agent expression is not a call (instruction or expression) any more, but an expression of a new syntactic kind, **Feature_agent**, denoting an agent, of a **PROCEDURE** type if f is a procedure. and a **FUNCTION** or **PREDICATE** type if f is a function. Both of these examples have no arguments, so they are closed on all operands; we will start adding arguments soon.

You can do with an agent expression all you are used to do with other expressions. You can assign it to an entity of the appropriate type; assuming f is a procedure of a class **CC**, you may write, in class **CC** itself:

*This example assumes that **CC** is non-generic, so that it is both a class and a type.*



```
p: PROCEDURE [CC, TUPLE]
...
p:= agent a0.f(a1, a2, a3)
...
p.call([])
```

Since all operands are closed — we have specified the target $a0$ and all the arguments $a1$, $a2$, $a3$ — the second formal generic is just **TUPLE**, and the call to **call** takes an empty tuple **[]**.

More commonly than assigning a call expression to an entity as here, you will pass it as actual argument to a routine, as in



```
object.do_something (agent a0.f(a1, a2, a3))
```

where **do_something**, in the corresponding class, takes a formal p declared as

```
p: PROCEDURE [CC, TUPLE]
```

or just

```
p: PROCEDURE [ANY, TUPLE]
```

presumably to call **call** on p at some later stage, as we will shortly learn to do. This was the scheme called **C** in the presentation of example applications: passing a completely closed agent to another component of the system, to let it execute the call when it chooses to. For example you can pass **agent your_list.start** or **agent your_list.extend(some_value)**.

← Scheme **C** was on page 724.

Keeping operands open

The examples just seen are still of limited interest because all their operands are closed. But you may want to keep some operands open for latter filling-in at call time, for example by an iteration or integration mechanism.

To specify an open target, you will replace the target by its type in braces, `{TARGET_TYPE}`. This is the **brace convention**. To specify an open argument, you will use the **question mark convention**: just replace the target by a question mark `?`.

Here are some examples, obtained by starting from the call `a0.f(a1, a2, a3)` and opening the target or some arguments.



```

-- Start with an agent closed on all operands:
s:= agent a0.f(a1, a2, a3)
-- Next, individually open the target and each successive argument:
t:= agent {T0}.f(a1, a2, a3)
u:= agent a0.f(?, a2, a3)
v:= agent a0.f(a1, ?, a3)
w:= agent a0.f(a1, p, ?)
-- An example with two open arguments, target closed:
x := agent a0.f(a1, ?, ?)
-- Arguments all open, target still closed:
y := agent a0.f(?, ?, ?)
-- Finally, open everything:
z := agent {T0}.f(?, ?, ?)

```

The respective types of these call expressions are, assuming that `f` is a procedure declared in the base class of `T0`, having formals declared of types `T1`, `T2` and `T3`:

```

s: PROCEDURE [T0, TUPLE]
t: PROCEDURE [T0, TUPLE [T0]]
u: PROCEDURE [T0, TUPLE [T1]]
v: PROCEDURE [T0, TUPLE [T2]]
w: PROCEDURE [T0, TUPLE [T3]]
x: PROCEDURE [T0, TUPLE [T2, T3]]
y: PROCEDURE [T0, TUPLE [T1, T2, T3]]
z: PROCEDURE [T0, TUPLE [T0, T1, T2, T3]]

```

If `f` were a function, the types would use `FUNCTION` instead of `PROCEDURE`, with an extra generic parameter representing the result type (except for a boolean-valued function, which would use `PREDICATE`).

The first generic parameter, *T0* in all of these examples, represents the current type (class with generic parameters if any) of the underlying feature. Here we assume for simplicity that *f* comes from a non-generic class *T0*.

→ “THE BASE CLASS AND TYPE”, 27.10, page 750.

← “CURRENT TYPE, FEATURES OF A TYPE”, 12.11, page 365



The second generic parameter, a tuple type, represent the sequence of types of open operands. For the first example, *t*, it’s just *TUPLE* with no parameters, since the agent has no open operands. For the other examples the parameters of the *TUPLE* type represent the types of the open operands. They indicate what argument types are permissible in calls to *call* (or *item* for a function) on the corresponding agents.

Here indeed are examples of valid uses of *call* on the previous agent examples. For each of them, the comment on the next line shows how we would have obtained the same effect through a normal call (call time same as construction time, not using agents).



```

val_0: T0; val_1: T1; val_2: T2; val_3: T3
... Assign values to val_0, val_1, val2, val_3 ...
s. call ([])      -- Note empty tuple: no open operands
   -- a0.f(a1, a2, a3)
t. call ([val_0])
   -- val_0.f(a1, a2, a3)
u. call ([val_1])
   -- a0.f(val_1, a2, a3)
v. call ([val_2])
   -- a0.f(a1, val_2, a3)
w. call ([val_3])
   -- a0.f(a1, a2, val_3)
x. call ([val_2, val_3])
   -- a0.f(a1, val_2, val_3)
y. call ([val_1, val_2, val_3])
   -- a0.f(val_1, val_2, val_3)
z. call ([val_0, val_1, val_2, val_3]) -- Must provide all operands
   -- val_0.f(val_1, val_2, val_3)

```

It should be clear by now how mechanisms such as *for_all* can manage to work on operations that work on their target, such as *is_married*, as well as others that work on an argument, such as *is_positive*. The type of an agent only describes, through the *OPEN* parameter, the tuple of types of operands. It doesn’t make any difference whether these open operands come from a target or an argument.

For example, both of the following boolean expressions

```
emplist.for_all (agent {EMPLOYEE}.is_married)
emplist.for_all (agent object.is_married (?))
```

will be valid if:

- Class *EMPLOYEE* has, as previously assumed, a feature *is_married: BOOLEAN*.
- *object* is of type *SOME_TYPE*, and *SOME_TYPE* has a feature *is_married (e: BOOLEAN)*.

The brace convention

Two of the examples used the brace convention to keep the target open:

```
t := agent {T0}.f(a1, a2, a3)
z := agent {T0}.f(?, ?, ?)
    -- Also expressible (see below) as just: agent {T0}.f
```

Applicable in any class text.

For the target, as noted, the question mark convention is not applicable, since the feature name does not suffice to identify the target type: many classes may have a feature called *f*.

For arguments we have no such problem since once we know *f* and its class we know the declared type of each of *f*'s formal arguments. This justifies the question mark convention for arguments.

Omitting the argument list

A further simplification of the notation is available when *all* arguments are open, as in **agent** *a0.f* (?, ?, ?). Then you may omit the parenthesized argument list, as in

```
agent a0.f
    -- Abbreviates agent a0.f(?, ?, ?)
```

A call of the form *a0.f* would be invalid, since *f* always requires three actual arguments. But with an **agent** expression the convention of omitting arguments creates no ambiguity; it simply means that we consider an agent built from *f* with all arguments open.



This fully abbreviated form has the advantage of conveying the idea that the denoted agent is a true “feature object”, carrying properties of the feature in its virginal state, not tainted by any particular choice of actual argument. The last two variants shown do not even name a target. This is the kind of object that we need for such *introspective* applications as writing a system that enables its users to browse through its own classes.

A summary of the possibilities

As a summary of the preceding examples, here is a summary of the ways to build a call agent:

Syntactical forms for a call agent

A call agent is of the form

agent *agent_body*

where *agent_body* is a **Call**, qualified (as in *x.r (...)*) or unqualified (as in *f(...)*) with the following possible variants:

- You may replace any argument by a question mark **?**, making the argument open.
- You may replace the target, by **{TYPE}** where **TYPE** is the name of a type, making the target open.
- You may remove the argument list (...) altogether, making all arguments open.

This is not a formal syntax definition, but a summary of the available forms permitted by the syntax and validity rules that follow.

27.6 USING AGENTS

Although we have studied only one of the two syntactical forms of agents, call agents (the other is inline agents), and not yet taken the trouble to look at the syntax, validity rules and precise semantics, we have enough background to explore applications of agents, starting with the examples sketched at the very beginning of this chapter, which we can now revisit and extend. We’ll see how to make them work in practice: not just the client side — registering an action to be executed for a certain GUI event, integrating a function, iterating an operation — but the suppliers too: the event processing, the integrator, the iterators.

→ *The rules start with “AGENT SYNTAX”, 27.11, page 751.*

GUI programming: establishing a direct connection to the Business Model

The first example illustrated the EiffelVision style of GUI programming. We wrote *The actual EiffelVision events are `select` and `pointer_button_press..`*



```
your_button.click_actions.extend(agent your_routine)
```

to specify that *your_routine* must be executed whenever the *button_press* event occurs on *your_button* during execution. Here is how things work. In your application, *your_button* denotes a graphical object, variously known as a “control” (the Windows terminology), a “widget” (the X Windows terminology) or a “context”; *click* denotes one of the events that may occur on this control. The list *your_button.click_actions* contains agents, representing the actions to be executed when the event occurs on the control. This is a plain list (from the EiffelBase library), to which we may, as here, apply the procedure *extend*, adding a new item at the end.

When EiffelVision detects that the event has occurred on the button, it will execute, for every element *item* of the list of agents, a call such as *The actual version needs arguments to `your_routine`; see next..*

```
item.call ([])
```

For the list *item* that represents *your_routine*, this will produce what we wanted: a call to *your_routine* in response to the event.

This setup assumes that *your_routine* is a routine without arguments. In reality, a routine to be executed as a result of a mouse event, such as a click, may need the *x*, *y* mouse coordinates of the event. Let’s call it *your_routine2*. What EiffelVision actually executes is

```
item.call ([mouse_horizontal, mouse_vertical])
```

using as arguments the cursor coordinates, part of the event’s information recorded in the event. This assumes of course that *your_routine2* can deal with these arguments. If *your_routine2* indeed takes two real values as arguments, the previous form of registering the agent



```
your_button.click_actions.extend(agent your_routine2)
```

is still applicable; as you will remember, it is a shortcut for

```
your_button.click_actions.extend(agent your_routine(?, ?))
```

Now assume that *your_routine* is a routine from the “Business Model” part of your application, meaning the part of the software that takes care of doing the real processing, independently of any GUI. The *x* and *y* values might be only some of the arguments that *our_routine* needs. For example *your_routine* might be the procedure



```
compute_stats (country: COUNTRY; year: INTEGER; x, y: REAL)
```

which, in a cartographical application, computes statistics for a certain *year* for the city closest to positions *x* and *y* on the map for a certain *country*. When loading the map for that country you may register *compute_stats*:

```
your_button.click_actions.extend  
  (agent compute_stats (Usa, 2002, ?, ?))
```

The beauty of the notion of closed and open arguments is that you can set some values (here the country and the year) at construction time, and leave others (here the mouse coordinates) to be filled in at call time.

To the EiffelVision mechanism, there is no difference between this case using *compute_stats* — a routine with four arguments, two of which we have closed at construction time — and the previous one involving *your_routine2* and its two open arguments. The call executed by the EiffelVision side, shown above as

```
item.call ([mouse_horizontal, mouse_vertical])
```

works properly in both cases.

This scheme, relying on open and closed arguments, has crucial practical consequences for the programming of GUI applications. Following the MVC model introduced by Smalltalk, it is often stated that GUI applications should include three components:

- *Model* (the acronym’s M), called the **Business Model** above: this is the part that does the actual computation, data manipulation and processing. A routine such as *compute_stats*, describing some important operation of the Business Model, belongs to this part of the system.
- *View* (the “V”): the purely graphical part of the application, taking care of presenting information visually and interacting with users. Notions such as buttons, other controls and events belong to that part.
- *Controller* (the “C”): software elements that connect the model with the view, by specifying what operations from the model must be executed in response to what user interface events.

Without agents, the Controller part, serving as glue between Model and View, can take up a significant amount of code, based for example on **command classes**. As the last example indicates, using agents can bring the need for such glue code down to a minimum, or even remove it altogether. The only Controller element that we used in this example to connect the button and event to the routine `compute_stats` from our model was the agent **agent compute_stats** (Usa, 2002, ?, ?). You don't have to write any other code: no new class, not even any special instructions.

The command class technique is described in detail in the book "Object-Oriented Software Construction, 2nd edition."



This is one of the great benefits of agents for GUI programming, as used extensively in EiffelVision: **you directly connect elements from the Business Model to elements from the User Interface**, without requiring any “glue code”. The notion of open and closed operands gives us remarkable flexibility: as long as a routine from the Business Model, such as `compute_stats`, takes arguments representing the coordinates, it doesn't matter what positions these arguments have in the routine, and what others it may have. Just leave the x and y arguments open when you connect the routine to the interface.

This ability to plug elements of the Business Model directly into the user interface is one of the principal attractions of the agent model.

One of the uses of command classes is to support **undoing and redoing** in an interactive system. It is easy to see how to provide this through agents too: just pass *two* agents, one representing the “do” operation and the other representing the “undo”. This technique — whose details the reader is invited to spell out — is used in many of ISE's interactive products supporting undo and redo.

Integrating a function



The next set of examples was about integration. We assumed functions

$$\begin{aligned} g(x: \text{REAL}): \text{REAL} \\ h(x: \text{REAL}; a: T1; b: T2): \text{REAL} \end{aligned}$$

and wanted to integrate them over a real interval such as $[0, 1]$, that is to say, approximate the two integrals

$$\int_{x=0}^{x=1} g(x) dx \qquad \int_{x=0}^{x=1} h(x, u, v) dx$$

We declare

$$\text{your_integrator: INTEGRATOR}$$

and, with the proper definition of function *integral* in class *INTEGRATOR*, to be seen shortly, we will obtain the integrals through the expressions



```
your_integrator.integral (agent g (?), 0.0, 1.0)
your_integrator.integral (agent h (? , u, v), 0.0, 1.0)
```

The question mark indicates, in each case, the open argument: the place where *integral* will substitute various real values for *x* when evaluating *g* or *h*.

Note that if we wanted in class *D* to integrate a real-valued function from class *REAL*, such as *abs* which is declared in *REAL* as

```
abs: REAL
    -- Absolute value
do ... end
```

we would obtain it simply through



```
your_integrator.integral (agent {REAL}.abs, 0.0, 1.0)
```

Let us now see how to write function *integral* to make all these uses possible. We use a primitive algorithm — this is not a treatise on numerical methods — but what matters is that any integration technique will have the same overall form, requiring it to evaluate *f* for various values in the given interval. Here class *INTEGRATOR* will have a real attribute *step* representing the integration step, with an invariant clause stating that *step* is positive. Then we may write *integral* as:

The boxed expression is where the algorithm needs to evaluate the function *f* passed to *integral*. Remember that *item*, as defined in class *FUNCTION*, calls the associated function, substituting any operands (here *x*) at the open positions, and returning the function's result. The argument of *item* is a tuple (of type *OPEN*, the second generic parameter of *FUNCTION*); this is why we need to enclose *x* in brackets, giving a one-argument tuple: [*x*].

In the first two example uses, **agent g (?)** and **agent h (? , u, v)**, this argument corresponds to the question mark operands to *g* and *h*. In the last example the call expression passed to *integral* was **agent {REAL}.abs**, where the open operand is the target, represented by {*REAL*}, and successive calls to *item* in *integral* will substitute successive values of *x* as targets for evaluating *abs*.

In the case of *h*, the closed operands *u* and *v* are evaluated at the time of the evaluation of the expression **agent h (? , u, v)**, and so they remain the same for every successive call to *item* within a given execution of *integral*.



```

integral
(f: FUNCTION [ANY, TUPLE [REAL], REAL];
 low, high: REAL): REAL
  -- Integral of f over the interval [low, high]
require
  meaningful_interval: low <= high
local
  x: REAL
do
  from
    x := low
  invariant
    x >= low ; x <= high + step
    -- Result approximates the integral over
    -- the interval [low, low.max (x - step)]
  until x > high loop
    Result := Result + step * f.item ([x])
    x := x + step
  end
end

```

Note the type *FUNCTION* [*ANY*, *TUPLE* [*REAL*], *REAL*] declared in *integral* for the argument *f*. It means that the corresponding actual must be a call expression describing a function from any class (hence the first actual generic parameter, *ANY*) that has one open operand of type *REAL* (hence *TUPLE* [*REAL*]) and returns a real result (hence *REAL*). Each of the three example functions *g*, *h* and *abs* can be made to fit this bill through a judicious choice of open operand position.

Iteration examples

The next set of initial examples covered iteration. In a class *CC* we want to manipulate both a list of integers and a list of employees



```

intlist: LINKED_LIST [INTEGER]
emplist: LINKED_LIST [EMPLOYEE]

```

and apply the same function *for_all* to both cases:



```

if intlist.for_all (agent is_positive (?)) then ... end
if intlist.for_all (agent over_threshold (?)) then ... end
if emplist.for_all (agent {EMPLOYEE}.is_married) then ... end

```


The function *for_all* is one of the iterators defined in class *TRAVERSABLE* of EiffelBase, and available as a result in all descendant classes describing traversable structures, such as *TREE* and *LINKED_LIST*. This boolean-valued function determines whether a certain property holds for every element of a sequential structure. The property is passed as argument to *for_all* in the form of a call expression with one open argument.

Our examples use three such properties of a very different nature. The first two are functions of the client class *CC*, assessing properties of their integer argument. The result of the first depends only on that argument:



```
is_positive (i: INTEGER): BOOLEAN
    -- Is i positive?
    do Result := (i > 0) end
```

Alternatively the property may, as in the second example, involve other aspects of *CC*, such as an integer attribute *threshold*:



```
over_threshold (i: INTEGER): BOOLEAN
    -- Is i greater than threshold?
    do Result := (i > threshold) end
```



Here *over_threshold* compares the value of *i* to a field of the current object. Surprising as it may seem at first, function *for_all* will work just as well in this case; the key is that the call expression **agent** *over_threshold* (?), open on its argument, is closed on its target, the current object; so the agent object it produces has the information it needs to access the *threshold* field.

In the third case, the argument to *for_all* is **agent** *{EMPLOYEE}.is_married*; this time we are not using a function of *CC* but a function *is_married* from another class *EMPLOYEE*, declared there as



```
is_married: BOOLEAN is do ... end
```

Unlike the previous two, this function takes no argument since it assesses a property of its target; We can still, however, pass it to *for_all*: it suffices to make the target open.

The types of the call expressions are the following:

```
PREDICATE [CC, TUPLE [INTEGER]]
    -- In first two examples (is_positive and over_threshold)

PREDICATE [EMPLOYEE, TUPLE [EMPLOYEE]]
    -- In the is_married example
```

This assumes again that CC is non-generic, so that it is both a class and a type. Remember that a PREDICATE is a FUNCTION with a BOOLEAN result type.

You may also apply *for_all* to functions with an arbitrary number of arguments, as long as you leave only one operand (target or argument) open, and it is of the appropriate type. You may for example write the expressions



```
intlist .for_all (agent some_criterion (e1, ?, e2, e3)
emplist .for_all (agent {EMPLOYEE}.some_function (e4, e5))
```

assuming in *CC* and *EMPLOYEE*, respectively, the functions

```
some_criterion (a1: T1; i: INTEGER; a2: T2; a3: T3)      -- In CC
some_function (a4: T4; a5: T5)                        -- In EMPLOYEE
```

for arbitrary types *T1*, ..., *T5*. Since operands *e1*, ..., *e5* are closed in the calls, these types do not in any way affect the types of the call expressions, which remain as above: *PREDICATE* [*CC*, *TUPLE* [*INTEGER*]] and *PREDICATE* [*EMPLOYEE*, *TUPLE* [*EMPLOYEE*]].

Let us now see how to write the iterator mechanisms themselves, such as *for_all*. They should be available in all classes representing traversable structures, so they must be introduced in a high-level class of EiffelBase, *TRAVERSABLE* [*G*]. Some of the iterators are unconditional, such as



```
do_all (action: ROUTINE [ANY, TUPLE [G]])
    -- Apply action to every item of the structure in turn.
require
    ... Appropriate preconditions ...
do
    from start until off loop
        action .call ([item])
    forth
    end
end
```

This uses the four fundamental iteration facilities, all declared in the most general form possible as deferred features in *TRAVERSABLE*: *start* to position the iteration cursor at the beginning of the structure; *forth* to advance the cursor to the next item in the structure; *off* to tell us if we have exhausted all items (**not** *off* is a precondition of *forth*); and *item* to return the item at cursor position.

Descendants of TRAVERSABLE effect these features in various ways to provide iteration mechanisms on lists, hash tables, trees and many other structures.

The argument *action* is declared as *ROUTINE [ANY, TUPLE [G]]*, meaning that we expect a routine with an arbitrary base type, with an open operand of type *G*, the formal generic parameter of *TRAVERSABLE*, representing the type of the elements of the traversable structure. Feature *item* indeed returns a result of type *G* (representing the element at cursor position), so that it is valid to pass as argument the one-argument tuple [*item*] in the call *action.call ([item])* that the loop repeatedly executes.



We normally expect *action* to denote a procedure, so its type could be more accurately declared as *PROCEDURE [ANY, TUPLE [G]]*. Using *ROUTINE* leaves open the possibility of passing a function, even though the idea of treating a function as an action does not conform to the Command-Query Separation principle of the Eiffel method.

Where *do_all* applies *action* to all elements of a structure, other iterators provide conditional iteration, selecting applicable items through another call expression argument, *test*. Here is the “while” iterator:



```

while_do
  (action: ROUTINE [ANY, TUPLE [G]]
   test: PREDICATE [ANY, TUPLE [G]])
  -- Apply action to every item of structure up to,
  -- but not including, first one not satisfying test.
  -- If all satisfy test, apply to all items and move off.
  require
    ... Appropriate preconditions ...
  do
    from start until
      off or else not action.test ([item])
    loop
      action.call ([item])
    forth
  end
end

```

Note how the algorithm applies *call* to *action*, representing a routine (normally a procedure), and *item* to *test*, representing a boolean-valued function. In both cases the argument is the one-element tuple [*item*].

The iterators of *TRAVERSABLE* cover common control structures: *while_do*; *do_while* (same as *while_do* but with “test at the end of the loop”, that is to say, apply *action* to all items up to and including first one satisfying *test*); *until_do*; *do_until*; *do_if*.

Yet another iterator of *TRAVERSABLE* is *for_all*, used in earlier examples. It is easy to write a *for_all* loop algorithm similar to the preceding ones. Here is another possible definition, in terms of *while_do*:



```

for_all (test: PREDICATE [G, TUPLE [G]]): BOOLEAN
    -- Do all items satisfy test?
    require
        ... Appropriate preconditions ...
    do
        while_do (agent nothing (?), test)
        Result := off
    end
  
```

using a procedure *nothing* (*x*: *G*) which has no effect (but needs an argument *x* for typing reasons, since the first argument of *while_do* must be of type *ROUTINE* [ANY, TUPLE [G]]). It is trivial to define *nothing* in terms of procedure *do_nothing*, from class *ANY*. We apply *nothing* as long as *test* is true of successive items; if we find ourselves *off*, we return true; otherwise we have found an element not satisfying the *test*.

→ *do_nothing* is cited in 35.6, page 930.

It is possible to avoid defining a procedure *nothing* by using an inline agent.

Assuming a proper definition of *do_until*, the declaration of *exists*, providing the second basic quantifier of predicate calculus, is nicely symmetric with *for_all*:



```

exists (test: PREDICATE [G, TUPLE [G]]): BOOLEAN
    -- Does at least one item satisfy test?
    require
        ... Appropriate preconditions ...
    do
        do_until (agent nothing (?), test)
        Result := not off
    end
  
```

27.7 TWO ADVANCED EXAMPLES

Before moving on to the last details of the agent mechanism, let's gain further appreciation for its power and versatility by looking at two interesting applications, error processing and "once per object" (followed in the next section by examples of the inline form).

Error processing without the mess

The first example addresses a frequent situation in which we perform a sequence of actions, each of which might encounter an anomaly that prevents continuing as hoped. The problem here is that it's difficult to avoid a complex, deeply nested control structure, since we may have to get out at any step. The straightforward implementation will look like this:

```

action1
if ok1 then
    action2
    if ok2 then
        action3
        ... More processing, more nesting ...
    end
end

```

For example we may want to do something with a file of name *path_name*. We first test that that *path_name* is not void. Then that the string is not empty. Then that the directory exists. Then that the file exists. Then that it is readable. Then that it contains what we need. And so on. A negative answer at any step along the way must lead to reporting an error situation and aborting the whole process.

The problem is not so much the nesting itself; after all, some algorithms are by nature complex. But often the normal processing is not complicated at all; it's the error processing that messes everything up, hiding the "useful" processing in a few islands lost in an ocean of error handling. If the error processing is different in each case (**not *ok1***, **not *ok2*** and so on) we can't do much about it. But if it is always of the form: "Record the error source and terminate the whole thing", then the above structure may seem too complicated. Although we may address this issue through exceptions, they are often overkill.

An agent-based technique is useful in some cases. It assumes that you write the various actions — *action1* ... *action3* above — as procedures, each with a body of the form

```

...Try to do what's needed...
controlled_check (execution_ok, "...Appropriate message...")

```

with *execution_ok* representing the condition that must be satisfied for the processing to continue. Then you can rewrite the processing above as just:

```

controlled_execute ([
  agent action1,
  agent action2 (...),
  agent action3 (...)
])

if controlled_glitch then
  warning (controlled_glitch_message)
  -- Procedure warning is an error reporting mechanism
end

```

This linear structure is much simpler than the original.

The features whose names start with *controlled_* come from the EiffelBase class *CONTROLLED_EXECUTION*, of which the class containing the above scheme should be a descendant. These procedures are not difficult to write; for example *controlled_check* sets *controlled_glitch* and *controlled_glitch_message*, and *controlled_execute* looks like this:

The routine as it appears in the library has a few extra instructions to record the glitch step and, on option, raise an exception.

```

controlled_execute
  (actions: ARRAY [PROCEDURE [ANY, TUPLE]])
  -- Execute actions, stopping if encountering a glitch.
  local
    i: INTEGER
  do
    from
      controlled_glitch := False; i := actions.lower
    until i > actions.upper or else controlled_glitch loop
      actions.item (i).call ([ ])
      i := i + 1
    end
  end

```

Once per object

The second example, also supported by an EiffelBase class, provides a “once per object” mechanism.

You know, of course, Eiffel’s “once routines”, executed only once per system execution. They define a “once per class” mechanism: all instances of a class share the result of a once function. (All these concepts are applicable to procedures, but for this discussion we restrict ourselves to functions.) Now assume you need functions that compute a result specific to each instance of the class, and computed just once for that instance, the first time it’s requested — if at all.

← For an introduction to once routines see [“ROUTINE BODY”](#), 8.5, page 222.

A typical application would be large pieces of information associated with objects of a certain type, but stored in a database; for example each instance of a class *COMPANY* may have *stock_history* information, of type *HISTORY*, which may be huge. We only want to retrieve the information on demand; given the size of the information and the number of instances of the class, it is not acceptable to load everything ahead of time. Even if an instance of *COMPANY* is in memory, we want to retrieve the associated *HISTORY* from the database only when and if we need access to the company's *stock_history*.

Agents provide us with a general solution to all problems of this kind. In class *COMPANY* you will simply declare

```
stock_history: ONCE_PER_OBJECT [HISTORY]
```

and obtain the value, when and if needed, as

```
stock_history.item (agent retrieved_history)
```

Here *retrieved_history* is the function that computes the needed result — the one that you want to call once for each object. That's all you have to do! Note that this scheme allows you to have as many “once per object” functions as you like in any given class. It relies on a general-purpose EiffelBase class *ONCE_PER_OBJECT* of the following form:

```
expanded class
  ONCE_PER_OBJECT [G]
feature -- Access
  item (f: FUNCTION [ANY, TUPLE, G]): G
    -- Value of f, computed once for each object;
    -- subsequent calls return same value for same object.
    do
      if not computed then
        internal_result := f.item ([])
        computed := True
      end
      Result := internal_result
    end
feature {NONE} -- Implementation
  computed: BOOLEAN
    -- Has item already been requested?
  internal_result: G
    -- Result, if already computed
end
```

27.8 USING INLINE AGENTS

The agents seen so far are of the `Call_agent` kind, relying on class features, such as `f` and `g` (integration examples), `integer_property` and `is_married` (iterator examples), `compute_stats` (EiffelVision example) and others. *agent is_positive means the same as agent is_positive (?).*

Sometimes, the *only* reason for writing a certain computation is to define an agent from it. To avoid adding a feature that will make the enclosing class more complicated, you may write the algorithm within the agent. The syntactical construct for this **inline** case, previewed at the beginning of this chapter, mirrors the definition of a routine — although, like any other agent construct, it is syntactically an expression. Here are some examples of inline agents, all to be used as expressions::



```
(agent (i: INTEGER): BOOLEAN do Result := is_positive (i) end)
    -- Equivalent to agent is_positive (?)

(agent (e: EMPLOYEE): BOOLEAN do Result := e.is_married end)
    -- Equivalent to agent {EMPLOYEE}.is_married

(agent (e, f: EMPLOYEE): BOOLEAN
  do Result := (e.salary > f.salary) end)

(agent (e, f: EMPLOYEE; p: POSITION): BOOLEAN
  do Result := (e.job = p) and (f.job = p)) end)
```

As noted in the comments, the first two of these examples have `Call_agent` equivalents, since they directly rely on existing routines of some class. But in the last two cases, there are no such routines.

The third agent (for example) denotes an object representing a boolean-valued operation that, for two objects of type `EMPLOYEE`, returns true if and only if the query `salary` yields a higher result for the first than for the second.

It is still possible to use a `Call_agent` in these cases, but this requires adding features to the enclosing class:



```
higher_salary (e, f: EMPLOYEE): BOOLEAN
    -- Does e have a higher salary than f?
  do
    Result := (e.salary > f.salary)
  end

same_job (e, f: EMPLOYEE; pos: POSITION): BOOLEAN
    -- Do e and f both have position pos?
  do
    Result := ((e.job = pos) and (f.job = pos))
  end
```


to enable rewriting the calls as **agent** *higher_salary* (abbreviating, as usual, **agent** *higher_salary* (?, ?)) and **agent** *same_job*. But if the only use of the given little algorithms is to define the corresponding agents, for example to pass them to some iterators, then you may want to avoid burdening the enclosing class with such routines, using inline agents instead.

The inline agents shown so far denote functions (*FUNCTION* or *PREDICATE*). Here is an example that passes an inline procedure agent to an iterator, to raise by 50 percent the salary of every employee called “Tina”:



```
emplist.do_all
  ((agent e: EMPLOYEE)
   require
     employee_exists: e /= Void
   do
     if equal (e.first_name, once "Tina") then
       e.set_salary (1.5 * e.salary)
     end
   end))
```

→ Defining the string as **once** is not strictly necessary but improves performance by avoiding repeated evaluations; see “[Basic manifest strings](#)”, page 796

The **require ... do ... end** part is a specimen of **Routine**; an inline agent indeed uses exactly the same Routine construct as the declaration of a routine in a class; so it can have all the applicable clauses, such as **Precondition** here, but also **Local_declarations**, **Postcondition** and **Rescue**.

← “[FEATURE BODIES](#)”, 5.11, page 143.

We can use an inline agent to simplify the earlier definition of *for_all* ← [Page 742](#) in terms of *while_do*, which required a function *nothing* (*x: G*) because *do_nothing* from *ANY*, with no argument, has the wrong signature. An inline agent avoids this:



```
for_all (test: PREDICATE [G, TUPLE [G]]): BOOLEAN
  -- Do all items satisfy test?
  do
    while_do ((agent (x: G) do do_nothing end), test)
    Result := off
  end
```

Inline agents do not give us anything fundamentally new, since we can always use call agents instead. They are useful if you want to avoid features such as *same_job* and *nothing* whose only purpose is to define agents.

The *methodological advice* is clear: if the computation becomes complex, it is usually better to add a feature to the class. The agent passed as argument to *do_all* in the last example is already complex enough to justify writing a separate function instead.

The inline form is particularly useful to express advanced contract specifications. Here is an example. Assume that in a class describing sequential structures (such as *LIST [G]* in EiffelBase) you write a procedure that appends an element. It might include this postcondition:



PURPOSE



METHOD CONTRACT



PURPOSE



```

extend (x: G)
  -- Add x at end; keep other items
  require
  ...
  do
  ...
  ensure
    one_more: count = old count + 1
    added_at_end: item (count) = x
    others_unchanged:
      (1 |..| old count).for_all
        ((agent (i: INTEGER): BOOLEAN
          do Result := equal (item (i), (old twin). item (i)) end))
  end

```

In the last postcondition clause — the one of interest for this discussion — *1 |..| old count* is the interval from 1 to **old count**, to whose items *for_all* applies the agent property on the next line. The property expresses that the item at position *i*, for arbitrary *i*, is equal to the original item at that position (more precisely, to the item at position *i* in **old twin**, a copy of the list taken on entry to the procedure). This is typical of how agents enable us to express non-trivial postcondition or invariant properties, stating that a whole set of items have not changed, or have a certain association with the corresponding set of items in another structure.

We could restate the inline agent (the argument to *for_all*) in non-inline form as **agent** *equal_item* (**old twin**, ?), but this assumes a function



```

equal_item (l: like Current; i: INTEGER): BOOLEAN
  -- Is item at position i equal to corresponding one in l?
  do
    Result := (item (i) = l.item (i))
  end

```

If you want to specify your software completely — expressing not only straightforward properties such as *item* (*count*) = *x*, but also those involving entire substructures — you may end up writing many such functions. Although they add interesting information, one may also feel that, being only used for assertions, they needlessly complicate the class. They may destabilize the software since any effort at better specification may cause the addition of a whole set of new features, used only in the assertions and of no other interest to clients of the class. Inline agents solve this problem.



Here is another example application. The agents described in this chapter represent delayed *calls*; you may have wondered whether we also need an expression construct to denote delayed *object creation*, perhaps something like **agent create** {*SOME_TYPE*} .*make* (*a1*,?). The answer is no, since we can achieve the intended effect (assuming we need it) by using a creation expression as part of an inline agent in



```
(agent (b1: B) do create {SOME_TYPE} .make (a1, b1) end)
```

where *B* is the type of *make*'s second argument.

You may view inline agents as **anonymous routines**, similar to anonymous *classes* (tuple types) and anonymous *objects* (tuples). This is particularly clear in the **Routine** case (...) ... **do ... end**, which has exactly the same form as a routine declaration:

```
r (...) is ... do ... end
```

(with, as noted, the possibility of including all relevant clauses, such as precondition, postcondition, rescue, local variable declarations). The only difference is that the inline agent doesn't use a routine name *r* — it doesn't need one. When such a routine is used with the sole purpose of being passed as argument to a routine expecting an agent, the anonymous form avoids cluttering the class with a full-status routine.

27.9 ACCESSING FEATURE PROPERTIES

Class **ROUTINE** and its descendants provide a starting point for many of the introspection needs that Eiffel applications may need.

The first introspection mechanism is a simple way, through class **ROUTINE** and its descendants, to gain access to the precondition and postcondition of a routine:

```
precondition (args: OPEN): BOOLEAN
    -- Do args satisfy routine's precondition in present state?

postcondition (args: OPEN): BOOLEAN
    -- Does current state satisfy routine's postcondition
    -- for operands args?
```

This enables you to check the precondition before you apply an agent, as in



```
if your_agent.precondition (your_operands) then
    your_agent.call (your_operands)
end
```

where *your_agent* is an agent expression and *your_operands* is a valid tuple of operands for that agent.

There is, as will be seen next, a similar facility for class invariants.

27.10 THE BASE CLASS AND TYPE



Introspection support is also one of the concerns behind the first generic parameter of *ROUTINE*, *PROCEDURE*, *FUNCTION* and *PREDICATE*. The specification

```
ROUTINE [BASE, OPEN → TUPLE]
```

includes, as first generic parameter, the type *BASE* representing the type (class with generic parameters) to which an agent's feature belongs. This is the type of the target expected by the feature.

→ For an inline agent, the agent's feature is its "associated feature"; see page 755.

The examples seen so far do not use *BASE* at all, because procedure *call* does not need it. If the agent is closed on its target, as in

```
y := agent a0.f(a1, ?, ?)
```

then it includes, here through *a0*, the target information that a later call to *call* may require. In the other case — open target — as in

```
t := agent {T0} .f(a1, a2, ?)
```

then the target type is specified, here *T0*, and provides the information needed to determine the right version of *f*. In this case the *BASE* generic parameter is in fact redundant, since it is identical to the first component of the tuple type corresponding to *OPEN*; the type of *t*, for example, is

```
ROUTINE [T0, TUPLE [T0, T3]]
```

where the two tuple components correspond to the two open operands: the target, and the last argument.

In both the closed target and open target cases, then, we don't need the *BASE* generic parameter if all we do with agents is execute *call* on them.

BASE is useful for other purposes. Without *BASE* a call closed on its target, as with *y* above, could not contain any information about the class (and associated type) where the call's associated feature is defined. To open the gate to full *introspection* services — enabling a system to explore its own properties — class *ROUTINE* uses a feature

```
base_type: TYPE [BASE]
```

that yields the type to which the agent's feature belongs. Class *TYPE* [*G*] from the Kernel Library provides information about a type *G* and its base class.

Class *TYPE* is, even more fundamentally than *ROUTINE* and its heirs, the starting place for introspection. Example features include:

- *name*: *STRING*, the upper name of the type's base class.



Syntax of call agents

We have encountered numerous examples of `Call_agent`, such as

First three examples unqualified, last one qualified.



```
agent f(a1, a2, a3)
agent f
agent {T0}.f(a1, ?, ?)
```

A `Call_agent` starts with the keyword `agent`. The part that follows, called a `Call_agent_body`, closely resembles a `Call`; we can't just use that earlier construct, however, since we must allow for the question mark and brace conventions, which have no equivalent in normal calls: ← *The syntax for `Call` was on page 626.*



Call agent bodies

```
Call_agent_body ≙ Agent_qualified | Agent_unqualified
Agent_qualified ≙ Agent_target "." Agent_unqualified
Agent_unqualified ≙ Feature_name [Agent_actuals]
Agent_target ≙ Entity | Parenthesized | Manifest_type
Agent_actuals ≙ "(" Agent_actual_list ")"
Agent_actual_list ≙ {Agent_actual "," ... }+
Agent_actual ≙ Expression | Placeholder
Placeholder ≙ [Manifest_type] "?"
```

--- FIX ---`Manifest_type` in the `syntax` of `Multi_branch` as

← *Page 486.*

An `Agent_target` may be of three kinds: `Entity`, `Parenthesized` and `Manifest_type`. The third (used in the last example) enables you to specify an open target by listing a type in braces. For an actual argument, you can use, besides an actual value, a `Placeholder` (question mark).

The possibility of using a `Manifest_type` or `Placeholder` to specify open operands is the principal difference between agent calls and normal calls. There is another difference, not immediately obvious from the syntax. In a `Call_agent` as well as a normal `Call`, the argument list, `Agent_actuals`, is optional. But omitting it doesn't have the same effect. If `f` is declared as having one or more arguments, a call of the form `a.f`, or its unqualified variant `f`, are invalid since they violate the Argument rule: you must always specify actual arguments, as in `a0.f(a1, a2, a3)`. For an agent call, however, corresponding forms such as ← *The Argument rule was on page 634.*

```
a0.f
f
{T0}.f
```

are valid; they are simply convenience abbreviations to indicate that all arguments are open, meaning respectively the same as

```
a0.f(?, ?, ?)
f(?, ?, ?)
{T0}.f(?, ?, ?)
```

You may in such a case omit the argument list, to indicate that all arguments (if any) are open. The Agent Call rule, introduced later in this chapter, explicitly allows this. It causes no ambiguity and (unless you prefer a fully explicit style) lets you avoid cluttering your class text with question marks.

→ Page 754 (see the explanation about clause 3 of the rule).

A final note on call agent syntax. You may build a **Call agent** not only from identifier features as in these examples, but also from an operator or bracket feature. Just designate the feature by its identifier, in conformance with the Feature Identifier principle: with a feature *your_name* **alias** "\$" for some binary operator §, use the identifier to build agent expressions such as

← “Feature Identifier principle”, page 153.



```
agent a.your_name (b)      -- All closed
agent your_name (?)       -- Open on argument
agent {T}.your_name (?)   -- Open on target and argument
```

and similarly with a feature *other_name* **alias** "[]".

Syntax of inline agents

We have seen that an **Inline agent** is like a routine declaration, but given inline, without a name, as in



```
(agent (e, f: EMPLOYEE): BOOLEAN
  -- Is the cumulated salary of e and f higher than threshold?
  require
    first_exists: e /= Void
    second_exists: e /= Void
  local
    salary_sum: REAL
  do
    salary_sum := e.salary + f.salary
    Result := (salary_sum > threshold)
  end)
```

This is reflected by the syntax given on the previous page, which specifies:

- An optional **Formal_arguments** list, as *(e, f: EMPLOYEE)*.
- An optional **Type_mark**, as in *: BOOLEAN*. If this part is present, the associated routine is a function; otherwise it is a procedure.
- A **Routine**, with all the possible trappings, including Precondition, Local_declarations,



As already noted, it is not recommended to have such extensive computations in inline agents: after all, an **Agent** is an expression, meant for example to be passed as argument to a routine. But this is just methodological advice; the whole Routine syntax is available if you wish to use it, including the optional **Precondition**, **Local_declarations**, **Postcondition** and **Rescue** clauses; even **Header_comment** and **Obsolete**. The only restriction (stated in the validity constraint given next) is that the routine must not be deferred.

27.12 AGENT VALIDITY



We may now add the validity rules. It is convenient to deal separately with **Call_agent** and **Inline_agent** cases.

Like the previous one, this section is not essential on first reading.

Validity of call agents

For call agents, it is useful first to define the notion of target type:



Target type of an call agent

The **target type** of a **Call_agent** is:

- 1 • If there is no **Agent_target**, the current type.
- 2 • If there is an **Agent_target** and it is an **Entity** or **Parenthesized**, its type.
- 3 • If there is an **Agent_target** and it is a **Manifest_type**, the type that it lists (in braces).

← The “current type” is the enclosing class, with generic parameters added if necessary to make up a type. See [“CURRENT TYPE, FEATURES OF A TYPE”, 12.11, page 365.](#)

The validity rule follows:



Call Agent rule

VPCA

A **Call_agent** involving a **Feature_name** fn , appearing in a class C , with target type $T0$, is valid if and only if it satisfies the following conditions:

- 1 • fn is the name of a feature f of $T0$.
- 2 • If there is an **Agent_target**, f is export-valid for $T0$ in C .
- 3 • If the **Agent_actuals** part is present, the number of elements in its **Agent_actual_list** is equal to the number of formals of f .
- 4 • Any **Agent_actual** of the **Expression** kind is of a type compatible with the type of the corresponding formal in f .

--- REMOVED CONDITION ON SEPARATE --- The rule’s phrasing makes certain forms of the construct automatically valid:

- If any **Agent_actual** is of the **Placeholder** kind, represented simply by a question mark, clause 4 does not apply, so the argument raises no type validity problem. This is as expected, since such an argument is left open for future filling-in.

- If there is no **Agent_actuals** part, clauses 3 and 4 do not apply. If f has no formals, we are calling an argumentless feature with no actuals, as we should. If f has one or more formal arguments, we view the absence of explicit actuals of an abbreviation for actuals that are all of the **Placeholder** kind (question marks): assuming f takes three arguments, **agent** $a0.f$ is simply an abbreviation for **agent** $a0.f(?, ?, ?)$. In this case the implicit arguments are all open, and hence automatically valid.

Clause 3 differs from its counterpart for normal calls, which required actual argument list to match the formal list if any. Instead we explicitly allow omitting actuals altogether, to signify that all arguments are open. ← “Argument rule”, page 634.

---- Clause ---- is a consistency condition for concurrent computation, and parallels a similar clause discussed in the chapter on normal calls.

Validity of inline agents

To define the validity of inline agents (also their semantics), it is convenient to consider this case as equivalent to the previous one, **Call_agent**, by treating any inline agent as equivalent to **agent** $f(\dots)$ where f is a fictitious routine added to the class. Here is the definition of this equivalence:



Associated feature of an inline agent

Every inline agent ia of a class C has an **associated feature**, defined as a fictitious routine f of C , such that:

- 1 • The name of f is chosen not to conflict with any other feature name in C and its descendants.
- 2 • The formal arguments of f are those of ia .
- 3 • f is secret (available for call to no class).
- 4 • The **Attribute_or_routine** part of f is defined by the **Attribute_or_routine** part of ia .
- 5 • f is a function if ia has a **Type_mark** (its return type being given by the **Type** in that **Type_mark**), a procedure otherwise.

← “Exported, selectively available, secret”, page 211.

Clause 2 lists, as arguments to f , not only the arguments to the inline agent but also the local variables of the enclosing routine. The local variables will indeed serve as *closed* arguments; this will be specified in the semantics given in the next section.

The validity rule follows:



Inline Agent rule *VPIA*

An **Inline_agent** a of associated feature f , is valid in the text of a class C if and only if it satisfies the following conditions:

- 1 • f , if added to C , would be valid.
- 2 • f is not deferred.

There is no other condition, since in particular The **Routine** part must be valid on its own; in particular, the Entity rule states that any entity appearing in the **Agent_body** must be a formal argument of the inline agent itself, such as *other* and *i* in

(*other*: **like** *Current*; *i*: **INTEGER**)
do *Result* := (*item* (*i*) = *other.item* (*i*)) **end**

or a local variable of an enclosing agent or routine, or a feature of the enclosing class.

Here are some properties following from the Inline Agent rule:



Inline Agent Requirements

VPIR

An **Inline_agent** *a* must satisfy the following conditions:

- 1 • No formal argument or local variable of *a* has the same name as a feature of the enclosing class.
- 2 • Every entity appearing in the **Routine** part of *a* is the name of one of: a formal argument of *a*; a local variable of *a*; a feature of the enclosing class; **Current**.
- 3 • The **Feature_body** of *a*'s **Routine** is not of the **Deferred** form.

These conditions are stated as another validity rule permitting compilers to issue more understandable error messages. It is not in the usual “if and only if” form (since the preceding rule, the more official one, takes care of this), but the requirements given cover the most obvious possible errors.

27.13 AGENT SEMANTICS



(Like the previous two, this section may be skipped on first reading.) The final part of the specification addresses the semantics of agents. It is organized in three parts:

- Call-agent equivalent of an inline agent (enabling the next two parts to restrict themselves to the **Call_agent** part).
- Open and closed operands.
- Type and value of an agent.

Call-agent equivalent of an inline agent

To define the validity of an inline agent *a*, it was convenient to define its associated feature. Then *a* itself can be viewed as if it were a **Call_agent**:

DEFINITION

Call-agent equivalent of an inline agent

The **call-agent equivalent** of an inline agent *ia* is the **Call_agent agent *f*** where *f* is the associated feature of *ia*.

This allows a simple specification for the semantics of inline agents:

Semantics of inline agents

The semantic properties of an inline agent are those of its call-agent equivalent.

Thanks to this rule, we can focus on call agents when defining the type and execution effect of agents.

Note how the formal arguments and local variables of the enclosing routine if any, and of any enclosing agents, serve as closed arguments to the agent. In reading earlier discussions of inline agents, you may have pondered two as yet unanswered questions:

- Is it permitted for an inline agent to refer to a local variable of the enclosing routine, and, if so, what does that mean?
- Call agents may have both closed and open operands. We have seen how to give an inline agent open operands: just specify them as arguments to the agent. But is there a way to give it closed operands too?

The rule just given answers both questions at once by giving a status to local variables of the enclosing routine: treat them as closed operands.

So a routine of the form

```
r
  local
    n: INTEGER
  do
    n := 1
    your_list.do_all ((i: INTEGER) do print (i + n) end)
  end
```

will print the successive values in *your_list* (assumed to be of type *LIST [INTEGER]*) all incremented by 1, the value of the local variable *n* at construction time. As specified by the last rule, this is the same effect as if the call were

```
do_all (agent print_incremented_value (?, n)
```

where *print_incremented_value* is the “fictitious routine” introduced by the definition of “associated feature” of an inline agent:

```
print_incremented_value (i: INTEGER; n: INTEGER)
do
    print (i + n)
end
```



In examining the above definition of call-agent equivalent, note that the validity rule on inline agents guarantees that there can be no name clash between the formal arguments and local variables of any enclosing agents and of the enclosing routine if any. (Nesting inline agents doesn't seem a desirable use of the mechanism, but no rule disallows it.)

← “*Inline Agent rule*”, page 755.

The semantics of inline agents also requires a specific rule on the meaning of *Result*. An inline agent may be embedded in a function of the class, or even in another function agent, causing a potential ambiguity. We decide that *Result* always refers to the result of the innermost agent:



Use of Result in an inline function agent

In an agent of the *Inline_agent* form denoting a function, the local variable *Result* denotes the result of the agent itself.

← “*Open operand position*” was defined on page 759.



This is a rather specific case and another approach would be to disallow function agents within functions or other function agents, or to use a special notation to remove the ambiguity. The rule as given seems preferable. If you need to refer to an outer *Result*, you may assign its value to a local variable and use that local variable in the innermost agent scope. This causes a little extra work, but only in a rare and special case.

Open and closed operands

It is useful to define precisely what “open” and “closed” mean for the operands of an agent expression:



Open and closed operands

The **open operands** of a *Call_agent* include:

- 1 • Any *Agent_actual* that is a *Placeholder*.
- 2 • The *Agent_target* if it is present and is a *Manifest_type*.

The **closed operands** include all non-open operands.

← The operands of a call were defined on page 723 as including its target, and its arguments if any.

From the definition of call-agent equivalent form we deduce that for an inline agent:

- The open operands are the agent's formal arguments, if any.

- The closed operands are the local variables and formal arguments of the enclosing routine and any enclosing agents.

An earlier definition also introduced the notion of *operand position*, which we can now extend to a definition of open and closed positions:

← “Operand position” was defined on page 723: the target position is 0, and the argument positions start at 1.

DEFINITION

Open and closed operand positions

The **open operand positions** of an Agent are the operand positions of its open operands, and the **closed operand positions** those of its closed operands.

Type and value of an agent expression

The preceding definitions enable us to specify the semantics of an agent expression. It suffices to give it for a **Call_agent**:

SEMANTICS

Type of an agent expression

Consider a **Call_agent** a , with a target of type T_0 . Let i_1, \dots, i_m ($m \geq 0$) be its open operand positions, if any, and let T_{i_1}, \dots, T_{i_m} be the types of f 's formal arguments at positions i_1, \dots, i_m (taking T_{i_1} to be T_0 if $i_1 = 0$).

The type of a is:

- **PROCEDURE** $[T_0, \text{TUPLE } [T_{i_1}, \dots, T_{i_m}]]$ if f is a procedure.
- **FUNCTION** $[T_0, \text{TUPLE } [T_{i_1}, \dots, T_{i_m}], R]$ if f is a function of result type R other than **BOOLEAN**.
- **PREDICATE** $[T_0, \text{TUPLE } [T_{i_1}, \dots, T_{i_m}]]$ if f is a function of result type **BOOLEAN**.

SEMANTICS

Agent Expression semantics

The value of an agent expression a at a certain *construction time* yields a reference to an instance **D0** of the type of a , containing information identifying:

- The associated feature of a .
- Its open operand positions.
- The values of its closed operands at the time of evaluation.

Although this will be an implicit consequence of the last rule, it doesn't hurt to state explicitly what some of the information in **D0** is good for: enabling calls on agent objects.



Effect of executing *call* on an agent

Let **D0** be an agent object with associated feature f and open positions i_1, \dots, i_m ($m \geq 0$). The information in **D0** enables a call to the procedure *call*, executed at any **call time** posterior to **D0**'s construction time, with target **D0** and (if required) actual arguments a_{i_1}, \dots, a_{i_m} , to perform the following:

- Produce the same effect as a call to f , using the closed operands at the closed operand positions and a_{i_1}, \dots, a_{i_m} , evaluated at call time, at the open operand positions.
- In addition, if f is a function, setting the value of the query *last result* for **D0** to the result returned by such a call.

← *last result* from class **FUNCTION**, giving the result of the last evaluation, was introduced on page 728.

Expressions

28.1 OVERVIEW

Through the various forms of **Expression**, software texts can include denotations of run-time values — objects and references.

Previous discussions have already introduced some of the available variants of the construct: **Formal**, **Local**, **Call**, **Old**, **Manifest_tuple**, **Agent**. The present one gives the full list of permissible expressions and the precise form of all but one of the remaining categories: operator expressions, equality and locals. The last category, constants, has its own separate presentation, just after this one.

← Chapter 23 on Call, Formal and Local; ““Old” expression”, . . . page 239; chapter 13 on tuples; chapter 27 on agents.

28.2 GENERAL FORM OF EXPRESSIONS

An expression will use one of the following variants:



	Expressions
Expression	\triangleq Basic_expression Special_expression
Basic_expression	\triangleq Read_only Local Call Precursor Equality Parenthesized Old Operator_expression Bracket_expression Creation_expression
Special_expression	\triangleq Manifest_constant Manifest_tuple Agent Object_test Once_string Address
Parenthesized	\triangleq "(" Expression ")"
Address	\triangleq "\$" Variable
Once_string	\triangleq once Manifest_string
Boolean_expression	\triangleq Basic_expression Boolean_constant Object_test

“Basic expressions” correspond to the common forms of expression derived from ordinary mathematical notation, such as variables and operator expressions. “Special expressions” include manifest constants — constants given directly by their values — as well as original Eiffel mechanisms such as agents and tuples.

Summarizing the variants of **Basic_expression**:

- **Read_only**, which includes formal arguments of routines and **Current**, and **Local** variables were seen in the discussion of entities. ← *“Entity, variable, read-only”, page 512.*
 - A **Call**, denoting a feature call, is an **Expression** if and only if the feature of the call is a query, that is to say, an attribute or a function. (Otherwise it would be an **Instruction**.) ← *Chapters 23 for the basics of calls and 25 about their validity.*
- As a special case of **Call**, you may use an attribute *x* as an expression in the text of its class. (In the syntax of Call, just use *x* as **Feature_name** of an **Unqualified_call**, without **Parenthesized_qualifier** or **Actuals**.) ← *Page 626.*
- **Precursor** enables the redefined version of a routine — in the case of expressions, a function — to refer to the original version. We studied the mechanism in connection with inheritance. ← *“ADDING TO INHERITED BEHAVIOR: PRECURSOR”, 10.24, page 299.*
 - **Equality** expressions cover both equality and inequality tests, using the symbols $=$ / \neq , as well as \sim and $/\sim$ for comparing objects. Although they are syntactically similar to the next category, **Operator_expressions**, it is preferable to treat them separately because their semantics, studied in an earlier chapter, is not that of a call. ← *“EQUALITY EXPRESSIONS”, 21.3, page 565.*
 - An **Operator_expression** is built using unary and binary operators. Operator expressions have the semantics of calls but a special syntax, requiring rules of operator precedence.
 - You can use **Parenthesized** subexpressions to override these rules.
 - A **Bracket_expression** again has the semantics of a call but uses bracket syntax, as in *your_table [your_index]*, typically an abbreviation for a feature call *your_table.item (your_index)* where *item* has been declared with a bracket alias: **alias** “[]”.
 - An **Old_expression**, usable only in a postcondition, denotes the earlier value of an expression. This was discussed with assertions. ← *““Old” expression”, page 239.*
 - **Creation_expression** yields a newly created object and was studied in the discussion of creation. ← *--- FILL IN REFERENCE ---*

Special_expression completes this panoply:

- A **Manifest_constant** has a fixed value, given by the form of the constant, as in the **Integer_constant** `-87`. A special case is a **Manifest_string**: we may view as constant because it denotes a fixed string descriptor object, but it gives access to a character sequence which may in fact change. (This is a potential source of confusion and will be explained in detail.) → *“MANIFEST STRINGS”, 29.8, page 794.*

Not all constants are manifest: by declaring a **constant attribute** you may use an **Identifier** to denote a constant value. Syntactically, as noted above, constant attributes used as expressions are a special case of **Call**, but it will be convenient to study them together with manifest constants.

- A **Manifest_tuple** is a tuple given by the list of its elements.
- **Agent** expressions, representing partially evaluated calls, were studied in the previous chapter.
- An **Object_test**, of the form $\{loc: T\} exp$ as studied in the discussion of eradicating void calls, yields true if and only if the expression exp is attached on evaluation to an object of type T , and then binds it locally to the entity loc .
- A **Once_string** is a manifest string constant qualified by the keyword **once** to indicate that if it appears in an expression the string will be evaluated just once, rather than denoting a new string object for each evaluation of the expression. Since this notion is closely tied to the semantics of strings it will be studied in the next chapter as part of the general discussion of strings. Non-once strings — the more common case — are examples of **Manifest_constant**, covered by the previous case.
- **Address** expressions, of the form $\$f$ where f is the name of a feature (or **Current** or **Result**) serve to pass the address of an Eiffel object to non-Eiffel software.

→ “GENERAL FORM OF CONSTANTS”, page 787.

→ “MANIFEST TUPLES”, 29.9, page 809.

→ “THE OBJECT TEST”, 24.3, page 658.

→ “Basic manifest strings”, page 796.

→ “PASSING THE ADDRESS OF AN EIFFEL FEATURE”, 31.8, page 833.

Finally, **Boolean_expression** requires its own construct because a few other constructs — assertion clauses, **Conditional**, **Exit** conditions of loops — specifically expect a boolean expression. **Object_test** is one of the variants.

← Unlabeled_assertion_clause: page 232.
Then_part: page 481.
Exit: page 495.

This chapter needs only occasionally to refer to the variants studied in depth in their own chapters : **Read_only**, **Local**, **Call** (for which we saw how to determine the value and type of a **Call** serving as an expression), **Precursor** (whose validity rule stated under what condition you may use a **Precursor** as an expression), **Equality**, **Agent**, **Old**, **Object_test**, **Address**, **Creation_expression**; for **Call**, we still need to define explicitly the value of a call expression. After introducing the notion of “subexpression”, the following sections will explore the remaining variants in the order of the syntax. Then we’ll explore general properties of expressions, in particular the notion of Equivalent Dot Form, how to determine the type of an expression, and the syntactical benefit (having to do with making semicolons *always* optional) of the distinction between **Basic_expression** and **Special_expression**.

← Clause 5 of “Precursor rule”, page 304.



As a general note on the use of expressions, remember that if you have an expression exp of type U and want to use it as if it were of a type T to which U conforms or converts, you may rely on the notation

← “CONVERTING AN EXPRESSION EXPLICITLY”, 15.9, page 416.

$\{T\} [exp]$

28.3 SUBEXPRESSIONS



This section defines a technical notion; you may skip it on first reading.

For some of the definitions and rules that follow, it is convenient to talk about the “subexpressions” of an expression: all the components of the expression that are themselves expressions whose value participates in the evaluation of the expression as a whole. This notion is mostly useful for operator expressions, but it’s convenient to define it for all other kinds as well:



Subexpression, operand

The **subexpressions** of an expression e are e itself and (recursively) all the following expressions:

- 1 • For a **Parenthesized** (a) or a **Parenthesized_target** ($|a|$): the subexpressions of a .
- 2 • For an **Equality** or **Binary_expression** $a \text{ § } b$, where § is an operator: the subexpressions of a and of b .
- 3 • For a **Unary_expression** $\diamond a$, where \diamond is an operator: the subexpressions of a .
- 4 • For a **Call**: the subexpressions of the **Actuals** part, if any, of its **Unqualified_part**.
- 5 • For a **Precursor**: the subexpressions of its unfolded form.
- 6 • For an **Agent**: the subexpression of its **Agent_actuals** if any.
- 7 • For a **qualified call**: the subexpressions of its **target**.
- 8 • For a **Bracket_expression** $f[a_1, \dots a_n]$: the subexpressions of f and those of all of $a_1, \dots a_n$.
- 9 • For an **Old** expression **old** a .
- 10 • For a **Manifest_tuple** $[a_1, \dots a_n]$: the subexpressions of all of $a_1, \dots a_n$.

In cases 2 and 3, the **operands** of e are a and (in case 2) b .

Clause 4 uses “the **Unqualified_call** part of a **Call**”: both of the available **variants**, **Object_call** and **Non_object_call**, indeed include an **Unqualified_call**.

← Syntax: page [626](#).

For example the subexpressions of $b + (c * [d, e])$ are the whole expression, b , $(c * [d, e])$, c , $[d, e]$, d and e .



Not every expression physically contained in an expression is a subexpression: according to the rule, $a + b + c$ has no subexpression other than itself; neither has $x + y * z$. Parts such as $a + b$, $b + c$, $x + y$, $y * z$, although valid expressions, are not subexpressions. This is because such examples use more than one binary operator in succession, giving rise to potential ambiguities; and indeed the part $x + y$ plays no role in determining the value of $x + y * z$.

The precedence rules which we'll study shortly let us define a *Parenthesized Form* for any expression, such as $x + (y * z)$ in the second part, with subexpressions x and $(y * z)$, both of which participate in the value of the expression as a whole,

In addition to y , z and the whole expression.

28.4 PARENTHESIZED EXPRESSIONS

You may enclose an arbitrarily complex expression in parentheses without changing its semantics:



Parenthesized Expression Semantics

If e is an expression, the value of the **Parenthesized** (e) is the value of e .

Indeed the parentheses have a syntactic role only. You can use a **Parenthesized**:

- To override default operator precedence in operator expressions as studied in the next section.
- To apply a certain operation to an expression when the syntax wouldn't permit it in the original form of the expression.

An important example of the second case is feature application to a complex expression. The syntax of a query call $exp.f$ (or $exp.f(args)$ with arguments) restricts the target exp to just a few possibilities: a single entity, as in $an_attribute.f$, or one or more other calls, as in $g(x).an_attribute.f$. You may not directly apply the feature to a manifest constant, as in $3.f$ (invalid) or to an operator expression, as in $a + b.f$ (which, if valid, would apply f just to b , not to the addition). You can achieve the desired effect by parenthesizing the target expression, as in

← Call and associated constructs, page 626.



$(3).f$
 $(a + b).f$

← Construct **Parenthesized_qualifier**, page 626.

(valid if f is applicable to the respective targets).

28.5 OPERATOR EXPRESSIONS

You may build operator expressions by combining simpler expressions through prefix and infix operators, using parentheses to remove ambiguities if necessary.

Operator expression basics

An example, from the postcondition of procedure *put_child_left* in class *LINKABLE* of EiffelBase, is

```
not (child_position = 2) implies
    child_position = old child_position + 1
```

This appears in class LINKABLE with some extra parentheses for clarity. The effect is the same, however, thanks to the precedence rules.

This uses the infix operators **implies** and **+** and the prefix operator **not**, applied to subexpressions involving **Old** and **Equality**.

Semantically, operator expressions bring nothing new: they are simply a different way to write calls, using conventional operator notation rather than dot notation. Since every feature with an operator alias also has a **Feature_name** (an identifier), you may ignore operators, writing instead calls in dot notation:

```
((child_position = 2).negated).implication
    (child_position = old (child_position.plus (1)))
```

Operator expression syntax

Here is the general form of operator expressions:



Operator expressions

```
Operator_expression ≙ Unary_expression | Binary_expression
Unary_expression ≙ Unary Expression
Binary_expression ≙ Expression Binary Expression
```

← *This syntax appeared originally on page 154.*

Both **Unary** and **Binary** operators can be one of the standard operators, or a “free” operator that you make up according to very flexible rules. The list of standard operators already appeared in the discussion of feature names: → *“Free operator”*, page 893.



Operators

```
Unary ≙ not | "+" | "-" | Free_unary
```

Binary \triangleq "+" | "-" | "*" | "/" | "//" | "\" | "^"
 "<" | ">" | "<=" | ">=" |
 and | or | xor | and then | or else | implies |
 Free_binary

Precedence and Parenthesized Form

The syntax for `Operator_expression` is ambiguous: it would make it possible to understand an expression such as



$$a + b + c * d$$

in several different ways (expressed with parenthesization):

$a + (b + (c * d))$	[1]
$a + ((b + c) * d)$	[2]
$(a + b) + (c * d)$	[3]
$(a + (b + c)) * d$	[4]
$((a + b) + c) * d$	[5]

The correct interpretation, according to the precedence rules given below, is [3].



You can always remove ambiguities by adding parentheses as in these last forms. In mathematical practice, however, it is customary not to require parentheses in simple cases based on “precedence”. This custom makes $a + b * c$ legal and gives it the same meaning as $a + (b * c)$, based on the convention that $*$ “binds tighter” than $+$.

To formalize this practice, we complement the syntax by **precedence rules**. Every possible operator has a precedence, a numerical value between 1 and 13 determined by the table below. The values themselves are not important; what matters is the comparison of the precedence values of any two operators appearing consecutively in an expression. For example, $*$ has precedence 8 and $+$ has precedence 3. In the absence of intervening parentheses, the one with the higher precedence binds tighter.





Operator precedence levels

- 13 **.** (Dot notation, in qualified and non-object calls)
- 12 **old** (In postconditions)
not + -Used as unary
 All free unary operators
- 11 All free binary operators.
- 10 **^** (Used as binary: power)
- 9 *** / // ** (As binary: multiplicative arithmetic operators)
- 8 **+ -** Used as binary
- 7 **..** (To define an interval)
- 6 **= /= ~ /~ < > <= >=** (As binary: relational operators)
- 5 **and and then**
 (Conjunctive boolean operators)
- 4 **or or else xor**
 (Disjunctive boolean operators)
- 3 **implies**(Implicative boolean operator)
- 2 **[]**(Manifest tuple delimiter)
- 1 **;** (Optional semicolon between
 an **Assertion_clause** and the next)

This precedence table includes the operators that may appear in an **Operator_expression**, the equality and inequality symbols used in **Equality** expressions, as well as other symbols and keywords which also occur in expressions and hence require disambiguating: the semicolon in its role as separator for **Assertion_clause**; the **old** operator which may appear in an **Old** expression as part of a Postcondition; the dot **.** of dot notation, which binds tighter than any other operator.

The operators listed include both standard operators and predefined operators (**=**, **/=**, **~**, **/~**). For a free operator, you cannot set the precedence: all free unaries appear at one level, and all free binaries at another level.

This precedence table is the basis for the rule removing potential syntactic ambiguities in operator expressions. We'll just work from a form that adds parentheses wherever needed:



Parenthesized Form of an expression

The **parenthesized form** of an expression is the result of rewriting every subexpression of one of the forms below, where \S and \ddagger are different binary operators, \diamond and \clubsuit different unary operators, and a, b, c arbitrary operands, as follows:

- 1 • For $a \S b \S c$ where \S is not the power operator \wedge : $(a \S b) \S c$ (left associativity).
- 2 • For $a \wedge b \wedge c$: $a \wedge (b \wedge c)$ (right associativity).
- 3 • For $a \S b \ddagger c$: $(a \S b) \ddagger c$ if the precedence of \ddagger is lower than the precedence of \S or the same, and $a \S (b \ddagger c)$ otherwise.
- 4 • For $\diamond \clubsuit a$: $\diamond (\clubsuit a)$
- 5 • For $\diamond a \S b$: $(\diamond a) \S b$
- 6 • For $a \S \diamond b$: $a \S (\diamond b)$
- 7 • For a subexpression e to which none of the previous patterns applies: e unchanged.

Since the notion of subexpression was defined recursively, the rewriting must be applied recursively too. Both notions are interesting for the case of an Operator_expression but are defined for general expressions, allowing the recursion to work properly.

The Parenthesized Form of

$$a + b * c \wedge \text{old } d$$

is

$$a + (b * (c \wedge (\text{old } d)))$$

The Parenthesized Form is not always *fully* parenthesized; it only adds the parentheses necessary to remove ambiguities. Here it doesn't put any around the full expression, or around entities a, b, c, d .

Operator \wedge gets a special treatment in clauses 1 and 2 of the definition because basic arithmetic types (*INTEGER*, *REAL* and their sized variants) use it as **power** operator: the mathematical notation a^{bc} is traditionally understood as meaning $a^{(bc)}$ — the only interesting interpretation since $(a^b)^c$ is just a^{b*c} .



Special cases in rules are unpleasant, but it is dangerous to go against long-standing mathematical conventions. Here a left-associative rule could cause errors for people trained in mathematics or physics. To avoid worrying about such issues, just use parentheses wherever there might be any doubt.

Clause 4 reflects that, in the above precedence table, all unary operators have the same precedence; and the last two clauses, that unary operators bind tighter than all binary operators.



\diamond and \ddagger can be the same operator, used as unary in one case and binary in the other. So clause 6 tells us that $a - - b$ — where the two signs must be separated by a break, lest we take them to start a comment — means $a - (-b)$.

→ “Syntax (non-production): Break rule”, . . . page 885.

To override the meaning implied by this rule, you may always use parentheses. For any Binary operator, the first operand of \S in

$(exp) \S other_exp$

is always exp , regardless of the precedence of \S and of the operators appearing in exp ; the last operand of \S in

$exp \S (other_exp)$

is always $other_exp$; and for any Unary operator \diamond , the expression

$\diamond (exp)$

always denotes the application of \diamond to the value of exp .



The precedence rules are easy to remember but competent Eiffel programmers mostly use them to understand the code of their macho colleagues. Don't hesitate to put parentheses around subexpressions to clarify intent and avoid errors. In particular, you should always use parentheses when a boolean expression uses different conjunctive and disjunctive operators in succession, as in $(a \text{ or } (b \text{ and } c))$.

We will build the Equivalent Dot Form of an expression, on which its validity and semantics are based, from its Parenthesized Form. In other words, thanks to this notion we can for all the rest of the discussion forget about any matters of ambiguity and operator precedence.

→ Clause “Equivalent Dot Form of an expression”, . . . page 780.

Accounting for target conversion

We need one more definition to handle all cases of operator expressions. It covers the mechanism that we studied in the chapter of conversions, allowing you to follow traditional mathematical practice by writing mixed-mode expressions such as $your_integer + your_real$ when you really mean to use the “+” operator from class *REAL*, converting the first operand to *REAL*. To make this possible, you must specify **convert** in the declaration of the operator, in class *INTEGER*:

← “MIXED-TYPE EXPRESSIONS: TARGET CONVERSION”, . . . 15.12, page 428.

plus alias "+" **convert** (other: *INTEGER*): *INTEGER*...

In this case the standard unfolding of *your_integer* + *your_real* into *your_integer*.*plus* (*your_real*) doesn't apply, since *REAL* neither conforms nor converts to *INTEGER*. We want to understand the expression as (*your_integer*.*converted_to_real*) + *your_real*. Because the first unfolding would be type-wise invalid, there is no danger of confusion.

A simple definition takes care of this case:



Target-converted form of a binary expression

The **target-converted form** of a *Binary_expression* $x \text{ § } y$, where the one-argument feature of alias § in the base class of x has the *Feature_name* f , is:

- 1 • If the declaration of f includes a **convert** mark and the type TY of y is not compatible with the type of the formal argument of f : ($\{TY\} [x]$) § y .
- 2 • Otherwise: the original expression, $x \text{ § } y$.

$\{TY\} [x]$ denotes x converted to type TY . This definition allows us, if the feature from x 's type TX cannot accept a TY argument but has explicitly been specified, through the **convert** mark, to allow for target conversion, and TY does include the appropriate feature accepting a TX argument, to use that feature instead.

The archetypal example is *your_integer* + *your_real* which, with the appropriate **convert** mark in the "+" feature in *INTEGER*, we can interpret as ($\{REAL\} [your_integer]$) + *your_real*, where "+" represents the *plus* feature from *REAL*.

(where $\{TY\} [x]$ denotes x converted to type TY). In fact that's all we need: ← [15.9, page 416](#).
 the validity and semantics, in this case, will simply rely — through the Equivalent Dot Form — not on the original expression but on its target-converted form. → [Clause 2, page 780](#).
 There is no need for any special rule, either for validity or for semantics.

Operator expression validity and semantics

Once no syntactical ambiguity remains, the validity and semantic properties of an *Operator_expression* are essentially those of a corresponding *Call*.

For every **Operator_expression** there will be an **Equivalent Dot Form**, syntactically a Call, illustrated above for a postcondition clause of class **LINKABLE**. As another example, here is the Equivalent Dot Form of our earlier expression $a + b + c * d$:

$$(a.\textit{plus}(b)).\textit{plus}(c.\textit{multiplied}(d))$$

This assumes that if x 's type has a base class C with operator features *plus* **alias** "+" and *multiplied* **alias** "*".

← "**OPERATOR FEATURES**", 5.15, page 154.

The next section gives a precise definition of the Equivalent Dot Form, although the above examples suffice to make the idea clear. Then the validity constraint on operator expressions is straightforward:



Operator Expression rule

VWOE

A **Unary_expression** § x or **Binary_expression** x § y , for some operator §, is valid if and only if it satisfies the following conditions:

- 1 • A feature of the base class of x is declared as **alias** "§".
- 2 • The expression's Equivalent Dot Form is a valid **Call**.

← The validity of calls was the subject of chapter 25.

The Feature Declaration rule tells us that a given operator may serve as alias for a unary feature (a feature without argument), or a binary feature (with one argument), or both, as in the case of + in **INTEGER** and **REAL**. In this last case, two features will match the requirement of clause 1; but that's OK because the form of the expression, unary or binary, will remove any ambiguity thanks to the definition of the Equivalent Dot Form.

← Page 162, relying on definition of "Alias Validity rule", page 163

→ Clause 1 and 2, page 780.

This rule ensures that every operator is used with the proper number of arguments. For example **INTEGER** and other basic arithmetic classes have a one-argument function *product* **alias** "*", but not zero-argument version, as would be required for a **Unary**. Then of the expressions

$$\begin{array}{l} 2 * 2 \\ * 2 \end{array}$$

WARNING: second expression not valid.

the first is valid but not the second.



The rule also explains why some binary operators can be used as "*multiary*" — meaning with three or more operands, of types all compatible with the type of the first — others are limited to two arguments. An example of multiary operator is + on integers; relational operators such as <, on the other hand, are binary but not multiary. This is clear from the Equivalent Dot Forms. With integer operands, the **Operator_expression**

$$1 + 2 + 3 + 4$$

has the Parenthesized Form

$$((1 + 2) + 3) + 4$$

← “*Precedence and Parenthesized Form*”, page 767.

yielding the valid Equivalent Dot Form

$$((1 \cdot \textit{plus} (2)) \cdot \textit{plus} (3)) \cdot \textit{plus} (4)$$

By the same rules, the Operator_expression

$$1 < 2 < 3$$

WARNING: this expression is not valid!

would yield the Equivalent Dot Form

$$(1 \cdot \textit{is_less} (2)) < 3$$

is not valid since the highlighted operand is of type **BOOLEAN**, but **BOOLEAN** does not have a function aliased to `<`, violating clause 1 of the Operator Expression rule.



If **BOOLEAN** had a function *is_less* alias “<”, perhaps with **false** considered less than **true**, this would still not make the expression valid: such a function would expect an argument of type **BOOLEAN**, not **INTEGER**. In this case it’s clause 2 that would fail. A true multiary operator, such as “+” on integers, must accept successive operands of the same or compatible type.

In summary, there is no need to define binary and multiary operators as separate syntactical categories. The grammar lists both kinds as **Binary**; whether a given operator may be used in multiary form depends on the signature of the corresponding function and on the precedence rules.



There remains to define the semantics of an Operator_expression. You are probably guessing from the preceding discussion that — as with validity — it is simply the semantics of its Equivalent Dot Form. You are guessing almost right; “almost” because (life not always being as simple as we would like) we must account for a special case, semistrict operators:

Expression Semantics (strict case)

The **value** of an **Expression**, other than a **Binary_expression** whose **Binary** is semistrict, is the value of its Equivalent Dot Form.

This semantic rule and the preceding validity constraint make it possible to forego any specific semantics for operator expressions (except in one special case) and define the value of any expression through other semantic rules of the language, in particular the rules for calls and entities.

This applies in particular to arithmetic and relational operators (for which the feature declarations are in basic classes such as *INTEGER* and *REAL*) and to boolean operators (class *BOOLEAN*): in principle, although not necessary as implemented by compilers, $a + b$ is just a feature call like any other.

The excluded case — covered by a separate rule — is that of a binary expression using one of the three **semistrict** operators: **and then**, **or else**, **implies**. This is because the value of an expression such as a **and then** b is not entirely defined by its Equivalent Dot Form $a \cdot \textit{conjoined_semistrict}(b)$, which needs to evaluate b , whereas the **and then** form explicitly ignores b when a has value *False*, as the value of the whole expression is *False* even if b does not have a defined value, a case which should not be treated as an error.

← “*PRECISE CALL SEMANTICS*”, 23.17, page 652; “*Entity Semantics rule*”, page 522.

→ “*Operator Expression Semantics (semistrict cases)*”, page 777.

28.6 SEMISTRICIT BOOLEAN OPERATORS

The semantic rule for operator expressions set out the special case of three boolean operators, known as “semistrict”. We’ll now take a look at these operators to understand why they are needed, and obtain the semantic rule for this case.

The ordinary (“strict”) boolean operators **not**, **and**, **or** and **xor**, defined in the Kernel Library class *BOOLEAN*, define operations on boolean values. The value of **not** a is true if and only if a has value false. The others are binary operators; the value they yield when applied to a first operand of value $v1$ and a second operand of value $v2$ is defined as follows:

- For **and**: true if and only if both $v1$ and $v2$ are false.
- For **or**: false if and only if either $v1$ or $v2$ is false.
- For **xor**: true if and only if $v1$ and $v2$ have different values. In other words, a **xor** b has the same value as $(a$ **or** $b)$ **and not** $(a$ **and** $b)$.

Three operators, also defined in *BOOLEAN*, complement **and** and **or**, from which they differ by a special semantic property known as semistrictness.

Semistrict operators

A **semistrict operator** is any one of the three operators **and then**, **or else** and **implies**, applied to operands of type *BOOLEAN*.

A general presentation of semistrictness appeared in 22.13.. You should not have any trouble understanding the present section even if you skipped the earlier, more theoretical discussion.

For operands of values $v1$ and $v2$ they yield the following results:

- **and then** (semistrict conjunction): false if $v1$ is false, otherwise the value of $v2$.

- **or else** (semistrict disjunction): true if $v1$ is true, otherwise the value of $v2$.
- **implies** (semistrict implication): true if $v1$ is false, otherwise the value of $v2$. (In other words, a **implies** b has the same value as **not a or else b** .)



At first sight, **and then** seems equivalent to **and**, **or else** to **or**, and **implies** to **or** with the first argument negated. The difference is that any one of these operators may in some cases yield a result on the sole basis of its first argument $v1$, if the value of $v1$ suffices to determine the outcome – even if the second argument does not have a value. They are “strict” (demand a value) for the first argument only, hence the term “semistrict”.

The difference arises for **and then** when $v1$ is false (result: false), for **or else** when $v1$ is true (result: true), and for **implies** when $v1$ is false (result: true). In these three cases the implementation must not evaluate the second argument $v2$. No such rule applies to **and** and **or**, which are not required to produce any result for an undefined second argument, and so may use a strict implementation as well as a semistrict one.

As a consequence, the semistrict operators, in contrast with their counterparts in standard mathematical logic, are not commutative: they do not treat their operands symmetrically. For example, a **and then** b does not necessarily have the same effect as b **and then** a . To be more accurate, any values these expressions yield will be the same, but it is possible for the second to yield a value when the first does not.

For a more complete discussion of strictness see the book "Introduction to the Theory of Programming Languages". For a study of various degrees of strictness in boolean operators see H. Barringer, J.H. Cheng and Cliff B. Jones, "A Logic Covering Undefinedness in Program Proofs", Acta Informatica, 21, 3, October 1984.

Because they enable you to write two-operand boolean expressions whose second operand need not have a value if the first operand's value leaves only one possible result, semistrict operators are particularly useful for a certain kind of loop used to traverse a data structure. Here is an example from a search routine in class *LINKED_LIST* in EiffelBase:



```

search_same (v: like first)
    -- Move cursor to first position (at or after current one)
    -- where v appears; move "off" if no such position.
do
    from
        ... (Initialization omitted)...
    variant
        count - position + 1
    until
        off or else (item = v)
    loop
        forth
    end
ensure
    (not off) implies (item = v)
end

```

The loop will terminate whenever the cursor moves after the last element (*off*), or hits an element whose value, as given by *item*, is equal to the argument *v*. The *Exit* expression tests for either of these conditions to occur. When the first condition (*off*) is true, however, we do not want to evaluate the second (*item = v*): not only would its contribution to the result be useless (since a disjunction with one true operand may have no value other than true); evaluating it would in fact be improper since function *item* is only defined when the cursor is on an actual element, which is not the case when it is *off*. (This is reflected in the precondition for *item*, which includes the condition *not off*.)

To guarantee the desired result, the *Exit* condition uses *or else* rather than *or*. In the same way, the postcondition only makes sense because of the semistrictness of *implies*. In other words, the semistrict semantics of *or else* and *implies* guarantees that *search_same* will work properly even if *v* does not appear in the list.

This common loop scheme is captured by *iterator* routines of EiffelBase, — *do_all*, *do_while*, *for_all* and others — declared in high-level classes such as *LINEAR* and hence available for most practical data structures. To use these routines, it suffices to pass them the appropriate agents as arguments, as in *your_list.for_all (agent your_condition)* which returns true if and only if every element of *your_list* satisfies *your_condition*.

This discussion leads us to the general semantic definition for nonstrict boolean operators:

Operator Expression Semantics (semistrict cases)

For a and b of type *BOOLEAN*:

- The value of a **and then** b is: if a has value false, then false; otherwise the value of b .
- The value of a **or else** b is: if a has value true, then true; otherwise the value of b .
- The value of a **implies** b is: if a has value false, then true; otherwise the value of b .

For each of the three forms, if the first condition listed holds, the computation of the expression's value must not cause evaluation of b .



The semantics of other kinds of expression, and Eiffel constructs in general, is **compositional**: the value of an expression with subexpressions a and b , for example $a + b$ (where a and b may themselves be complex expressions), is defined in terms of the values of a and b , obtained from the same set of semantic rules, and of the connecting operators, here $+$. Among expressions, those involving semistrict operators are the only exception to this general style. The above rule is not strictly compositional since it tells us that in certain cases of evaluating an expression involving b we should not consider the value of b . It's not just that we *may* ignore the value of b in some cases — which would also be true of a **and** b (strict) when a is false — but that we *must* ignore it lest it prevents us from evaluating the expression as a whole.

It's this lack of full compositionality that makes the above rule more **operational** than the semantic specification of other kinds of expression. Their usual form is “*the value of an expression of the form X is Y* ”, where Y only refers to values of subexpressions of X . Such rules normally don't mention order of execution. They respect compositionality and leave compilers free to choose any operand evaluation order, in particular for performance. Here, however, order matters: the final requirement of the rule *requires* that the computation first evaluate a . We need this operational style to reflect the special nature of nonstrict operators, letting us sometimes get a value for an expression whose second operand does not have any.

28.7 BRACKET EXPRESSIONS

What makes a bracket expression possible is a feature declared with a `bracket alias` clause, as in

← “*BRACKET FEATURE*”, 5.17, page 158.



```
item alias "["(key: H): G ...
```

which — if this declaration appears in `HASH_TABLE`, and `your_table` is of type `HASH_TABLE [T, U]` — allows writing `your_table [your_key]` as an abbreviation for `your_table.item (your_key)`.

The Kernel Library class `ARRAY [G]` relies on this technique to allow accessing array elements through the notation `your_array [n]` as a synonym for `your_array.item (n)` for an integer `n`. You are not limited to one argument: a class `MATRIX3 [G]` describing three-dimensional matrices may have

```
item alias "[" (i, j, k): G ...
```

allowing element access under the form `your_matrix [n1, n2, n3]`.

This mechanism is also useful in connection with assigner procedures: adding `assign put` (after `G`) to any of these examples, with a procedure `put` having the appropriate signature, allows you to use assignment syntax, as

```
your_matrix [n1, n2, n3] := v
```

in the last example, an abbreviation for `your_matrix.put (v, n1, n2, n3)`. The left side is, again, a `Bracket_expression`.

The syntax is simple:



Bracket expressions

```
Bracket_expression  $\triangleq$  Bracket_target "[" Actuals "]"
```

```
Bracket_target  $\triangleq$  Target | Once_string |  
Manifest_constant | Manifest_tuple
```

`Target` covers every kind of expression that can be used as target of a call, including simple variants like `Local` variables and formal arguments, as well as `Call`, representing the application of a query to a target that may itself be the result of applying calls.

Examples of `Bracket_expression` are



```
your_table [your_key]  
your_matrix [n1, n2, n3]  
table_list.i_th (i) [your_key]
```

In the first two cases, the `Call_chain` is just a single query, `your_table` or `your_matrix`; such a `Bracket_expression` could appear respectively in class `HASH_TABLE` or `MATRIX3`. The last example, with a longer `Call_chain`, assumes that in the base class for `table_list` there's a function `i_th` returning a table.

The `Bracket_target` used to the left of the bracket part allows a number of expression variants; `Call_chain` is the most common, permitting bracket expressions such as `f[x]` but also `a.b.f[x]` (to be understood again as an abbreviation: for `a.b.f.item(x)` for the appropriate `item` function). One of the other possibilities is `Manifest_tuple`, as in `[a, b, c][i]`, taking advantage of a bracket alias for `item` in `TUPLE`. If you want a more complex expression as target, use a `Parenthesized_target`, as in

```
(a + b) [i]
```

which will be valid if the type of `a + b` has a bracket feature.

The reason for the restriction of `Bracket_target` to specific kinds of expressions is — as you might not have guessed! — the need to make the semicolon optional in all cases without causing any syntactical ambiguity. If you are interested in understanding this fully, you'll find the details in the final [section](#) of this chapter.

→ [“EXPRESSIONS AND THE SEMICOLON”, 28.12, page 784.](#)

The Equivalent Dot Form of a `Bracket_expression` simply involves replacing the expression by a call in dot notation, using the associated feature. For the above three examples it is:

```
your_table.item (your_key)  
your_matrix.item (n1, n2, n3)  
table_list.i_th (i).item (your_key)
```

These examples all assume `item` as the `Feature_name` for the bracket feature; this is indeed the most common choice, but of course you may choose any name you like.

Here is the validity rule:



Bracket Expression rule

VWBR

A **Bracket_expression** $x [i]$ is valid if and only if it satisfies the following conditions:

- 1 • A feature of the base class of x is declared as **alias** "[]".
- 2 • The expression's Equivalent Dot Form is a valid **Call**.

The Feature Declaration rule ensures that at most one feature satisfies clause 1. The Equivalent Dot Form, as defined below, relies on that feature.

← Page [162](#), clause [7](#):
see clause [2](#) of "[Alias Validity rule](#)", page [163](#)

28.8 THE EQUIVALENT DOT FORM



This section defines precisely the notion of Equivalent Dot Form, already introduced informally through examples, and used extensively in the previous sections. It may be skipped on first reading.

For a full specification of the validity and semantics of an **Operator_expression** or **Bracket_expression**, we need a precise description of how to obtain its Equivalent Dot Form. Because such expressions may involve components which are expressions of other kinds (such as calls or constants), the definition must in fact be applicable to any kind of expression. In the following definition the most important cases are the first three, giving dot equivalents for the non-dot forms (operators, bracket):

Equivalent Dot Form of an expression

Any **Expression** e has an **Equivalent Dot Form**, not involving (in any of its subexpressions) any **Bracket_expression** or **Operator_expression**, and defined as follows, where C denotes the base class of x , pe denotes the Parenthesized Form of e , and x', y', c' denote the Equivalent Dot Forms (obtained recursively) of x, y, c :

- 1 • If pe is a **Unary_expression** § $x: x'.f$, where f is the **Feature_name** of the no-argument feature of alias § in C .
- 2 • If pe is a **Binary_expression** of target-converted form $x \& y: x'.f(y')$ where f is the **Feature_name** of the one-argument feature of alias § in C .
- 3 • If pe is a **Bracket_expression** $x [y]: x'.f(y')$ where f is the **Feature_name** of the feature declared as **alias** "[]" in C .
- 4 • If pe has no subexpression other than itself: pe .
- 5 • In all other cases: (recursively) the result of replacing every subexpression of e by its Equivalent Dot Form.

In the first three cases, the Operator Expression and Bracket Expression rules seen earlier in this chapter guarantee that there is a feature f of the given alias. The Feature Declaration rule then ensures that in all of the first three cases exactly one feature f satisfies the requirements.

← Pages [772](#) and [780](#).

← Page [162](#), relying on definition of “[Alias Validity rule](#)”, page [163](#).



The Operator Expression and Bracket Expression rules both rely on the definition of Equivalent Dot Form, raising the appearance of circular reasoning. But we are only interested in Equivalent Dot Forms of expressions that satisfy clause 1 of their respective rules; this is enough to make the definition of Equivalent Dot Form applicable, and then to use it in the rule’s second clause. So this mutual dependency does not cause circularity.

In case [2](#) we draw the feature f not from the original expression but from its target-converted form as presented in the preceding section. It will usually be identical, but allows us for example to accept `your_integer + your_real`, treating it as `(your_integer.converted_to_real) + your_real`.

Case [4](#) is the terminal case of the recursion, covering `Formal`, `Local`, `Manifest_constant`, and any `Call` consisting of a single query with no arguments. Case [5](#) makes sure that we apply the rule recursively to all constituents of a complex expression.

Case applies, among others, to a `parenthesized` expression (`f`), for which it gives us (`f'`) where `f'` is, recursively, the Equivalent Dot Form of `f`.

28.9 BOOLEAN EXPRESSIONS

For `Boolean_expression`, the grammar at the beginning of this chapter gave three kinds: `Boolean_constant`, `Object_test` and `Basic_expression`. The two boolean constants are `True` and `False`. `Object_test` has its own validity rule. → Page [788](#). The third case must satisfy an obvious constraint:

Boolean Expression rule VWBE

A `Basic_expression` is valid as a `Boolean_expression` if and only if it is of type `BOOLEAN`.

Here the “type” of a `Basic_expression` is the result of applying the Expression Type definition appearing [below](#).

→ Page “[Type of an expression](#)”, page [783](#).

28.10 ENTITIES

Entities do not appear as a separate case in the syntax for Expression because they form a special case of Call (more precisely `Unqualified_call`). But their role as expressions or components of expressions deserves a few comments.

First, as a reminder, the syntactic definition:



Entities and variables

Entity \triangleq Variable | Read_only
 Variable \triangleq Attribute | Local
 Attribute \triangleq Identifier
 Local \triangleq Identifier | Result
 Read_only \triangleq Formal | *Current*
 Formal \triangleq Identifier

This syntax appeared originally on page 512.

The associated constraint, called the Entity rule, required any entity to be one of: attribute; local variable of the enclosing routine if any (including *Result* if it is a function); formal argument of the enclosing routine or inline agent; feature of a call; *Current*.

← “Entity rule”, page 513.

Together with the Call rule, the Entity rule governs the use of identifiers in expressions. A simple consequence of these two constraints is:



Identifier rule

VWID

An **Identifier** appearing in an expression in a class *C*, other than as the feature of a qualified Call, must be the name of a feature of *C*, or a local variable of the enclosing feature or inline agent if any, or a formal argument of the enclosing feature or inline agent if any, or the Object-Test Local of an Object_test.

The restriction “other than as the feature of a qualified **Call**” excludes an identifier appearing immediately after a dot to denote a feature being called on a target object: in $a + b.c(d)$, the rule applies to *a*, *b* (target of a **Call**) and *d* (actual argument), but not to *c* (feature of a qualified **Call**). For *c* the relevant constraint is the Call rule, which among other conditions requires *c* to be a feature of the base class of *b*'s type.

In the Equivalent Dot Form, a actually appears as target of a call, and b both as argument of a call and target of another.

The Identifier rule is not a full “if and only if” rule; in fact it is conceptually superfluous since it follows from earlier, more complete constraints. Language processing tools may find it convenient as a simple criterion for detecting the most common case of invalid **Identifier** in expression.

--- REFERENCE TO ENTITY EVALUATION SEMANTICS

28.11 THE TYPE OF AN EXPRESSION

Every expression has a type; this notion is central to the validity rules governing (among others) assignment, argument passing and the construction of larger expressions from smaller ones.

This **static type** of the expression, entirely deduced from declarations in the software text, shouldn't be confused with the *dynamic type* of its value at some instant of execution.

← “*Dynamic type*”, page 606.

We are now in a position to define precisely the notion of static type for each kind of expression.

A full definition must remove the effect of genericity: if a is of type $D [U]$ and x is an attribute or function declared of type G in class $D [G]$, the type we want for $a.x$ is not G — meaningless outside of class C — but U . This has been taken care of by the Generic Type Adaptation rule, which tells us to apply the actual-to-formal parameter substitutions whenever our types involve generic derivations. By referring to this rule, the following Expression Type definition can ignore genericity for its own specific cases:

Type of an expression

The type of an **Expression** e is:

- 1 • For the predefined **Read_only Current**: the current type.
- 2 • For a routine's **Formal** argument : the type declared for e .
- 3 • For an **Object-Test** local: its declared type.
- 4 • For **Result**, appearing in the text of a query f : the result type of f .
- 5 • For a local variable other than **Result**: the type declared for e .
- 6 • For a **Call**: the type of e as determined by the Expression Call Type definition with respect to the current type.
- 7 • For a **Precursor**: (recursively) the type of its unfolded form.
- 8 • For an **Equality**: **BOOLEAN**.
- 9 • For a **Parenthesized** (f): (recursively) the type of f .
- 10 • For **old** f : (recursively) the type of f .
- 11 • For an **Operator_expression** or **Bracket_expression**: (recursively) the type of the Equivalent Dot Form of e .
- 12 • For a **Manifest_constant**: as given by the definition of the type of a manifest constant.
- 13 • For a **Manifest_tuple** $[a_1, \dots, a_n]$ ($n \geq 0$): **TUPLE** $[T_1, \dots, T_n]$ where each T_i is (recursively) the type of a_i .
- 14 • For an **Agent**: as given by the definition of the type of an agent expression.
- 15 • For an **Object_test**: **BOOLEAN**.
- 16 • For a **Once_string**: **STRING**.
- 17 • For an **Address** $\$v$: **TYPED_POINTER** $[T]$ where T is (recursively) the type of v .
- 18 • For a **Creation_expression**: the Explicit_creation_type.

→ The current type is obtained from the current class by adding the formal generic parameters, if any. See 12.11, page 365.

← “*Type of a Call used as expression*”, page 655.

← “*Type of an agent expression*”, page 759.

Case 6, which refers to a definition given in the discussion of calls, also determines case 11, operator and bracket expressions.

28.12 EXPRESSIONS AND THE SEMICOLON



We end this review of expressions with a syntactical note (which you may skip on first reading). The distinction between `Basic_expression` and `Special_expression` has, among others, a syntactic purpose. Eiffel's Semicolon rule specifies that the semicolon as separator is always optional. It must be applicable to any `Assertion_clause`, which can be an `Unlabeled_assertion_clause` and hence directly follow another clause, which could end with

```
... f[x]
```

using a `Bracket_expression`, the application of f to x . To a naive parser, however, this could look like two successive clauses:

```
... f;  
[x]
```

WARNING: not valid.

without the semicolon. The second line, a `Manifest_tuple`, is also an expression, and hence a possible assertion clause if it were valid. It is *not* valid, since a tuple cannot be boolean as required for an assertion clause; but that's validity information, whereas it should be possible to parse software texts on the basis of syntactical information only.

Fortunately, the syntax avoids any such problem thanks to the division between `Basic_expression` and `Special_expression`. `Unlabeled_assertion_clause`, and every context where similar ambiguities could arise, only accept a `Basic_expression`; all the constructs such as `Manifest_tuple` that could cause such ambiguities are part of `Special_expression`.

This technique no loss of generality because if you do want to start a component (for example an `Unlabeled_assertion_clause`) with a legitimate expression that, syntactically, is a `Special_expression`, you can just put it in parentheses: as `Parenthesized` is part of `Basic_expression` this does the trick.

In some cases, you may also use a `Parenthesized_target`. Note for example the following assertion, valid if f is of type `BOOLEAN`:

```
require
  f                               -- No semicolon necessary
  ([x, y, z]).count > 0
```

Valid, assuming the proper declarations (but not the recommended style).

This assertion includes two clauses; the first is true if and only if f is true, and the second is trivially true since it states that a 3-item tuple has a positive number of items.

Such cases are extreme, and in fact the conscientious programmer always labels assertion clauses:

```
require
  property_1: f
  property_2: ([x, y, z]).count > 0
```

The recommend style.

But this is only a recommendation. The syntax rule guarantee the basic Eiffel right of omitting semicolons between elements on different lines — greatly enjoyed by all users of the language.

Constants

29.1 OVERVIEW

Expressions, just studied, include the special case of constants, whose values cannot directly be changed by execution-time actions. This discussion goes through the various kinds. Particular attention will be devoted to the various forms, single- and multi-line, of *string* constant.

Along with constants proper, we will study two notations for “manifest” objects given by the list of their items: manifest tuples and manifest arrays, both using the syntax [*item*₁, ... *item*_{*n*}].

29.2 GENERAL FORM OF CONSTANTS

A **Constant** expression has a value that does not change at run time, and is the same for all instances of a class.

A **Constant** is required as “inspect constant” in **Multi_branch** instructions (chapter 17).

The form is:



Constants	
Constant	≜ Manifest_constant Constant_attribute
Constant_attribute	≜ Feature_name

A **Constant_attribute** denotes a constant value, specified in the attribute’s declaration as a **Manifest_constant**. The use of an identifier as **Constant_attribute** is subject to an obvious constraint:

Constant attributes were discussed in chapter 18



Constant Attribute rule	VWCA
A Constant_attribute appearing in a class C is valid if and only if its Feature_name is the <u>final name</u> of a <u>constant attribute</u> of C .	

← The rules for recognizing constant attributes and other feature categories were given in 5.12, page 145.

To apply this rule, you must look at the declaration of the attribute and check that, according to the rules for distinguishing between various kinds of feature, it indeed defines a constant attribute.

If not a `Constant_attribute`, a `Constant` will be a `Manifest_constant`, whose form directly determine both a type and a value:



Manifest constants

```

Manifest_constant  $\triangleq$  [Manifest_type] Manifest_value
Manifest_type  $\triangleq$  "{" Type "}"
Manifest_value  $\triangleq$  Boolean_constant |
                Character_constant |
                Integer_constant |
                Real_constant |
                Manifest_string |
                Manifest_type

Sign  $\triangleq$  "+" | "-"

Integer_constant  $\triangleq$  [Sign] Integer
Character_constant  $\triangleq$  "'" Character "'"
Boolean_constant  $\triangleq$  True | False
Real_constant  $\triangleq$  [Sign] Real

```

→ The syntax for `Manifest_string` appears later in this chapter, page 795.

For clarity and consistency, and to avoid mistakes, we put a restriction — not expressible in BNF-E — on the use of signs:

← A similar rule applied to operators: "Syntax (non-production): Alias Syntax rule", page 151. See also rules on characters

Syntax (non-production): Sign Syntax rule

If present, the `Sign` of an `Integer_constant` or `Real_constant` must immediately precede the associated `Integer` or `Real`, with no intervening tokens or components (such as breaks or comments).

Similarly, for characters:

Syntax (non-production): Character Syntax rule

The quotes of a `Character_constant` must immediately precede and follow the `Character`, with no intervening tokens or components (such as breaks or comments).

In general, breaks or comment lines may appear between components prescribed by a BNF-E production, making the last two rules necessary to complement the grammar: for signed constants, you must write `-5`, not `- 5` etc. This helps avoid confusion with operators in arithmetic expressions, which may of course be followed by spaces, as in `a - b`. Similarly, you must write a character constant as `'A'`, not `' A '`.



To avoid any confusion about the syntax of `Character_constant`, it is important to note that a character code such as `%N` (New Line) constitutes a single `Character` token.

→ “*Syntax (non-production): Manifest character*”, page 895.

The following sections study the role of the optional `Type` in braces, then the various cases, except for `Boolean_constant`, about which it suffices to note that this construct only has two specimens, `True` and `False`, whose values are different (when compared for equality).



Beyond basic types, there is also a need for specifying constant values of complex types. This is addressed through **once functions**. The body of a once function is executed at most once, to compute the result of the first call (if any). All subsequent calls return the same result as the first, without further computation.

On once functions (and once routines in general) see 8.5, page 222, and the semantics of calls in chapter 23.

For functions of reference types, this yields constant *references*. The scheme is particularly useful for objects containing shared information and may be illustrated as follows:

```
shared: SOME_REFERENCE_TYPE
    -- A reference to an object shared by
    -- all instances of the enclosing class
once
    create Result ... (...);
    ... Further operations on Result, if needed, to update
        the attached object ...
end
```

Calls to `shared` always return a reference to the object created and initialized by the first call. Only the reference is constant here, not the object itself since clients can change its fields through procedure calls

```
shared.some_procedure (...)
```

29.3 FORCING A TYPE ON A CONSTANT

The syntax for `Manifest_constant` on the preceding page specifies



Manifest types

`Manifest_constant` \triangleq [`Manifest_type`] `Manifest_value`

A **Manifest_value** directly specifies a value, for example *3.141592* or *"ABC"*. That value always determines a type: *REAL* and *STRING* in these two examples. Usually this is also the type that you want for the **Manifest_constant** as a whole and, if so, you don't need to qualify the **Manifest_value** further. For example if you use *1*, a specimen of the lexical construct **Integer**, as a constant, the type rules imply that it will be understood as a **Manifest_constant** of type *INTEGER*. This is usually the desired result; but if you want the expression to be of a different type — a sized integer variant — you may specify a **Manifest_type**, as in



```
{INTEGER_8} 1
```

This syntax does not imply a conversion, but simply forces an explicit type. You will need it in some cases, for example, to specify large integer values. Depending on a global setting that you can override through a compilation option, *INTEGER* is a synonym for either *INTEGER_32* or *INTEGER_64*. Assume it means *INTEGER_32*. The maximum value of that type is approximately 2^{31} . If you want to express the value 2^{32} , representable as an *INTEGER_64* but not as an *INTEGER_32*, you may not write *4_294_967_296* (even though that's the correct mathematical value) since it's invalid as an *INTEGER_32*, being beyond the bounds. You may, however, use

← See also [“*EXPRESSION CONVERTIBILITY: THE ROLE OF PRECONDITIONS*”](#), 15.10, page 420.

→ The optional underscores let you group digits for readability. See [“*INTEGERS*”](#), 32.16, page 899.



```
{INTEGER_64} 4_294_967_296
```

29.4 THE TYPE OF A CONSTANT

---- EXPLAIN



Type of a manifest constant

The type of a **Manifest_constant** of **Manifest_value** *mv* is:

- 1 • For $\{T\}$ *mv*, with the optional **Manifest_type** present: *T*. The remaining cases assume this optional component is absent, and only involve *mv*.
- 2 • If *mv* is a **Boolean_constant**: *BOOLEAN*.
- 3 • If *mv* is a **Character_constant**: *CHARACTER*.
- 4 • If *mv* is an **Integer_constant**: *INTEGER*.
- 5 • If *mv* is a **Real_constant**: *REAL*.
- 6 • If *mv* is a **Manifest_string**: *STRING*.
- 7 • If *mv* is a **Manifest_type** $\{T\}$: *TYPE [T]*.

As a consequence of cases [3](#) to [6](#), the type of a character, string or numeric constant is never one of the sized variants but always the fundamental underlying type (*CHARACTER*, *INTEGER*, *REAL*, *STRING*). Language mechanisms are designed so that you can use such constants without hassle — for example, without explicit conversions — even in connection with specific variants. For example:

- You can assign an integer constant such as 10 to a target of a type such as *INTEGER_8* as long as it fits (as enforced by validity rules).
- You can use such a constant for discrimination in a Multi_branch even if the expression being discriminated is of a specific sized variant; here too the compatibility is enforced statically by the validity rules.

Case [7](#) involves the Kernel Library class *TYPE*.

The *Manifest_type* notation is only applicable to manifest constants of types with sized variants:



<h3>Manifest-Type Qualifier rule</h3> <p><i>VWMQ</i></p> <p>It is valid for a <i>Manifest_constant</i> to be of the form $\{T\} v$ (with the optional <i>Manifest_type</i> qualifier present) if and only if the type <i>U</i> of <i>v</i> (as determined by cases 2 to 7 of the definition of the <u>type of a manifest constant</u>) is one of <i>CHARACTER</i>, <i>STRING</i>, <i>INTEGER</i> and <i>REAL</i>, and <i>T</i> is one of the <u>sized variants</u> of <i>U</i>.</p>
--

→ “*Basic types and their sized variants*”, page 817.



The rule states no restriction on the value, even though an example such as $\{INTEGER_8\} 256$ is clearly invalid, since 256 is not representable as an *INTEGER_8*. The Manifest Constant rule addresses this.

Do not confuse this *Manifest_type* notation $\{T\} const$, for a constant *const*, with the mechanism for expressing conversions explicitly: ----- REWRITE THIS FOR NEW NOTATION ----- $\{T\} [exp]$ which simply applies a function *adapted* from class *TYPE* to the target *exp*, an arbitrary expression, triggering any necessary conversion in the process. The $\{T\} const$ notation only applies to constants and **will not cause a conversion**, as noted above in the example of $\{INTEGER_8\} 1$.

← “*CONVERTING AN EXPRESSION EXPLICITLY*”, 15.9, page 416.

The effect of adding a *Manifest_type* follows from the informal description:

<h3>Manifest Constant Semantics</h3> <p>The value of a <i>Manifest_constant</i> <i>c</i> listing a <i>Manifest_value</i> <i>v</i> is:</p> <ol style="list-style-type: none"> 1 • If <i>c</i> is of the form $\{T\} v$ (with the optional <i>Manifest_type</i> qualifier present): the value of type <i>T</i> denoted by <i>v</i>. 2 • Otherwise (<i>c</i> is just <i>v</i>): the value denoted by <i>v</i>.

--- In case [2](#), the lexical rules of the language ensure that the form of the constants uniquely determines one of the types listed. For example, [123](#) is an integer, but [12.3](#) is a real; [{T}](#) is a *TYPE*; and so on.

Sometimes we just need to refer to the value explicitly listed for a constant, ignoring any *Manifest_type*. The following definition captures this notion:

Manifest value of a constant

The **manifest value** of a constant is:

- 1 • If it is a *Manifest_constant*: its value.
- 2 • If it is a constant attribute: (recursively) the manifest value of the *Manifest_constant* listed in its declaration.

29.5 INTEGER CONSTANTS

An integer constant – a specimen of *Integer_constant* – consists of an Integer, possibly preceded by a sign. (Integer, a lexical construct, describes unsigned integers.) Example specimens of *Integer_constant* are:

Construct Integer is described as part of the lexical specification in [32.15, page 898](#).

```
0
253
-57
+253
```

29.6 REAL CONSTANTS

A real constant – a specimen of *Real_constant* – consists of a Real, possibly preceded by a sign.

Real, a lexical construct, describes floating-point numbers. Without a scaling factor, the possible forms of Real are

Real is described in [32.16, page 899](#).



```
a.
.b
a.b
```

where *a* and *b* are specimens of Integer. Any of these may be followed by the letter *E*, an optional sign and an Integer to indicate scaling by a power of ten.

Here are some example specimens of `Real_constant`:



```
46.
54.
24.36
-34.65
-34.65E-12
45.21E2
+45.21E2
```

29.7 CHARACTER CONSTANTS

A `Character_constant` is a character enclosed in single quotes, as in



```
'c'
```

The following constant attribute declarations define symbolic names for some specimens of `Character_constant`:

```
Upper_z: CHARACTER is 'Z';
Dollar_sign: CHARACTER is '$';
blank: CHARACTER is ' ';
```

A `Character_constant` consists of exactly three characters: the first and the third are single quotes `'`; the middle one is a `Character` other than a single quote.

Allowing a quote would not cause any ambiguity (since there are always exactly three characters altogether), but the rule is consistent with the convention for double quotes in a `Manifest_string`, as studied in the next section.



The value of a `Character_constant` is its middle `Character`.

To understand the above syntactic definition, you must realize that a `Character` is either a key corresponding directly to a printable character (such as `A` or `$`) or one of a set of multiple-key special character codes beginning with the percent sign *percent*. Examples of such codes are: → *“Special characters and their codes”, page 897.*

- `%N` for a new-line.
- `%'` for a single quote.
- `%B` for a backspace.
- `%/"91"/%` for the character of ASCII code 91.

91 is the (decimal) code of the opening bracket[, which you may also write as %(.

For example, a class text may include constant attribute declarations such as



```
New_line: CHARACTER is '%N';
Single_quote: CHARACTER is '%'
```

Because a new-line is not a **Character**, the three characters of a **Character_constant** must appear on the same line. Of course, you may define a constant whose value is a new-line character by using `%N` as middle **Character**.



In spite of appearances, the presence of `'` in a **Character_constant**, as in the declaration of *Single_quote* above, does not violate the prohibition of the quote character as a constant's **Character**: `%'`, as all the special character codes, is considered to be a single character, although it consists of two signs (percent and single quote). This is explained in detail in the specification of characters.

29.8 MANIFEST STRINGS

A **Manifest_string** denotes an instance of the Kernel Library class *STRING*, studied in a later chapter. → See [36.7, page 937](#), about class *STRING*.

As the following syntax indicates, there are two ways to write a manifest string:

- A **Basic_manifest_string**, the most common case, is a sequence of characters in double quotes, as in *"This text"*. Some of the characters may be special character codes, such as `%N` representing a new line. This variant is useful for such frequent applications as object names, texts of simple messages to be displayed, labels of buttons and other user interface elements, generally using fairly short and simple sequences of characters. You may write the string over several lines by ending an interrupted line with a percent character `%` and starting the next one, after possible blanks and tabs, by the same character.
- A **Verbatim_string** is a sequence of lines to be taken exactly as they are (hence the name), bracketed by `"{` at the end of the line that precedes the sequence and `}"` at the beginning of the line that follows the sequence (or `"[` and `]"` to left-align the lines). No special character codes apply. This is useful for embedding multi-line texts; applications include *description* entries of **Notes** clauses, inline C code, SQL or XML queries to be passed to some external program.

An example of the first of these uses is:



note

```
description: "[
    Constants covering kinds of user interface events.
    This class is meant to be used as ancestor
    by classes needing its facilities.
]"
```

Here are the syntax rules for these two variants:



Manifest strings

```
Manifest_string ≜ Basic_manifest_string |
    Verbatim_string

Basic_manifest_string ≜ "' ' String_content "'
String_content ≜ {Simple_string Line_wrapping_part ...}+
Verbatim_string ≜ Verbatim_string_opener
    Line_sequence
    Verbatim_string_closer

Verbatim_string_opener ≜ "' '[Simple_string] Open_bracket
Verbatim_string_closer ≜ Close_bracket [Simple_string] "'

Open_bracket ≜ "[" | "{"
Close_bracket ≜ "]" | "}"
```

In the “basic” case, most examples of `String_content` involve just one `Simple_string` (a sequence of printable characters, with no new lines, as defined in the description of lexical components). For generality, however, `String_content` is defined as a repetition, with successive `Simple_string` components separated by `Line_wrapping_part` to allow writing a string on several lines. Details below.

→ The “extreme case” arises when one of the lines in the `Line_sequence` begins with `]`. See “[Verbatim strings](#)”, page 798.

In the “verbatim” case, `Line_sequence` is a lexical construct denoting a sequence of lines with arbitrary text. The reason for the `Verbatim_string_opener` and the `Verbatim_string_closer` is to provide an escape sequence for an extreme case (a `Line_sequence` that begins with `]`), but most of the time the opener is just `"["` or `"{"` and the closer `"]"` or `"}"`. The difference between brackets and braces is that with `"{ ... }"` the `Line_sequence` is kept exactly as is, whereas with `"[...]"` the lines are left-aligned (stripped of any common initial blanks and tabs). Details below.

As with some other constructs, we need to clarify the use of breaks through the definition of `Line_sequence`:

← Similar rules applied to operators, in “[Syntax \(non-production\): Alias Syntax rule](#)”, page 151, and to signed constants, in “[Syntax \(non-production\): Sign Syntax rule](#)”, page 788.

Syntax (non-production): Line sequence

A specimen of `Line_sequence` is a sequence of one or more `Simple_string` components, each separated from the next by a single `New_line`.

and a consistency constraint on manifest strings:

Syntax (non-production): Manifest String rule

In addition to the properties specified by the grammar, every `Manifest_string` must satisfy the following properties:

- 1 • The `Simple_string` components of its `String_content` or `Line_sequence` may not include a double quote character except as part of the character code `%"` (denoting a double quote).
- 2 • A `Verbatim_string_opener` or `Verbatim_string_closer` may not contain any break character.

Like other “non-production” syntax rules, the last two rules capture simple syntax requirements not expressible through BNF-E productions.

Because a `Line_sequence` is made of simple strings separated by a single `New_line` in each case, a line in a `Verbatim_string` that looks like a comment is not a comment but a substring of the `Verbatim_string`.

The details of both variants now follow, after an explanation of the **once** keyword.

Basic manifest strings

To denote the actual string content, you have the choice between `Basic_manifest_string` and `Verbatim_string`.

An example `Basic_manifest_string` is:

```
"This Manifest_string contains 43 characters"
```



The value is a `STRING` object, which represents a sequence of characters. In this example the sequence contains all the characters given except for the two enclosing double quotes, which play a purely syntactical role.

Any of the characters may be a character code as discussed [earlier](#) for [Character_constant](#). This enables you to include a double quote character into the string: use its code, `%"`. Similarly

← “[CHARACTER CONSTANTS](#)”, 29.7, page 793.



```
"First line%NNew line"
```

describes a string with two lines, separated by a newline character represented as `%N`.

Whether the **value** of a [Basic_manifest_string](#) is text extending over just one line or several, you have the freedom to write the **specification** of the string in the Eiffel text over one or more lines. In particular you may, for readability, write a long [Basic_manifest_string](#) over several lines in the source text. For this you will use one or more **line wrapping parts**, interrupting the string with a percent sign `%` at the end of a line, and resuming it with a percent sign on the next line.

An example using two line wrapping parts (shown by the shaded zones) is the [Manifest_string](#)

First wrapping part: shaded on first two lines. Second part: shaded on last two lines.



```
"This Manifest_string con%
%ains 43 %
%characters"
```

There is a space at the end of the second line, after '43'.

The sole purpose of the line wrapping form is to enable you to write a [Manifest_string](#) on several lines, but without retaining the line separations in the string that it denotes. So the contents of the last example do not tell a lie: the resulting string is indeed exactly the same as in with the first example of this section; it is simply written on three lines rather than one. Note that the initial blanks or tabs on the second and third lines, before the percent sign, are there only for layout and do not contribute to the string.

More generally, the division of the string into two or more lines in the source text has no effect on the value of the string. In the above example, the value is a one-line string (which we may also write as just `"This Manifest_string contains 43 characters"`). By adding `%N` characters we would turn this into a multi-line string values. In contrast, the [Verbatim_strings](#) to be seen next will retain the line formatting of the string as it appears in the Eiffel text.

Here are the definition and rules formalizing the preceding properties. The syntax of [Basic_Manifest_string](#) involves one or more [Line_wrapping_part](#), with the following definition:



Line_wrapping_part

A [Line_wrapping_part](#) is a sequence of characters consisting of the following, in order: `%` (percent character); zero or more blanks or tabs; [New_line](#); zero or more blanks or tabs; `%` again.

This construct requires such a definition since it can't be specified through a context-free syntax formalism such as BNF-E.



The use of `Line_wrapping_part` as separator between a `Simple_string` and the next in a `Basic_manifest_string` allows you to split a string across lines, with a `%` at the end of an interrupting line and another one at the beginning of the resuming line. The definition allows blanks and tabs before the final `%` of a `Line_wrapping_part` although they will not contribute to the contents of the string. This makes it possible to apply to the `Basic_manifest_string` the same indentation as to the neighboring elements. The definition also permits blanks and tabs after the initial `%` of a `Line_wrapping_part`, partly for symmetry and partly because it doesn't seem justified to raise an error just because the compiler has detected such invisible but probably harmless characters.

As to the semantics:



Manifest string semantics

The value of a `Basic_manifest_string` is the sequence of characters that it includes, in the order given, excluding any line wrapping parts, and with any character code replaced by the corresponding character.

→ "[Special characters and their codes](#)", page 897.

Verbatim strings

A `Verbatim_string` is a sequence of lines meant to be retained exactly as they are (except for possible left-alignment).

→ A break character is a space or a tab. See "[BREAKS](#)", 32.5, page 881.



You may bracket a `Verbatim_string` between a line ending with `"[`, possibly followed by break characters, and a line beginning with `]"`, possibly preceded by break characters.

The earlier example showed a *description* entry in a `Notes` clause:



```
note
  description: "[
                Constants covering kinds of user interface events.
                This class is meant to be used as ancestor
                by classes needing its facilities.
                ]"
```

The reason for allowing break characters after the initial `"[` and before the final `]"` is, as above, to avoid raising an error in harmless cases.

As the examples below illustrate, there may be other Eiffel elements preceding the initial `"[` on the same line, and following the final `]"` on the same line.

The beginning and ending delimiters of a `Verbatim_string` will usually be `"[` and `]"` as above, but the syntax gave a more general convention:

← This syntax first appeared on page 795.



Verbatim strings

```
Verbatim_string  $\triangleq$  Verbatim_string_opener
                    Line_sequence
                    Verbatim_string_closer
```

```
Verbatim_string_opener  $\triangleq$  ''' [Simple_string] Open_bracket
```

```
Verbatim_string_closer  $\triangleq$  Close_bracket [Simple_string] ' '''
```

```
Open_bracket  $\triangleq$  "[" | "{"
```

```
Close_bracket  $\triangleq$  "]" | "}"
```

In this general form the string is bracketed by " $\alpha\{$ and $\}\alpha$ ", or " $\alpha[$ and $] \alpha$ ", where α — the `Simple_string` of the `Verbatim_string_opener` and `Verbatim_string_closer` — is any sequence of characters not including a double quote ". In most cases, including all the examples in this section, α is an empty string, but a non-empty α is useful in the case — unavoidable if we want to have a completely general mechanism — in which one of the lines of the string begins with the closing delimiter `] "` (or `}"` if the opening delimiter was `"{`). To handle such a string as a `Verbatim_string`, choose a string α such that no line of the text begins with $\alpha]$ " (or $\alpha}"$), possibly preceded by breaks.

This convention is an application of the general language design rule that any convention for quoting text, using specific characters or character sequences as delimiters, must have an **escape** convention making it possible to quote a text that includes the delimiters themselves.

Here is an example of this convention:



note

description: "++[

This class, from a hypothetical Eiffel parser, is in charge of parsing Verbatim strings that end with]" or some variant thereof.

]++"

Because of the highlighted line beginning, we can't use "[and]" as delimiters; instead we choose "++[and ++]", making sure that no line in the string begins with]++". This is of course an extreme case, and most uses of `Verbatim_string` will rely on the default delimiters.

This observation leads to the constraint on verbatim strings::



Verbatim String rule

VWVS

A `Verbatim_string` is valid if and only if it satisfies the following conditions, where α is the (possibly empty) `Simple_string` appearing in its `Verbatim_string_opener`:

- 1 • The `Close_bracket` is] if the `Open_bracket` is [, and } if the `Open_bracket` is {.
- 2 • Every character in α is printable, and not a double quote ".
- 3 • If α is not empty, the string's `Verbatim_string_closer` includes a `Simple_string` identical to α .

Truly identical: in strings, letter case is significant.

Next, the semantics. Whatever the delimiters, the value of a `Verbatim_string` is given by its `Line_sequence` (the sequence of lines that make it up, delimiters excluded), taken exactly as it is, except for left-alignment if the delimiters used brackets [] rather than braces. The characters in the `Line_sequence` are retained exactly as they are. No special character codes apply: if %N (for example) appears in the `Line_sequence`, it will be understood as two characters, a percent and an *N*.

This rule assumes a view of strings as plain sequences of characters, line separations being marked by a special "new line" character.

Here is the semantic rule that states these properties:



Verbatim string semantics

The value of a `Line_sequence` is the string obtained by concatenating the characters of its successive lines, with a "new line" character inserted between any adjacent ones.

The value of a `Verbatim_string` using braces { } as `Open_bracket` and `Close_bracket` is the value of its `Line_sequence`.

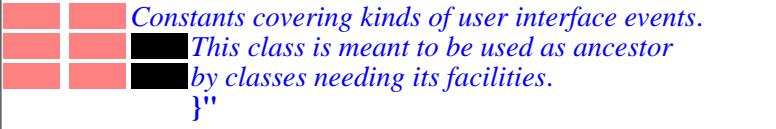
The value of a `Verbatim_string` using brackets [] as `Open_bracket` and `Close_bracket` is the value of the left-aligned form of its `Line_sequence`.



This semantic definition is **platform-independent**: even if an environment has its own way of separating lines (such as two characters, carriage return %R and new line %N, on Windows) or represents each line as a separate element in a sequence (as in older operating systems still used on mainframes), the semantics yields a single string — a single character sequence — where each successive group of characters, each representing a line of the original, is separated from the next one by a single %N.

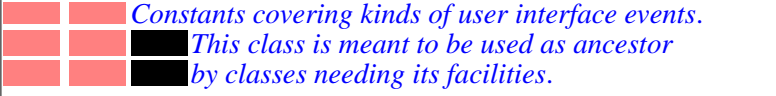
(“**Left-aligned form**” should be intuitively clear, but is defined rigorously [below](#).) The difference between the two kinds of brackets follows from whether you want to left-align the string. If you don’t, use braces, as in [“Prefix, longest break prefix, left-aligned form”, page 804](#).



note
description: "{

Constants covering kinds of user interface events.
This class is meant to be used as ancestor
by classes needing its facilities.
}"

The shaded rectangles show the positions of tab characters.

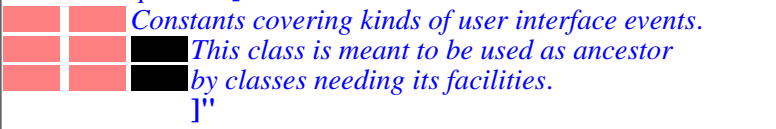
where the various tab positions have been highlighted; the value of the string retains all these tabs:


Constants covering kinds of user interface events.
This class is meant to be used as ancestor
by classes needing its facilities.


This is the value of the `Verbatim_string` of the above example.

Originally, however, we had written this example using square brackets “[...]” to request left alignment:



note
description: "[

Constants covering kinds of user interface events.
This class is meant to be used as ancestor
by classes needing its facilities.
]"

This means that the string’s value doesn’t include the two initial tabs (highlighted above as lightly shaded rectangles) common to all three lines. It does, however, include the extra tab (dark rectangle) that appears only on the last two lines. So that value is

Constants covering kinds of user interface events.

This class is meant to be used as ancestor
by classes needing its facilities.



Why go into all this trouble by having two kinds of delimiters, one implying left alignment and the other not? The reason is cosmetic. In many cases, you want the left-aligned version of a verbatim string. This example, involving *description* entries of **Notes** clauses, is typical. Language processing tools may store the values into a reuse repository to facilitate content-based retrieval of classes; in ISE Eiffel, for example, all such entries are turned into **META** tags of the generated HTML documentation, so that you can retrieve them through Web search engines. But without automatic left-alignment you would have to start your class text as

```

note
  description: "{
Constants covering kinds of user interface events.
This class is meant to be used as ancestor
by classes needing its facilities.
}"
  author: "Jane Programmer"
class YOUR_CLASS ...

```

WARNING: ugly layout, not recommended. See text.

which messes up the indentation and layout. Using the brackets "[...]" instead of the braces "{ ... }" solves the problem: you indent the text in a way that looks nice in your software text

```

note
  description: "[
Constants covering kinds of user interface events.
This class is meant to be used as ancestor
by classes needing its facilities.
]"
  author: "Jane Programmer"
class YOUR_CLASS ...

```

WARNING: ugly layout, not recommended. See text.

The same observation applies if you use a **Verbatim_string** for inline C code in an external function, as in this extract from an example in the [discussion of interfacing Eiffel with C](#), involving two verbatim strings: → ["Specifying C code inline", page 844.](#)



```

an_inline_function (x,y: INTEGER): INTEGER
  external "[
    C
    inline
    use <stdio.h>, /path/user/her_include.h
  ]"
  alias "[
    if ($x > cvar) {
      some_c_function ($y, cvar++);
    }
  ]"
end

```

*Warning: the content of the **alias** clause represents C, not Eiffel. The indentation is shown as it will appear in the class text (where a feature declaration is already indented one step, as part of a **Feature_clause**).*

or in a hypothetical example using an embedded SQL string:



```

example
do
  sql_statement := database.prepare ("[
    SELECT emp
    FROM EMPLOYEE
    WHERE salary >= 50000
  ]")
end

```

The indentation in these last two examples should not be retained in the strings actually passed to the appropriate tools — a C compiler, an SQL query processor for a Database Management System. (The C compiler probably don't care, but other tools might!), The indentation is only beneficial to the reader Eiffel text, who might cringe when seeing a version left-aligned for the sake of the external tool, such as

```

example
do
  sql_statement := database.prepare ("{
SELECT emp
FROM EMPLOYEE
WHERE salary >= 50000
  }")
end

```

WARNING: ugly layout, not recommended. See text.

which breaks the layout of Eiffel texts. Using the bracket form "[...]" solves this problem by allowing you to keep a pleasant layout without retaining the extra initial tabs or spaces in the string's semantics.

→ "[LAYOUT](#)", 34.9, page 915.

About "tabs or spaces": the recommended practice is always to use tabs for indentation. Some older text editors, however, prefer spaces.

The following auxiliary definitions give a precise meaning to the notion of “left-aligned form” used above to specify the semantics of a verbatim strings using brackets as delimiters: ← “*Verbatim String rule*”, page 800.



Prefix, longest break prefix, left-aligned form

A **prefix** of a string s is a string p of some length n ($n \geq 0$) such that the first n characters of s are the corresponding characters of p .

The **longest break prefix** of a sequence of strings ls is the longest string bp containing no characters other than spaces and tabs, such that bp is a prefix of every string in ls . (The longest break prefix is always defined, although it may be an empty string.)

The **left-aligned form** of a sequence of strings ls is the sequence of strings obtained from the corresponding strings in ls by removing the first n characters, where n is the length of the longest break prefix of ls ($n \geq 0$).

Obviously, if you are passing a verbatim string to an external tool that attaches a semantic value to initial tabs and spaces on a line, and want to control the string character-by-character without any intervention from the left-aligning process implied by these definitions, you should use the brace form "{ ... }" even if this means uglifying the Eiffel text layout. (Yes, semantic correctness should in the end win over esthetic appeal.) But in that case you are probably better off anyway using the non-verbatim form of manifest strings, control characters and all.

Choosing between basic and verbatim manifest strings

You can indeed write any `Verbatim_string` as a `Basic_manifest_string`; for example to get the same result as our earlier Verbatim_string, you may use: ← This refers to the non-left-aligned version, using braces, on page 801.



note

```
description: "%TConstants covering kinds of user %  
%interface events.%N%  
%%T%TThis class is meant to be used%  
%as ancestor%N%  
%%T%Tby classes needing its facilities"
```

Note the need to include explicit codes `%N` (new line) and `%T` (tab) and, if the text is too long for pleasant formatting, break it into several lines with “line wrapping parts”. The two divisions do not necessarily coincide: in this example the string has been broken after *of user* and *used*, but there is no new line at those positions in the string’s value.



Since the words being sent to separate lines (*user* and *interface*, *used* and *as*) must be separated by a space, you mustn't forget to include that space — highlighted above — at the end of the interrupted line or the beginning of the continuation line. It's this need to code everything explicitly that makes `Basic_manifest_string` tedious to use for describing multi-line strings; `Verbatim_string` avoids the problem.

“Once” string expressions



The syntax for expressions in the preceding chapter included the `Once_string` expression, of the form `once some_manifest_string`, for example `once "This text"`. This is actually an expression, not a constant; you can use it in contexts that expect an expression, for example an assignment or argument passing

```
your_string_entity := once "This text"
your_procedure (once "This text")
```

but not where only a `Manifest_string` will do, for example a manifest declaration `your_constant is "This text"`, a Debug key, an Obsolete message. But since this notion is closely connected to the semantics of strings it is appropriate to study it here.

The possibility of qualifying a manifest string constant by `once` when you use in an expressions corresponds to different semantics for the manifest string:

- Without `once`, every evaluation of the string creates a new object.
- With `once`, only the first evaluation creates an object; every subsequent one yields a reference to that initial object.

This is of course the same difference as between a function declared with `do` and one declared with `once`. In fact you may understand a manifest string `"This text"` as a call to a function of the form\

← See [“ROUTINE BODY”, 8.5, page 222](#), about once routines.



```
new_string: STRING
do create Result.make_from_string ("This text") end
```

whereas the manifest string `once "This text"` is equivalent to a call to

```
once_string: STRING
once create Result.make_from_string ("This text") end
```

Which one of the two forms should you use? Each has advantages and drawbacks depending on the circumstances. If you find yourself using a text message in a loop, as in



```
from... loop ... until
...
  print (once "Message text")
end
```

you should probably use the **once** form, to avoid creating lots of identical objects.



It is almost always better in this case to declare a constant attribute:

```
Your_message: STRING is "Message text"
  -- Note that once "Message text" is valid here,
  -- but would make no difference.
...
from... loop ... until
...
  print (Your_message)
end
```

which avoids the problem altogether and is in line with general methodology principles (don't use literal constants in algorithms!). The **once** form is available, however, for designers who feel they truly need manifest strings with no symbolic names.



Because they are shared, however, once manifest strings can cause some surprises. Consider the following extract, with *text* of type *STRING*:

```
text := ""
text.append (i.out)
  -- i.out is the string representation of the integer i
```

Assume that this appears in the body of a loop, and that each iteration of the loop increments *i* by one. You probably expect that each loop iteration will reinitialize *text* to an empty string, then extend it with the string value of *i*, setting it to "1" the first time, "2" the second time and so on. This is indeed what happens with the extract as given since the empty string is recreated each time. But if for the first instruction you use

```
text := once ""
```

then only one string is ever created, so that in loop iterations the string will take successive values "1", "12", "123" and so on. This is seldom the desired result and explains why the **once** behavior is not the default. (If novices are going to have bad surprises, better be it because of bad initial performance — which can be fixed through a **once** declaration — than because of wild and seemingly buggy behavior.)



The general rule, then, is:

- Use **once** for manifest strings that will not change at all — although it is even better in most cases to give them symbolic names such as *Your_message* above, and stop worrying.
- Use non-**once** strings as soon as anything can change, keeping in mind then that each use of the constant will create a new object.

Run-time model for manifest strings

As explained above, a **Manifest_string** *m* defines an associated **Simple_string** *s*. For example, in the **Manifest_string** appearing in the declaration

Message: **STRING** is "Example 1"

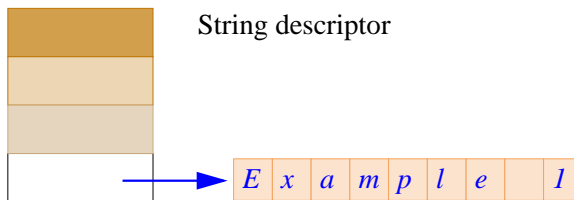
m is "Example 1" and *s* is the **String** made of the nine characters *Example 1*.

For most practical purposes you may view *s* as the value of *m*. In a more precise specification of the semantics, however, this is not quite correct. Although the nuance is somewhat fine, you should understand it even if this is your first reading, because of a potential confusion that has been known to surprise newcomers.

The problem is that the value of *m*, like any other value, must be an object or a reference to an object, and that a **String** (a sequence of characters) is not appropriate for this purpose. The desired object should be an instance of some class, so that you can apply features to it: for example a routine to which you pass *Message* as actual argument may need to access properties of *Message* such as its length, through the features of some appropriate class. But there is no class whose instances are just arbitrary sequences of characters.

There does exist a class meant for representing character strings: the Kernel Library class *STRING*, which indeed served as type for *Message* in the above example declaration. But an instance of *STRING* is not a sequence of characters: it is a string descriptor, which must of course provide access to the characters but may also include other information such as the string length. Most importantly, because a string descriptor is an instance of a class, all the features of that class are applicable to it. Class *STRING* offers many routines for operations on strings such as accessing the character at a given position, extracting a substring and appending another string.

*The details of class *STRING*, its representation and its features are given in [36.7, page 937](#).*



The details of string representation do not matter for this discussion, although the above figure shows a possible implementation, where the string descriptor includes, among various possible fields, a reference to the actual character sequence.

The character sequence is a "special object", as introduced in [19.2, page 506](#). See the discussion of strings in [36.7](#).

What does matter is that the value of an entity declared of *STRING* type, such as *Message* above, is a string descriptor, not a character sequence.

What difference does it make? Only one consequence is of practical concern: a *Manifest_string* is not a "constant" in the sense that most people would expect. It will always refer to the same string **descriptor**, but not necessarily to the same *character sequence*; this is because the contents of the descriptor may be changed through procedure calls.

Here is an example showing how this can happen:



```
Message: STRING is "Example 1"
...
Message.put ('2', 9)
```

where the call to *put*, a procedure of class *STRING*, replaces the character at position 9 (originally *1*) by the character *2*. As a result, if a later instruction prints *Message*, the output will be

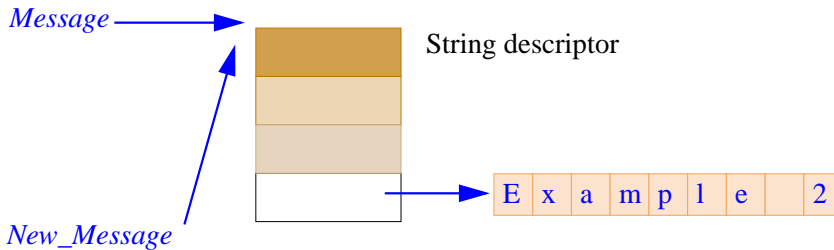
```
Example 2
```

The same potential problem may arise through a feature call applying to another entity than the original "constant" if there has been an assignment; for example, the assignment



```
New_message := Message;
```

produces a situation where both entities refer to the same character sequence:



The characters are assumed to be Example 2 as a result of the call to put in the previous assignment..

Then any operation on *Message* will have the same effect as the corresponding operation on *New_message*. For example, after

```
Message.put ('3', 9)
```

an instruction to print *New_message* will produce *Example 3*.

29.9 MANIFEST TUPLES



The next form of **Expression** is the **Manifest_tuple**, defining a tuple through a list of expressions representing the tuple's successive items.

An example of **Manifest_tuple**, of type *TUPLE [INTEGER, REAL, INTEGER]*, is



```
[27, 3.5, m + n]
```

for *m* and *n* of type *INTEGER*. The value of this expression is a tuple of three elements, having the values given.

If the list of expressions in a **Manifest_tuple** is empty, it describes an array with no elements:

```
[]
```



Among other applications, you may use a **Manifest_tuple** to obtain the effect of a variable number of arguments for a routine: if one of the formal arguments of a routine is declared with the type **ARRAY [T]** for some *T*, then an actual argument may be an expression list

See [8.4, page 221](#).

```
[e1, e2, ... en]
```

such that every *ei* is of a type conforming to *T*. The number *n* of elements in the list is arbitrary, so that you indeed obtain the same effect as if routines were permitted to have a variable number of arguments. Examples of this technique appeared in the discussion of routines.

The *ei* do not need, of course, to be all of the same type, as long as their types all conform to *T*. By choosing a more specific or more general *T* (based on a class lower or higher in the inheritance graph), you restrict or extend the set of acceptable types for the *ei*.

Here again is the syntax of manifest tuples:

← From the tuple chapter, page [373](#).



Manifest tuples

Manifest_tuple \triangleq "[" Expression_list "]"

Expression_list \triangleq {Expression ", " ... }

There is no constraint on manifest tuples.[TO BE COMPLETED -- FOLLOWING RULE REMOVED]: This is the rule that may be used in practice to ascertain whether a **Manifest_tuple** is appropriate as actual argument to a routine, or whether an assignment of the form

```
a := [e1, e2, .. en]
```

is valid. For example, with the routine specification



```
average_age (group: ARRAY [PERSON]): INTEGER
```

then the call in

```
group_average := average_age
  ([Fiordiligi, Dorabella, Guglielmo, Ferrando, Alfonso])
```

is valid if *Fiordiligi* and *Dorabella* are of type **LADY**, *Guglielmo* and *Ferrando* of type **GENTLEMAN**, Alfonso of type **PHILOSOPHER**, and all these classes are descendants of **PERSON**. If you add to the **Manifest_tuple** an expression whose type does not conform to **PERSON**, the **Manifest_tuple** ceases to be a valid expression of type **ARRAY [PERSON]**.

As another example of applying the Manifest Array rule, this time recursively, here is a valid expression of type `ARRAY [ARRAY [INTEGER]]`:

```
<<<<-3, 41>>, <<0>>, <<45, 31, -27>>, <<>>>>
```

describing an array whose elements are integer arrays of two, one, three and zero elements successively.

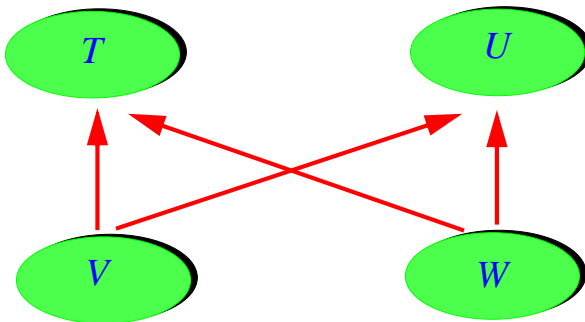


As you may have noted, the Manifest Array rule departs slightly from the style using elsewhere in this book to talk about types, since it does not define "the type of" a `Manifest_tuple`, but instead tells us how to ascertain whether a `Manifest_tuple` is "a valid expression of type" `ARRAY [T]` for given `T`. This is because, in contrast with the other expressions studied in this chapter (and any other Eiffel component that has a value), a `Manifest_tuple` does not have a single type of the form `ARRAY [T]` for a single `T`. You may see this by considering the `Manifest_tuple` in

```
v1: V; w1: W
...
some_routine (<<v1, w1>>)
```

where `V` and `W` are non-generic classes with the inheritance structure illustrated on the adjacent page.

Here the given `Manifest_tuple` is a valid argument for `some_routine` if the formal argument is declared either as `ARRAY [T]` or as `ARRAY [U]` (as well as `ARRAY [X]` for any `X` to which `T`, `U` or both conform). If we tried to define "the type of the `Manifest_tuple`, however, `ARRAY [T]` and `ARRAY [U]` would be equally good candidates.



Fortunately, this inability to settle for a single type will not cause any difficulty in the two situations which require obtaining type information about a `Manifest_tuple` *ma*:

The Manifest Array rule enables us to ascertain conformance of *ma* to a certain type (the type of the target in an assignment, or of a routine's formal argument) without any ambiguity, as illustrated by the above examples.

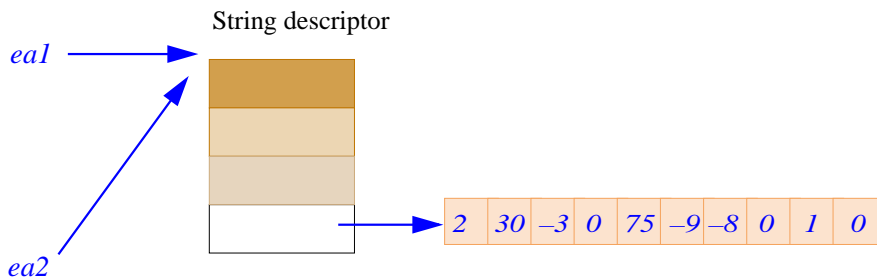
The dynamic type set of *ma*, needed to ascertain system-level call validity, is the set of all types of the form `ARRAY .[T]` for every type *T* in the dynamic type set of any of the elements of *ma*. This is in line with the handling of genericity in the definition of the dynamic type set.



The value of a `Manifest_tuple` made of *N* expressions is an array of bounds 1 and *N*, whose elements are the values of the successive expressions in the `Manifest_tuple`. In this definition, an “array” is an instance of the Kernel Library class `ARRAY`.

Arrays and class ARRAY are discussed in [36.4, page 934](#).

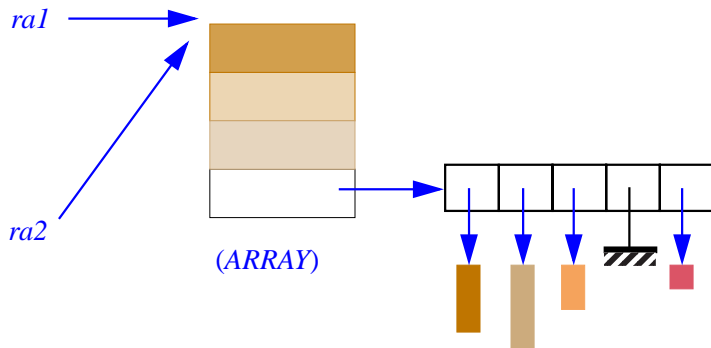
The situation here is similar to what we encountered above for strings: an instance of `ARRAY` is in fact an array descriptor, which must of course provide access to the actual elements, but may also include other information such as the number of elements and the bounds. This also means that an array descriptor is a normal object – an instance of class `ARRAY`, with all the features of this class applicable to it.



The last figure illustrates the notion of array descriptor; it is a conceptual representation of the situation resulting from

```
ea1, ea2: ARRAY {INTEGER};
...
ea1 := <<27, 54, -3, 7, 1, 0, 10, 546, -40>>
ea2 := ea1
```

In this case the array's elements are objects (instances of the expanded type *INTEGER*). Below is an illustration of the effect of similar operations on arrays *ra1* and *ra2* of type *ARRAY* $.[T]$ for some reference type *T* =



This could be the result of:

```

ra1, ra2: ARRAY [T];
a, b, c, d, T;
...
create {T} a ...
create {U} b ...
create {V} c ...
create {W} d ...
ra1 := <<a, b, c, d>>;
ra2 := ra1

```

U, V, W are types conforming to *T*.

29.10 SEMANTICS OF CONSTANT ATTRIBUTES

--- To be completed ---

Basic types

30.1 OVERVIEW

The term “basic type” covers a number of expanded class types describing elementary values: booleans, characters, integers, reals, machine-level addresses. The corresponding classes — *BOOLEAN*; *CHARACTER*, *INTEGER*, *REAL* and variants specifying explicit sizes; *POINTER* — are part of ELKS, the Eiffel Library Kernel Standard.

The following presentation explains the general concepts behind the design and use of these classes.

The specification (flatshort form) of the corresponding classes appears in the ELKS appendix. Many of the features and invariants as they appear there are self-explanatory, reflecting the standard operations and properties of arithmetic. → “*BOOLEAN*”, [A.6.8 CLASS, page 982](#) to “*POINTER*”, [A.6.18 CLASS, page 995](#).

ARRAY, *STRING* and sized variants, also described by ELKS classes, are not basic types even though they share some of their properties; for one thing, they are reference rather than expanded. We’ll study them separately.

→ Chapter [36](#).

30.2 EXPANSION STATUS

The classes describing the basic types are expanded; this is the desired semantics for the vast majority of cases. With reference types, an entity declared as

n: *INTEGER*

would denote a reference to an object containing an integer. This would waste space and time; besides, we would still need a way to set the integer to a specific value, such as the integer [3](#).

Instead, the semantics is what a casual reader would expect: *n* represents an integer.

The denoted values — integers, booleans, characters, ... — are objects. These objects are both normal and special:

- They are normal because you can treat them, for all practical purposes, as you would any other Eiffel object (instance of some class). They are instances of classes (*INTEGER* and such) and have the applicable features.
- They are special, however, to the compiler: even though $a + b$ is, conceptually, a function call, you most likely do not want such an operation, for integers a and b , to trigger a routine call at run time. Any performance-focused Eiffel compiler will know about *INTEGER* and other basic types, and will handle the addition as efficiently as a compiler for languages such as C where it is a frozen, built-in operation.

The idea is to get the best of both worlds: the generality, power and universality of the object-oriented mechanisms; the efficiency of predefined operations.

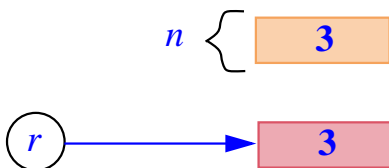
If you do want to manipulate references to basic values you may simply use entities of *ANY*, since it is a reference class:



```
n: INTEGER
r: ANY
...
n := 3
...
r := n
```

With a target r of reference type and a source n of expanded type, the semantics of reattachment implies that r will be assigned a reference to a new copy of n , :

← “*SEMANTICS OF REATTACHMENT*”,
22.7, page 593.



Assigning a basic value to a reference target

The other way around you may use an *Object_test* to get the integer back:

```
attached r local m: INTEGER then
    n := m
else
    ... Handle case in which r is not attached to an integer ...
end
```

30.3 BASIC CLASSES AND THEIR INHERITANCE STRUCTURE

The following definition provides the exact list of types defined as basic:



Basic types and their sized variants

A **basic type** is any of the types defined by the following ELKS classes:

- *BOOLEAN*.
- *CHARACTER*, *CHARACTER_8*, *CHARACTER_32*, together called the “**sized variants** of *CHARACTER*”.
- *INTEGER*, *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *NATURAL*, *NATURAL_8*, *NATURAL_16*, *NATURAL_32*, *NATURAL_64*, together called the “**sized variants** of *INTEGER*”.
- *REAL*, *REAL_32*, *REAL_64*, together called the “**sized variants** of *REAL*”.
- *POINTER*.

Like basic types, *STRING* has “sized variants”:



Sized variants of *STRING*

The sized variants of *STRING* are *STRING*, *STRING_8* and *STRING_32*.

CHARACTER and its sized variants cover the notion of character as used in strings; *CHARACTER_8* corresponds to extended (8-bit) ASCII-like codes, *CHARACTER_32* to Unicode. *INTEGER* describes signed integers; the sized variants correspond to representations using a specified number of bits and, in the case of *NATURAL* variants, unsigned integers. *REAL* and its sized variants correspond to floating point numbers.

By convention the definitions include each of the elementary notions, such as *INTEGER*, among its own “sized variants”.



Calling *NATURAL* a “sized” variants of *INTEGER*, a small abuse of language, simplifies the description of the numerous properties that apply consistently to all integer-related types, signed or unsigned.

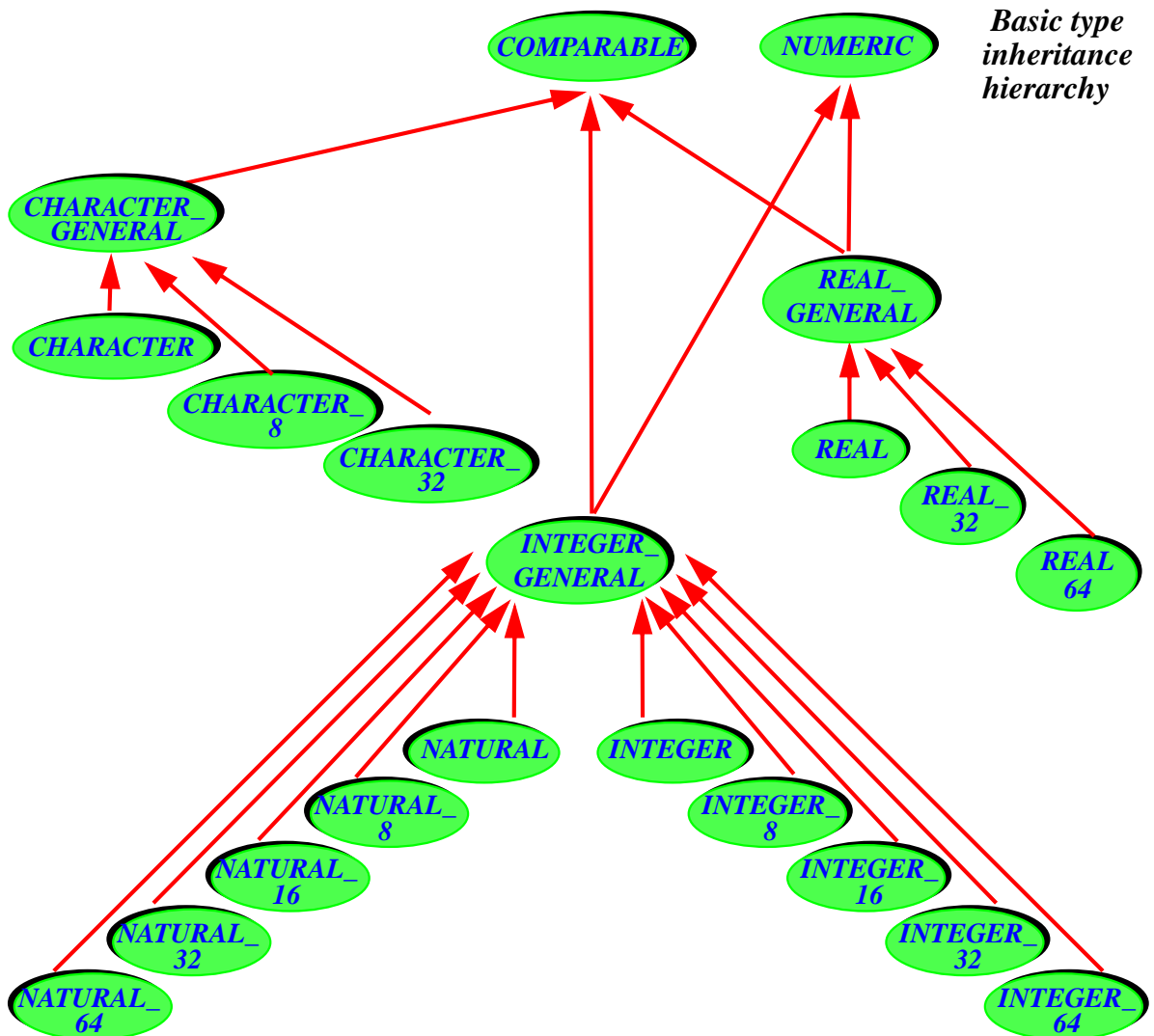
The figure on the next page illustrates the part of the ELKS inheritance hierarchy relevant to the basic types. All the basic types are descendants of either of both of the deferred ELKS classes

- *COMPARABLE*, based on the mathematical the notion of **total order**, introduces comparison operations such as “less than” and “greater than or equal”. → Full specification in: [“PART COMPARABL E”, A.6.3, page 977.](#)

- ***NUMERIC***, based on the mathematical the notion of **ring**, introduces arithmetic operations such as addition and multiplication. → Full specification in: [“*NUMERIC*”, A.6.6, page 980.](#)

Character, integer and real classes do not directly inherit from these but from an intermediate class describing common features of each category:

- ***CHARACTER_GENERAL*** (an heir of ***COMPARABLE*** but not ***NUMERIC***) introduces features common to all ***CHARACTER*** variants.
- ***INTEGER_GENERAL***, describing integers of arbitrary size, introduces features common to all ***INTEGER*** and ***NATURAL*** variants.
- Similarly, ***REAL*** and its sized variants inherit from ***REAL_GENERAL***, describing floating point numbers of arbitrary precision.





The formal arguments and results of the routines of class *NUMERIC* are all of type *NUMERIC*. To ensure the validity of arithmetic operations, the class texts for *INTEGER*, *REAL* and sized variants redeclare the arguments and results to be of types *INTEGER*, *REAL* and so on.

In spite of these redeclarations, you may use the traditional forms of mixed-type arithmetic; for example you may add an integer to a real number. Such combinations derive from the conversion mechanisms provided by the language, which explicitly allow an expression such as $3 + 4.5$, and prescribe interpreting it as $3. + 4.5$, all operands being converted to the “heavier” type *REAL*.

← Chapter 15 and “Accounting for target conversion”, page 770.

We now review the basic semantic properties of the types involved.

30.4 BOOLEANS

→ Full specification in: “*BOOLEAN*”, A.6.8 CLASS, page 982.



Boolean value semantics

Class *BOOLEAN* covers the two truth values.

The reserved words *True* and *False* denote the corresponding constants.

Class *BOOLEAN* provides the boolean operators, discussed in the chapter on expressions, which also explained the difference between strict operators (**and**, **or**) and their semistrict variants (**and then**, **or else**)

← “*SEMISTRIC BOOLEAN OPERATORS*”, 28.6, page 774.

30.5 CHARACTERS

→ Full specification in: “*CHARACTER*”, A.6.9 CLASS, page 983.



Character types

The reference class *CHARACTER_GENERAL* describes properties of characters independently of the character code.

The expanded class *CHARACTER_32* describes Unicode characters; the expanded class *CHARACTER_8* describes 8-bit (ASCII-like) characters.

The expanded class *CHARACTER* describes characters with a length and encoding settable through a compilation option. The recommended default is Unicode.

Every character has an associated positive integer code, given by the query *code*, with a value no greater than *Maximum_character_code*.

30.6 INTEGERS



Integer types

The reference class *INTEGER_GENERAL* describes integers, signed or not, of arbitrary length. The expanded classes *INTEGER_xx*, for *xx* = 8, 16, 32 or 64, describe signed integers stored on *xx* bits. The expanded classes *NATURAL_xx*, for *xx* = 8, 16, 32 or 64, describe unsigned integers stored on *xx* bits.

The expanded classes *INTEGER* and *NATURAL* describe integers, respectively signed and unsigned, with a length settable through a compilation option. The recommended default is 64 bits in both cases.

→ Full specification in: [“INTEGER_GENERAL”, A.6.10, page 984.](#)

INTEGER corresponds to the default precision provided by each Eiffel implementation, normally 64, but settable through a compilation option. This convention ensures optimal performance since on many platforms a specific precision (for example 32) will yield the most efficient floating-point operations. The practical advice is:



- In most ordinary computation, you may rely on *INTEGER*, after checking the default semantics for your compiler, 64 on most platforms.
- If the correctness of your algorithm requires a specific length, you may use *INTEGER_xx* for the appropriate *xx*.
- If you are reusing software with routine arguments or results of type *INTEGER*, use the compilation option if necessary to ensure compatibility with the length used by the rest of your algorithms.

At the time of writing, 32 is more common than 64 on most platforms. The choice of 64 as the recommended default is based on the assumption that 64-bit integers will become dominant. In the meantime implementations may use 32 as the default if this still provides the best performance.

In addition to the ordinary arithmetic and relational operators, boolean operators are available on integers, treated as bit sequences. Found in class *INTEGER*, they include:

- *conjoined alias* "&"
- *conjoined alias* "&"
- *left_shifted alias* "|<<"
- *right_shifted alias* "|>>"
- *negated*

One — *negated* — is unary; the others are binary.

30.7 REALS



Floating-point types

The reference class [*REAL GENERAL*](#) describes floating-point numbers with arbitrary precision. The expanded classes [*REAL_{xx}*](#), for *xx* = 32 or 64, describe IEEE floating-point numbers with *xx* bits of precision.

The expanded class [*REAL*](#) describes floating-point numbers with a precision settable through a compilation option. The recommended default is 64 bits.

→ Full specification in: [*“REAL GENERAL”, A.6.15, page 991.*](#)

[*REAL*](#) corresponds to the default precision provided by each Eiffel implementation, normally 64, but settable through a compilation option. This convention ensures optimal performance since on many platforms a specific precision (for example 64) will yield the most efficient floating-point operations. The practical rule for numerical computation is:

- In most ordinary computation, you may rely on [*REAL*](#), after checking the default semantics for your compiler, 64 on most platforms.
- If the correctness of your algorithm requires a fixed precision, you may use [*REAL₃₂*](#) or [*REAL₆₄*](#). Alternatively, you may use [*REAL*](#) and a compilation option to set the corresponding precision (but make sure to document this requirement).
- If you are reusing software with routine arguments or results of type [*REAL*](#), use the compilation option if necessary to ensure compatibility with the precision used by the rest of your algorithms.



30.8 ADDRESSES

Address semantics

The expanded class [*POINTER*](#) describes addresses of data beyond the control of Eiffel systems.

→ Full specification in: [*“POINTER”, A.6.18, page 995.*](#)

This is in particular the type for [*expressions of the Address form*](#), [*\\$ expr.*](#)

Since [*POINTER*](#) objects have no components accessible to the Eiffel side, the class has no exported features of its own.

← [*“PASSING THE ADDRESS OF AN EIFFEL FEATURE”, 31.8, page 833.*](#)

Interfacing with C, C++ and other environments

31.1 OVERVIEW: THE COMPONENT COMBINATOR

Object technology as realized in Eiffel is about **combining components**. Not all of these components are necessarily written in the same language; in particular, as organizations move to Eiffel, they will want to reuse their existing investment in components from other languages, and make their Eiffel systems interoperate with non-Eiffel software.

Eiffel is a “pure” O-O language, not a hybrid between object principles and earlier approaches such as C, and at the same time an **open** framework for combining software written in various languages. These two properties might appear contradictory, as if consistent use of object technology meant closing oneself off from the rest of the programming world. But it’s exactly the reverse: a hybrid approach, trying to be O-O as well as something completely different, cannot succeed at both since the concepts are too distant. Eiffel instead strives, by providing a coherent object framework — with such principles as Uniform Access, Command-Query Separation, Single Choice, Open-Closed and Design by Contract — to be a *component combinator* capable of assembling software bricks of many different kinds.

The following presentation describes how Eiffel systems can integrate components from other languages and environments.

The more frequent case of external interfaces is *call-out*: Eiffel routines calling non-Eiffel ones. The reverse need (foreign to Eiffel, or *call-in*) also exists. The mechanisms described in this chapter cover both.

Many applications will be happy enough to use the pure Eiffel mechanisms described in the rest of this book, and will not require any direct interfaces with other languages. (The next section explains what circumstances may including foreign software in an Eiffel system.) If you are mostly interested in understanding the techniques of Eiffel proper, you should probably get familiar with the principles of external calls by reading this section and the next four, and move on to the next chapter.





If you do study the details, you will note that they include, particularly in the specific external sublanguages supporting interaction with C, C++ and Dynamic Link Libraries, a number of specific mechanisms that may appear too rich when compared to the general sobriety of Eiffel's design. Do not be put off by this wealth of possibilities; the aim is not to complicate Eiffel but to enable Eiffel developers to take full advantage of non-Eiffel software at minimum effort. Any new, advanced technology such as Eiffel must provide effective bridges to older technologies, so that its users can leverage off existing investment. In particular, having powerful C and C++ interface sublanguages won't detract you from the simplicity of Eiffel programming; the effect instead will be that if you *do* have to interface with C and C++ you will be able to do everything you need on the Eiffel side, rather than having to write special "glue code" in those languages. Eiffel programmers, remarkably, prefer to program in Eiffel; carefully crafted interface sublanguages enable them to talk freely to the rest of the world without having to leave their language, techniques and tools of choice.

"Even under extreme duress, 99.9873% of Eiffel programmers still choose Eiffel", in Proc. of STOOP-SOLOW (joint meeting of Society for Torturing Object-Oriented Programmers and Society for Observing the Limits of Object Work), Sing-Sing (NY), Jan. 2001, pp. 5670-8782.

In accordance with the terminology used for the different forms of **Routine_body** in the syntax specifications, the discussion will use the term **internal routine** for any Eiffel routine accessible to language processing tools, and **external routine** for other routines. The name "external" refers to the routine as viewed from the Eiffel text; the form of the routine as it appears in its original language will be called the **foreign routine**.

→ In special cases the "other" language might be Eiffel itself. See below.



The semantic specifications presented in this chapter involve the semantics of languages other than Eiffel. Granting non-Eiffel software access to Eiffel objects may defeat the properties guaranteed by the semantic rules of this book. You should exercise care to confine the foreign languages to their proper role, avoiding unwanted interference with Eiffel object structures and algorithms.

31.2 WHAT EIFFEL CAN DO WITH THE REST OF THE WORLD

Here is some of what you can do with the foreign language facilities described in this chapter.

- You may declare an Eiffel routine as **external**, specifying that it comes from a foreign language. To the rest of the Eiffel software, the routine looks as if it were a normal Eiffel routine; but calls to it will execute the foreign code, which must of course have been compiled by a compiler for the foreign language. This is possible in principle for any foreign language, and guaranteed for C, C++, Java and Fortran 95.
- You may specify that an external routine, known in Eiffel under a certain name, had **another name** in its native language, for example if that name is not legal in Eiffel.
- You may specify that an external routine is actually implemented by a **C macro**, avoiding the overhead of function calls.

- You may associate a function and a procedure — a “getter” and a “setter” — to a C structure (“**struct**”), so that a call to the function will automatically access, and a call to the procedure modify, a specified field of that structure.
- You may even **include C code inline** in the body of an external routine, so that the external routine is in this case “internal” in the sense that it is specified within the Eiffel code, rather than elsewhere.
- You may use from Eiffel the routines of a **DLL** (Dynamic Link Library). You may specify the library and routines in your Eiffel text or, to make the process fully dynamic, you may obtain or compute this information at run time, just when you need to access the DLL elements.
- You may use from Eiffel all the facilities of a C++ class: **member functions, static members, data members, constructors, destructors**.
- You may use the *Legacy++* tool to produce a **C++ class wrapper**: an Eiffel class, automatically generated, that makes *all* the facilities of a C++ class (as listed above: member functions, data members and so on) available to the rest of the Eiffel system.
- Going the other way around, you may use the *Cecil* library to let external software do everything with an Eiffel system that you can do in Eiffel: create Eiffel objects, call on them any of the features of the corresponding classes, and so on. In other words Cecil lets you treat an Eiffel system as a **package** that the rest of the world can use as a library.
- That library can be dynamic: you can **generate a DLL** from an Eiffel system.
- You can also generate **COM components** (for Microsoft’s Component Object Model) and even XYZ components for execution on the XYZ virtual machine.

The next sections describe these mechanisms in detail, after a brief review of the proper role of foreign software elements in the development process.

31.3 WHEN TO USE EXTERNAL SOFTWARE



Why use external software? After all, Eiffel is a complete programming language, and many systems do not need any external software.

Four cases, however, may require interfacing Eiffel classes with software written in other languages:

- 1 • Reuse of older software elements.
- 2 • Use of libraries written in other languages.
- 3 • Access to low-level platform-dependent properties.
- 4 • Use of Eiffel as a tool for re-engineering of software.

Both cases [1](#) and [2](#) result from the obvious observation that Eiffel developments do not proceed alone in the software world, but must be combined with other products. In case [1](#), an organization may want to reuse previously developed elements as part of a new system. In case [2](#), the system will use existing primitives providing facilities in a specialized area — graphics, databases, user interfaces, expert systems...

In case [3](#), you need to access primitives which depend on the hardware or the operating system, available through external routines.

In case [4](#), an older non-Eiffel system must be converted to more modern software technology, but you want to proceed in stages. A possible strategy is to start by isolating appropriate abstractions in the existing software, and to build classes around them; the architecture of the resulting system will be expressed in Eiffel, using the structural mechanisms described in this book — classes, information hiding, genericity, inheritance, assertions — but the actual computations will still be performed by external routine calls. Here Eiffel serves as a packaging mechanism more than as a down-to-details programming language. This effort may be a first step towards more thorough re-engineering of the software, encompassing the internals as well as the structure. This is not an all-or-nothing decision: you may redo some of the components in Eiffel, for example the most advanced or innovative ones, and leave some others in the original language if they are stable and satisfactory.

The external facilities, detailed in the rest of this chapter, include:

- The possibility of specifying a routine as **External**, to indicate that it is written in another language and compiled separately; this notion will occupy the major part of the discussion.
- As a special case of the **External** mechanism, the C-Eiffel Interface Sublanguage, and the corresponding C++ facilities, enabling Eiffel software to take advantage of special foreign facilities such as C's macros and C++'s constructors (next section).
- The **Legacy++** tool for automatic Eiffel wrapping of C++ classes.
- **Cecil**, the C-Eiffel Call-In Library, allowing other languages to use almost all of Eiffel's facilities. (The initial C is in the acronym for historical reasons, but Cecil can be used from any other language.)

31.4 REGISTERED LANGUAGES AND THE ROLE OF C

Eiffel's external facilities depend in part — especially in the call-in case — on the properties of external languages; short of covering every programming language in existence, the specification cannot be exhaustive. It includes explicit knowledge about a few languages, said to be the **registered languages**, currently C, C++, Java, Fortran 95 and Eiffel itself. Any Eiffel compiler must support an interface to the registered languages, as described in this chapter.

Including Eiffel among the registered “foreign” languages is more a matter of completeness than of obvious necessity. Although in principle this allows you to integrate previously compiled Eiffel classes as if they were external software, better ways are usually available; a good Eiffel environment should be able to treat such classes like other Eiffel classes and perform all the relevant type checking. Another possible use of Eiffel as registered foreign language is to integrate Eiffel classes compiled with another compiler, although better interoperability mechanisms are desirable.

Among the registered languages, C, and its more recent variant C++, play a particular role for a number of technical, political and historical reasons:

- Since the mid-nineteen seventies, C has become the low-level *lingua franca* of computing, available on almost all platforms and known to a growing majority of programmers.
- Almost all dominant operating systems are written in C sometimes with more recent additions in C++.
- Most programs — from operating systems and database management systems to graphical libraries, object request brokers and other component-based development tools, development environments and many others — provide an Application Programming Interface (API) for C programs if they provide an API at all. When they offer more than one API, the one for C is often the reference. So a carefully engineered C binding is critical for many industrial developments.
- C compilers have benefited from wide use and several decades of research on compilation technology, aimed at producing efficient code.
- Although C has undergone changes, source code portability is reasonably good for programmers who follow some basic precautions.
- Many Eiffel implementations, such as ISE Eiffel, compile to C, taking advantage of the preceding properties, in particular wide availability, portability, and efficient code generation.
- A high-level language, Eiffel needs a good intermediary to access facilities from the machine and the operating system. C, more effective as a tool for use by *programs* than by humans, plays that role quite well. Libraries such as EiffelBase go to C when they occasionally must get out of the high-level language framework to access the nuts and bolts of the machine. C then plays for the Eiffel programmer exactly the same role that assembly language plays for the C programmer.

For all these reasons a special set of facilities — almost a mini-language within Eiffel, the *C-Eiffel Interface Sublanguage* — is available for those programmers who need fine-tuned access to C mechanisms from Eiffel. The Sublanguage allows you for example to use C macros, “structs”, include files, C dynamic link libraries (DLLs), or even to include *inline C code* in Eiffel routines. → “[THE CINTERFACE SUBLANGUAGE](#)”, 31.11, page 842.

Similar possibilities also exist for C++, giving Eiffel access to the components of C++ classes — member functions, constructors, destructors — and complemented by the automatic Legacy++ wrapper. → “[THE C++ INTERFACE SUBLANGUAGE](#)”, 31.12, page 847.

The role of these facilities is quite clear: to take the best advantage of C software, while writing *as little C as possible*. Eiffel programmers prefer writing Eiffel. They know that the world isn’t all Chanel perfumes and candlelight dinners, and that once in a while one must tender to the more mundane necessities of life. But then they expect the Eiffel compiler, through the Eiffel-C interface, to do much of the grunt work, and limit their use of C to the indispensable minimum.

31.5 BASICS OF EXTERNAL ROUTINES

We now start the study of the basic foreign affairs construct, **External**.

As seen in the [discussion of routines](#), the **Routine_body** of an **Effective routine**, instead of using the more common **Internal** form (beginning with **do** or **once**), may be of the **External** form, which indicates that a call to the routine is a call to some outside software component. ← **Routine_body** was discussed in [8.5, page 222](#). The syntax is on [page 222](#). The syntax for **External** appeared on [page 829](#); it is reproduced below.

An **External** clause begins with the keyword **external**, followed by a **Manifest_string** indicating the language in which the routine is written. It may also contain an **External_name** subclause, beginning with **alias**, giving the routine’s name in its language of origin (or, in the case of inline C routines, the actual C text).

Here is an example of external routine



```
f_close (filedesc: INTEGER): INTEGER
    -- Close file associated with filedesc;
    -- record status in result.

require
    descriptor_exists: exists (associated_file (filedesc))

external
    "C"

ensure
    zero_iff_ok:
        (Result = 0) = closed (associated_file (filedesc))

end
```

As this example shows, an external routine may have a **Precondition** and a **Postcondition**.



Function `f_close` performs a certain action and returns a status report through its result. This technique is not normally employed by Eiffel functions, which should instead record the status in an attribute; in communicating with external software, however, there may be no better way.

You may use an **External_name** subclause, beginning with **alias**, to refer to an external routine through a name other than the one it has in the foreign language. For example:



```
file_status (filedesc: INTEGER): INTEGER
  external
    "C"
  alias
    "_fstat"
  end
```

The **alias** specifies that any call to `file_status` will cause a call to the C function of name `_fstat`. There are two possible reasons for such a subclause:

- The native name may be legal in the foreign language but not in Eiffel, as in the `file_status` example where the function name `_fstat`, legal in C, is illegal in Eiffel since it starts with an underscore.
- Even if the foreign name abides by Eiffel rules, it may violate the naming conventions of your project.



In the absence of an **alias** subclause, the feature name passed to the external software is the **lower name** of the feature.

← The lower name is the name all in upper case. See [“TEXTUAL CONVENTIONS”](#), 2.13, page 102.



So even if you give to an external feature a name following the letter case conventions of another language, such as `SetValue` for an external routine implemented in C, the name passed to C will be `setvalue`. Even if it is implemented as an external routine, an Eiffel feature should follow Eiffel conventions: call it `set_value` and use **alias** `"SetValue"`.

Here is the basic syntax of **External** routine bodies:



```
External routines
  External ≙ external External_language [External_name]
  External_language ≙ Unregistered_language |
    Registered_language
  Unregistered_language ≙ Manifest_string
  External_name ≙ alias Manifest_string
```

The **External** clause is the mechanism that enables Eiffel to interface with other environments and serve as a “component combinator” for software reuse and particularly for taking advantage of legacy code.

By default the mechanism assumes that the external routine has the same name as the Eiffel routine. If this is not the case, use an **External_name** of the form **alias** "ext_name". The name appears as a **Manifest_string**, in quotes, not an identifier, because external languages may have different naming conventions; for example an underscore may begin a feature name in C but not in Eiffel, and some languages are case-sensitive for identifiers whereas Eiffel is not.

Instead of calling a pre-existing foreign routine, it is possible to include **inline** C or C++ code; the **alias** clause will host that code, which can access Eiffel objects through the arguments of the external routine.

The language name (**External_language**) can be an **Unregistered_language**: a string in quotes such as "**Cobol**". Since the content of the string is arbitrary, there is no guarantee that a particular Eiffel environment will support the corresponding language interface. This is the reason for the other variant, **Registered_language**: every Eiffel compiler must support the language names "**C**", "**C++**" and **dll**. Details of the specific mechanisms for every such **Registered_language** appear below.

Some of the *validity* rules below include a provision, unheard of in other parts of the language specification, allowing Eiffel language processing tools to rely on *non-Eiffel tools* to enforce some conditions. A typical example is a rule that requires an external name to denote a suitable foreign function; often, this can only be ascertained by a compiler for the foreign language. Such rules should be part of the specification, but we can't impose their enforcement on an Eiffel compiler without asking it also to become a compiler of C, C++ etc.; hence this special tolerance.

The general *semantics* of executing external calls appeared as part of the general semantics of calls. The semantic rules of the present discussion address specific cases, in particular inline C and C++.

Although you may intermix routines of the **External** and **Internal** forms, it is common practice to separate the two categories, grouping external routines into their own **Feature_clause**. In some cases you will even find “wrapper” classes consisting mostly or entirely of external routines, encapsulating a set of external facilities into an abstraction usable directly by the rest of the Eiffel software.



31.6 EXECUTING AN EXTERNAL CALL

Before exploring the varieties of foreign interfacing mechanisms, we must understand the precise semantics of external calls, previewed in the general discussion of call semantics. Only three aspects differ from the semantics of **Internal** routines:

- 1 • Actual-formal argument association.
- 2 • Value to be returned, if the routine is a function.
- 3 • Execution of the **Routine_body**

The next section will cover items [1](#) and [2](#). Item [3](#), the simplest, was handled by the general [discussion of call semantics](#). Quoting: --- CHECK ---

← “[PRECISE CALL SEMANTICS](#)”, [23.17](#), [page 652](#).

If *df* is an external routine, the effect of the call is to execute that routine on the actual arguments given, if any, according to the rules of the language in which it is written.

Here *df* is the version of *f* to be applied to the given target, deduced from the rules of call semantics (dynamic binding).

In addition to its official arguments, an Eiffel routine has access to the **current object** – the target of the current call. This important property does not necessarily hold for a foreign routine:

← *The notion of current object was defined on [page 649](#).*

- If the foreign routine was written independently of Eiffel, it does not use the current object. Accordingly, the call, as specified by the above semantics, will not pass the current object. A typical case is a call to a primitive of a pre-existing graphics or database package.
- Another case is that of foreign routines specifically written for the needs of an Eiffel application. Such routines may need access to the current object; you must then explicitly pass *Current* as one of the arguments.

← “[Current object and routine](#)”, [page 648](#).

31.7 ARGUMENT AND RESULT TRANSMISSION

The semantics of passing arguments, and of returning the result for a function, raises the problem of attachment between Eiffel values and foreign entities.

For internal routines, the [semantic rule](#) was simple, being deduced (like the semantics of **Assignment** instructions) from the semantics of the direct reattachment mechanism: at call time, each formal argument becomes attached to the corresponding actual; at return time, the result of a function is the final value attached to the function’s *Result* entity.

← “[PRECISE CALL SEMANTICS](#)”, [23.17](#), [page 652](#). The semantics of direct reattachment was in [22.7](#), [page 593](#).

The semantic specification of a direct reattachment allowed flexible combinations of expanded and reference types in the source and target. Here is the table which gave the effect in all four possible cases:

<i>SOURCE</i> → <i>TARGET</i> ↓	Reference	Expanded
Reference	[1] Reference reattachment	[3] Clone
Expanded	[2] Copy (fails if source void)	[4] Copy

← This table originally appeared on page 596.

This specification takes both types – source and target – into account, particularly in cases 2 and 3 where one is expanded and the other is not.

For external calls, however, we cannot afford such semantic flexibility, since the target is the formal argument, and we have no way of knowing how the foreign routine has declared it. The semantic definition must rely on properties of the actual argument alone.

To depart as little as possible from the rules for internal routines, the convention for external routines, follow the semantics of direct reattachment, interpreted as if each formal argument were declared with **exactly** the same type as the corresponding actual.

This implies that only cases 1 and 4 of the above table make sense: either the actual argument is of a reference type, in which case the foreign routine will receive a reference, or it is of an expanded type, in which case the foreign routine will receive a copy of the attached object.

*This also applies to **Current** if it is one of the actual arguments: with the semantics of **Current**, defined by case 2, page 652, what is passed is a reference to the current object if the enclosing class is non-expanded, otherwise the current object itself.*

For the result of a function, the rule is similar: depending on the type declared for the function’s result, the Eiffel side will expect the foreign routine to return a reference or an object.

Clearly, using foreign routines which will handle Eiffel values requires care. You must trust that the routine can manipulate the values it obtains from the Eiffel side, and, if it is a function, produces results which conform to what you expect. So the types of arguments and result must be common to Eiffel and the external language.

For basic types, this property depends on both the foreign language and its implementation.

→ The basic types (chapter 30) are **BOOLEAN**, **CHARACTER**, **INTEGER**, **REAL**, their sized variants and **POINTER**.

For other types, no major problem will arise for a foreign routine which, given an object or reference, just needs to do a “store and forward”: pass on the value to other routines, possibly keeping a copy in a variable of a suitable type. To do anything more with an Eiffel object, the routine must access its internal structure; it may avoid relying on implementation-dependent properties of object representation by using one of the following two portable mechanisms:

- The features of class **INTERNAL** from EiffelBase provide access to the internal properties of objects (such as the various field values) with an implementation-independent interface.



- The Cecil library, described at the end of this chapter, allows foreign languages to access Eiffel features.

31.8 PASSING THE ADDRESS OF AN EIFFEL FEATURE

In some cases a foreign routine may need to call Eiffel routines, or to access fields of Eiffel objects.



Foreign access to Eiffel routines may be necessary in particular for the implementation of so-called **callback** mechanisms as they appear in such areas as user interfaces, graphics and databases. Callback enables routines to “plant” the address of one or more routines into another routine *r* at initialization time. Later, at various places in its own algorithm, *r* will call the planted routines. Because planting is dynamic, the text of *r* does not show what actual routines will be called at the corresponding steps; it only contains “holes” where different applications may plant different routines. Often, *r* is a high-level loop, known as an **event loop**, which will repeatedly execute ritual actions (such as reading user input or updating the screen) through the planted routines.

In this description, you will have recognized the notion of **iterator** discussed in the presentation of inheritance and deferred features; indeed, the Eiffel techniques introduced for iterators, relying on deferred routines and dynamic binding, offer simpler, safer and more elegant alternatives to call-back. But you may need to use an existing call-back mechanism implemented in another language, with individual planted operations to be provided by Eiffel features. So you need the ability to pass to an external routine the address of an Eiffel feature.

← On how to implement a call-back mechanism in Eiffel, see [10.15, page 277](#).

The supporting construct is the **Address** form of Actual argument. An **Address**, introduced as part of the syntax for **Actuals** in the discussion of calls, is simply an actual argument of the form

← The syntax for **Actuals** appeared on page [626](#).

$\$feature_or_parenthesized_expression$

Here *feature_or_parenthesized_expression* can be the name of an Eiffel feature, a parenthesized expression such as $(a + b)$, as well as *Current* or, in a function, *Result*. In all cases what is passed is an address. For a feature this enables the foreign software to call the feature; for an expression it gives it access to a location containing the value of the expression. The latter is useful for a foreign routine that expects not a value but an address containing that value.



This **Address** form of **Actual** argument is only useful for passing such addresses to external routines. **Internal** (Eiffel) routines do not need it, since the dynamic binding mechanism provides a better way to tell a supplier what feature it should call at a certain stage of the supplier's execution: you just pass the supplier an entity attached to a certain object; the dynamic type of that object, which may vary from one execution to the next, determines the applicable routine versions.

← On dynamic binding, see [23.12](#) and [23.13](#), starting on page [638](#).

Here is the syntax for an **Address** argument:

```
Address ≙ "$" Address_mark
Address_mark ≙ Variable
```

Feature_name is the most common case.

As to the validity constraint, we saw it as part of the **Argument rule**, which makes $\$f$ valid as actual argument to a call if and only if f , when an **Extended_feature_name**, is the final name of a feature of the class.

← Page [634](#).

An **Address** argument, as noted, describes the address of a routine or expression. It is subject to a constraint:



Address rule

VZAR

An **Address** is valid if and only if its **Address_mark** is of a reference type.

==== DISCARD ==== An expanded type would not make sense here as its values have copy rather than reference semantics.

How do we describe an “address” in Eiffel? A basic type is available for that purpose: **POINTER**, described by a Kernel Library class. Hence the type rule:



Address Type rule

An argument of the **Address** form is of type **POINTER**.

As a consequence, the declaration for the corresponding formal argument in the receiving routine must be of the form

```
ir2 (...; from_eiffel: POINTER; ...) is ...
```



or the corresponding declaration in a foreign language.



Note that this routine can indeed be an **Internal** Eiffel routine as well as an external one. Although you might expect **Address** actual arguments to be permitted only in calls to external routines, there is no such constraint: it may be useful for an **Internal** routine *ir1* to pass the address of a routine *r* to another internal routine *ir2*, so that *ir2* may itself pass *r* to an external routine *er*. Were this not permitted, *ir1* would need to call *er* directly, which may not be the desired scheme.

*The hypothetical constraint, an addition to the argument validity rule of page 634, would require the called routine *df* to be external.*

We must prevent *ir2* from performing any operation on its argument *r* other than passing it along to another routine. This simply follows from the properties of class **POINTER**, which has no exported features except for the universal, harmless features *copy*, *clone*, *equal* and consorts from **ANY**. So all you can do on an argument of type **POINTER** — other than copying it, cloning it, comparing for equality and so on — is to pass it on to someone else.

Address semantics

The value of an **Address** expression is an address enabling foreign software to access the associated **Variable**.

The manipulations that the foreign software can perform on such addresses depend on the foreign programming language. It is the implementation's responsibility to ensure that such manipulations do not violate Eiffel semantic properties.

--- REWRITE (MOSTLY REMOVE) THE REST OF THIS SECTION



Now the semantics of an **Address** argument $\$f$ being passed to a routine *r*. We must distinguish between the possible cases for *f*:

- 1 • If *f* is an **Extended_feature_name** (as noted, the most common case), the corresponding feature have a version *df* applicable to the current object, taking into account possible renaming and redefinition. *df* is the feature that a call *x.f (...)* would execute, according to the rules of dynamic binding, when *x* is attached to an object of the current type. The value passed to *r* is the address of *df*. This applies to both routines and variable attributes; for an attribute, the call will pass the address of the field corresponding to *df* in the current object. Clearly, this is useful only if the foreign language can deal with addresses of fields and routines.
- 2 • If *f* is a constant attribute or a **Parenthesized** expression, what is passed to the routine is the address of a memory location containing its value.
- 3 • If *f* is **Current**, the value passed is the address of the current object.

- 4 • If f is *Result*, the value passed is the address used to store the result to be returned by the enclosing function.

In case **1**, where f denotes a feature, foreign software elements will be able to call that feature. Such calls require one extra argument, appearing at the first position and corresponding to the target of the call. Assume



```
some_routine (a1: A; b1: B) is...
```

Calls to *some_routine* in Eiffel texts may be qualified or unqualified:

```
target.some_routine (x, y)  
some_routine (x, y)
```

Assume now that a call to an external routine *ext* makes the address of *some_routine* available to a foreign language:

```
ext (... , $ some_routine , ...)
```

Let *sr* be the formal argument for *some_routine* in the foreign routine corresponding to *ext*. The foreign routine will call *some_routine* with one extra actual argument, appearing at the first position:

```
sr (target, x, y)  
sr (current_object, x, y)
```

*These calls to **sr** appear here in Eiffel syntax, but the convention for calls in the foreign language may be different.*

The extra argument denotes the call's target, which in Eiffel appeared before the dot (as in the case of *target*) or not at all (as with *current_object*). It denotes an object or object reference.

The above calls to *sr* from a foreign language are examples of what what the beginning of this chapter defined as the **call-in** case: exercising Eiffel mechanisms from the outside. To take this scheme to its full realization the foreign software needs:

- A way to manipulate Eiffel objects safely (protecting them, in particular, from the Eiffel garbage collector).
- A clear correspondence between the types of Eiffel and those of the foreign language.
- An adequate calling mechanism for features.

The Cecil library, described later in this chapter, provides all of this. But we are not ready yet to move on to call-in facilities, since we are not finished with call-out. In addition to the language-independent call-out constructs just studied, Eiffel's external interface offers special support for C and C++ — languages important enough to deserve mini-sUBLANGUAGES of their own in the Eiffel syntax for **External** features. → *“THE CECIL LIBRARY”, 31.16, page 865.*

31.9 SPECIAL INTERFACE SUBLANGUAGES

We saw that the syntax for declaring a routine as **External** involves a language name: ← *This syntax appeared first on page 829.*



External languages

External \triangleq **external** External_language [External_name]

External_language \triangleq Unregistered_language | Registered_language

Unregistered_language \triangleq Manifest_string

External_name \triangleq **alias** Manifest_string

The **External_language** may be an **Unregistered_language** — a plain **Manifest_string** describing an arbitrary language; this is useful only if that language is known to your specific Eiffel compiler, or uses default argument passing conventions that will work with Eiffel. But it may also be a **Registered_language**, covering DLL routines, which may come from any language, and the four languages guaranteed to be handled properly:



Registered languages

Registered_language \triangleq C_external | C++_external |
DLL_external

IL_external refers to the Intermediate Language of the Microsoft .NET framework.

The cases of , **C_external**, **C++_external** and **DLL_external** give rise to special sublanguages with a host of detailed possibilities, reviewed in the next three sections. Note that all the C possibilities are also available for C++, so in practice the third sublanguage is a superset of the second.

31.10 GENERAL SUBLANGUAGE MECHANISMS

The specific sublanguages — **C_external**, **C++_external** and **DLL_external** — offer common techniques for specifying certain elements:

- Routine signatures.
- Files needed to use the external software, for example C include files or the files containing a DLL.
- Types used to establish a precise correspondence between the type systems of Eiffel and those of other languages (for example, between an Eiffel *INTEGER* and a C *int*).

Before going into the specific sublanguages, let us review these shared facilities in turn.

Specifying an external routine signature

Since external languages have their own type systems, you may need to specify that a certain routine expects certain types for its arguments. In languages such as C and C++ that support “casts” (forced conversions), these types will be used for casting the arguments.

To specify types in the relevant sublanguages you may include an *External_signature* in the string specifying the language, as in the C external function declaration



```
your_external (a, b: INTEGER): INTEGER
  external
    "C signature (int, int)"
  end
```

The *External_signature* part in this example is

```
(int, EIF_INTEGER_32)
```

indicating that the associated C function expects two arguments of the C type *int* (integer). The names listed must be types of the external language, such as *int* for a C routine. *EIF_INTEGER_32* is a type used for the correspondence between Eiffel and C types, as explained in a [later section](#).

→ “[Controlling the Eiffel-C type correspondence](#)”, page 846.

It doesn’t matter that *int* and *EIF_INTEGER_32* are not valid Eiffel type names: remember that an *External_signature* such as the above, like everything else in the sublanguages under discussion, appears in a string.



As you will have noted, the *External_signature* only lists types for arguments; for a function, you cannot specify a type, because the compiler will make sure that the function’s result is converted back to the result type specified for the Eiffel routine. (In this respect the construct name *External_signature* and the keyword *signature* are a little misleading, since elsewhere in the description of Eiffel the word “signature” covers both result and argument types, but it still seems to be the best name here.)

The syntax of `External_signature` is straightforward:



External signatures	
<code>External_signature</code>	\triangleq signature [<code>External_argument_types</code>] [: <code>External_type</code>]
<code>External_argument_types</code>	\triangleq "(" <code>External_type_list</code> ")"
<code>External_type_list</code>	\triangleq { <code>External_type</code> "," ... }*
<code>External_type</code>	\triangleq <code>Simple_string</code>

The `External_signature`, if at all present, must cover all arguments:



External Signature rule		<i>VZES</i>
An <code>External_signature</code> in the declaration of an external <u>routine</u> <i>r</i> is valid if and only if it satisfies the following conditions:		
1 • Its <code>External_type_list</code> contains the same number of elements as <i>r</i> has formal arguments.		
2 • The final optional component (: <code>External_type</code>) if present if and only if <i>r</i> is a <u>function</u> .		
A <u>language processing tool</u> may delegate enforcement of these requirements to non-Eiffel tools on the chosen <u>platform</u> .		

The rule does not prescribe any particular relationship between the argument and result types declared for the Eiffel routine and the names appearing in the `External_type_list` and the final `External_type` if any, since the precise correspondence depends on foreign language properties beyond the scope of Eiffel rules.

→ On this correspondence in the C case, see "[Controlling the Eiffel-C type correspondence](#)", page 846.

The specification of a non-external routine never includes C-style empty parenthesization: for a declaration or call of a routine without arguments you write *r*, not *r* (). The syntax of `External_argument_types`, however, permits () for compatibility with other languages' conventions.

The last part of the rule allows Eiffel tools to rely on non-Eiffel tools if it is not possible, from within Eiffel, to check the properties of external routines. This provision also applies to several of the following rules.

External signature semantics

An `External_signature` specifies that the associated external routine:

- Expects arguments of number and types as given by the `External_argument_types` if present, and no arguments otherwise.
- Returns a result of the `External_type` appearing after the colon, if present, and otherwise no result.

Specifying external files

To use an external routine, you may need to provide one or more file names:

- A C or C++ function may rely on some “include files”; for example, the type `EIF_INTEGER_32` used by *your_example* above must have a C definition, to which the C function must have access. It will find it in an include file, which you may specify from the Eiffel side.
- To use an external routine from a DLL, you must indicate the file that contains the DLL.

An `External_file_use` part, starting with `use`, enables you to say which files you need. Here is its application to the preceding example, assuming you want function *your_external* to have access to two C include files:



```
your_external (a, b: INTEGER): INTEGER
  external "[
    C
    signature (int, int)
    use <stdio.h>, "/path/user/her_include.h"
  ]"
end
```



This example and several that follow use a multi-line `Verbatim_string`, written between an opening `"["` and a closing `"]"`. We could also use a plain string without this convention, but then the internal double quote signs `"`, in the specification of the path name, would have to be written `%"`; also, interrupted lines would need to finish with a `%`, and continuation lines to start with a `%`.

← “*MANIFEST STRINGS*”, 29.8, page 794.

Here is the syntax of `External_file_use`:



External file use

```
External_file_use  $\triangleq$  use External_file_list
External_file_list  $\triangleq$  {External_file " , " ... }+
```

```

External_file  $\triangleq$  External_user_file | External_system_file
External_user_file  $\triangleq$  ' "' Simple_string ' "'
External_system_file  $\triangleq$  "<"Simple_string ">"

```

As the syntax indicates, you may specify as many external files as you like, preceded by **use** and separated by commas. You may specify two kinds of files:

- “System” files, used only in a C context, appear between angle brackets `<>` and refer to specific locations in the C library installation.
- The name of a “user” file appears between double quotes, as in `"/path/user/her_include.h"`, and will be passed on literally to the operating system. Do not forget, when using double quotes, that this is all part of an Eiffel **Manifest_string**: you must either code them as `%"` or, more conveniently, write the string as a **Verbatim_string**, the first line preceded by `"[` and the last line followed by `]"`.

An **External_file** refers to file and path names. Different operating systems have different conventions to denote paths; to avoid worrying about these differences, the examples of this chapter assume the Unix/Linux style using forward slash characters, as in `/path/usr/file.c`. This convention is also understood by most C compilers on Windows, even though the native Windows style uses backslash characters, as in `d:\path\usr\file.c`. VMS has its own notation.

The difference between the two forms of **External_file** is that a **C_user_file**, of the form `"path_name"`, denotes a file through its exact location in the file system, whereas a **C_system_file** of the form `"<file_name>"` is relative to the location of standard include files — such as `stdio.h` for standard C input and output — in the C installation.

In either case, any files listed must exist and have the expected contents:



External File rule

VZEF

An **External_file** is valid if and only if its **Simple_string** satisfies the following conditions:

- 1 • When interpreted as a file name according to the conventions of the underlying platform, it denotes a file.
- 2 • The file is accessible for reading.
- 3 • The file’s content satisfies the rules of the applicable foreign language.

A language processing tool may delegate enforcement of these conditions to non-Eiffel tools on the chosen platform.

Condition [3](#) means for example that if you pass an include file to a C function the content must be C code suitable for inclusion by a C “include” directive. Such a requirement may be beyond the competence of an Eiffel compiler, hence the final qualification enabling Eiffel tools to rely, for example, on compilation errors produced by a C compiler.

The “conventions of the underlying platforms” cited in condition [1](#) govern the rules on file names (in particular the interpretation of path delimiters such as `/` and `\` on Unix and Windows) and, for an `External_system_file` name of the form `<some_file.h>`, the places in the file system where `some_file.h` is to be found.

External file semantics

An `External_file_use` in an external routine declaration specifies that foreign language tools, to process the routine (for example to compile its original code), require access to the listed files.

31.11 THE C INTERFACE SUBLANGUAGE

The first special sublanguage that we study, `C_external`, addresses the needs of applications developers who need sophisticated access to C mechanisms (also provided for C++). You can of course limit yourself to the mechanisms described so far, simply declaring an external routine as `external "C"`. But to exert more control on how your Eiffel software uses C mechanisms, you may use a whole slate of special C interface facilities:

- You can specify that a certain external routine is implemented on the C side as a **macro**, saving the overhead of function calls.
- You can use an `External_signature`, as studied above, to force a certain type signature (“*prototype*”) for the arguments and result of the C function in the Eiffel-generated C code.
- You can request specific **include files** for certain C functions, using the `External_file_use` construct just studied.
- You can directly access C structures (“structs”) and their components.
- You can even include the C code of an external routine in line, removing the need to maintain two separate source files, an Eiffel class file and a C compilation unit (`.c` file).

The next paragraphs describe these possibilities. They are complemented by the C++-specific facilities of the following section.

Syntax specification

Here is the syntax specification for the C interface sublanguage. First we remind ourselves of the context:



	External languages
External	\triangleq external External_language [External_name]
External_language	\triangleq Unregistered_language Registered_language
Unregistered_language	\triangleq Manifest_string
External_name	\triangleq alias Manifest_string
Registered_language	\triangleq ... C_external ... Others ...

← This appeared first on page [829](#).

Now the C_external case of Registered_language:



	C externals
C_external	\triangleq ' " ' C ' inline [External_signature] [External_file_use] ' " '

The `C_external` mechanism makes it possible, from Eiffel, to use the mechanisms of C. The syntax covers two basic schemes:

- You may rely on an existing C function. You will not, in this case, use `inline`. If the C function's name is different from the lower name of the Eiffel routine, specify it in the `alias (External_name)` clause; otherwise you may just omit that clause.
- You may also write C code *within* the Eiffel routine, putting that code in the `alias` clause and specifying `inline`.

In the second case the C code can directly manipulate the routine's formal arguments and, through them, Eiffel objects. The primary application (rather than writing complex processing in C code in an Eiffel class, which would make little sense) is to provide access to existing C libraries without having to write and maintain any new C files even if some "glue code" is necessary, for example to perform type adaptations. Such code, which should remain short and simple, will be directly included and maintained in the Eiffel classes providing the interface to the legacy code.

The `alias` part is a `Manifest_string` of one of the two available forms:

- It may begin and end with a double quote `"`; then any double quote character appearing in it must be preceded by a percent sign, as `%"`; line separations are marked by the special code for "new line", `%N`.
- If the text extends over more than one line, it is more convenient to use a `Verbatim_string`: a sequence of lines to be taken exactly as they are, preceded by `"[` at the end of a line and followed by `]"` at the beginning of a line.

← See "[MANIFEST STRINGS](#)", 29.8, page 794.

In this `Manifest_string`, you may refer to any formal argument *a* of the external routine through the notation `$a` (a dollar sign immediately followed by the name of the argument). For *a* you may use either upper or lower case, lower being the recommended style as usual.

We now explore these capabilities, and look further into how you can match Eiffel types with their C counterparts.

Specifying C code inline

In all the preceding mechanisms, the C code resides outside of the Eiffel text, in its own separate files. Although this separation of elements written in different languages is usually appropriate, you may not like the idea of having to look after different places, and find it easier to manage your software by keeping everything at the same place. It is indeed possible to include C code within the declaration of an external routine. This way you don't need to include any external C file in your system.

This possibility is appropriate mostly for short C routines concentrated in "wrapper" classes providing Eiffel interfaces to C libraries.



A **C_special** part may specify **inline**, optionally followed by the usual specifications of a C signature and include files. This indicates that the actual C text appears in the **alias** clause (**External_name**), which is required in this case. Here is an example including both an explicit signature and an include file (which might contain the declaration of a C variable *cvar*):



```

an_inline_function (x,y: INTEGER): INTEGER
  external "[
    C
    inline
    use <stdio.h>, /path/user/her_include.h
  ]"
  alias "[
    if ($x > cvar) {
      some_c_function ($y, cvar++);
    }
  ]"
end

```

Warning: the content of the **alias** clause represents C, not Eiffel.

The **Manifest_string** appearing in the **alias** clause is C code meant to be passed on exactly as it is (except for the replacement of elements in quotes, as explained next) to a C compiler. The most convenient way to express it is to use, as here, a **Verbatim_string**, so that all the lines between the initial **"[** and the final **]"** are plain C text, with no need for special codes to represent characters such as quotes, or to mark the beginning and end of a line.

SEMANTICS

The only exception to the verbatim interpretation of the string as C code is the convention allowing the C code to access entities from the enclosing Eiffel text. Any occurrence in the **alias** part of a substring of the form *\$eiffel_entity*, where *eiffel_entity* is a formal argument of the routine or an attribute of the enclosing class, denotes the corresponding Eiffel entity, which the Eiffel compiler will replace by the appropriate access code for the benefit of the C compiler. *\$x* and *\$y* in the above extract are examples of this facility; they denote the function's *x* and *y* arguments.

This use of the **\$** operator is consistent with the **Address form** of arguments, serving to pass Eiffel features to external languages.

← "[PASSING THE ADDRESS OF AN EIFFEL FEATURE](#)", 31.8, page 833.

SYNTAX

Note that *eiffel_entity* must follow the **\$** sign with no intervening space. Any occurrence in the C text of a **\$** sign not immediately followed by an Eiffel entity is considered C text to be taken verbatim.

Here is the validity rule for inline C functions:



C external rule

VZCC

A **C_external** for the declaration of an external routine *r* is valid if and only if it satisfies the following conditions:

- 1 • At least one of the optional **inline** and **External_signature** components is present.
- 2 • If the **inline** part is present, the external routine includes an **External_name** component, of the form **alias** *C_text*.
- 3 • If case 2 applies, then for any occurrence in *C_text* of an **Identifier** *a* immediately preceded by a dollar sign \$ the lower name of *a* is the lower name of a formal argument of *r*.

C Inline semantics

In an external routine *er* of the **inline** form, an **External_name** of the form **alias** *C_text* denotes the algorithm defined, according to the semantics of the C language, by a C function that has:

- As its signature, the signature specified by *er*.
- As its body, *C_text* after replacement of every occurrence of \$*a*, where the lower name of *a* is the lower name of one of the formal arguments of *er*, by *a*.

Controlling the Eiffel-C type correspondence

EXAMPLES

In passing arguments to C functions, and getting results back into Eiffel entities, you need to know exactly how the types will match. Eiffel provides (through the C library of the supporting environments) a set of predefined C types used, by default, to represent the types of Eiffel values passed to and from external C routines. If you are writing external C functions specifically for use in connection with Eiffel software, you should use these types (obtained from a standard include file provided with the Eiffel delivery) to declare the functions' arguments and results.:

Eiffel type	Corresponding C type with declaration
<i>BOOLEAN</i>	typedef unsigned char EIF_BOOLEAN
<i>CHARACTER</i>	typedef unsigned char EIF_CHARACTER
<i>INTEGER_8</i>	typedef unsigned char EIF_INTEGER_8
<i>INTEGER_16</i>	(16-bit integer) EIF_INTEGER_16
<i>INTEGER</i>	(32-bit integer) EIF_INTEGER_32

*Eiffel to C
default type
correspondence*

<i>INTEGER_64</i>	(64-bit integer)	<i>EIF_INTEGER_64</i>
<i>REAL_32</i>	(32-bit float)	<i>EIF_REAL_32</i>
<i>REAL</i>	(64-bit integer)	<i>EIF_REAL</i>
<i>POINTER</i>	<code>typedef char *</code>	<i>EIF_POINTER</i>
Any reference type	<code>typedef char *</code>	<i>EIF_REFERENCE</i>

The C type definitions given in parentheses are platform-dependent. For example “32-bit integer” will be `typedef long` on many platforms, but not all.

This will not work, however, if you are using pre-existing C functions, written without knowledge of Eiffel. In such a case the declarations will not match those generated by the Eiffel compiler using the correspondence above, and you may get C compilation errors. Fortunately, the type checking of C is more bark than bite. You can easily pacify it by “casting” the type of arguments and results, that is to say, specifying explicit types.

It would be unpleasant to have to do the casting manually on the C code (if only because we are, as noted, trying through all the facilities described here to limit the amount of C programming to be done). The *External_signature* facility is here to help. It allows you to specify the exact set of casting types for the arguments and result, so that the C compiler will find what it expects. Here is a typical use:

```
your_external (a, b: INTEGER): INTEGER
  external
    "C (int, int): EIF_INTEGER_32"
  end
```

This example assumes that the C function requires arguments of the C type *int* (integer) and returns a result also of that type, which must be cast into an *EIF_INTEGER_32*.

31.12 THE C++ INTERFACE SUBLANGUAGE

In addition to the mechanisms available to all external routines, all the C-specific techniques of the previous sections are available for use with C++ code. So is the Cecil library described in a later section and allowing external software to call Eiffel. In addition, the C++ interface sublanguage offers a number of specific mechanisms:

- You can create instances of C++ classes from Eiffel, using the C++ “constructor” of your choice.
- You can apply to these objects all the corresponding operations from the C++ class: executing functions (“methods”), accessing data members, executing destructors.

- You can use the **Legacy++** tool to produce an Eiffel “wrapper class” encapsulating all the features of a C++ class, so that the result will look to the rest of the Eiffel software as if it had been written in Eiffel.

The syntax specification

The C++-specific mechanisms come under the construct `C++_external`, one of the variants of `Registered_language`, itself one of the possibilities for `External_language`.



C++ externals

```

C++_external ≙ ' "' C++
                inline
                [External_signature]
                [External_file_use]
                ' "'

```

As in the C case, you may directly write C++ code which can access the external routine’s argument and hence Eiffel objects. Such code can, among other operations, create and delete C++ objects using C++ constructors and destructors.



Unlike in the C case, this inline facility is the *only* possibility: you cannot rely on an existing function. The reason is that C++ functions — if not “static” — require a target object, like Eiffel routines. By directly writing appropriate inline C++ code, you will take care of providing the target object whenever required.

Conditions on C++ features



C++ external rule *VZC+*

A `C++_external` part for the declaration of an external routine *r* is valid if and only if it satisfies the following conditions:

- 1 • The external routine includes an `External_name` component, of the form **alias** *C++_text*.
- 2 • For any occurrence in *C++_text* of an `Identifier` *a* immediately preceded by a dollar sign \$, the lower name of *a* is the lower name of a formal argument of *r*.

Processing C++ features

A `C++_external`, if present, indicates one of the following, all illustrated by examples in the next sections:

- If the special feature’s declaration starts **function**, it indicates that the Eiffel feature will call a C++ *member function* (also known as a “method”) from the class listed. The function’s name is by default the same as the name of the Eiffel feature; as usual, you can specify a different name through the **alias** clause of the external declaration.
- If the declaration starts with **static**, it indicates a call to a C++ *static function*.
- If the declaration starts with **new**, it indicates a call to one of the *constructors* in the C++ class, which will create a new instance of that class and apply to it the corresponding constructor function.
- If the declaration starts with **delete**, it indicates a call to a *destructor* from the C++ class. In this case the Eiffel class will inherit from *MEMORY* and redefine the *dispose* procedure to execute the destructor operations whenever the Eiffel objects are garbage-collected.
- If the declaration starts with **data member**, it indicates access to a *data member* (attribute in Eiffel terminology) from the C++ class.
- If it starts with **structure**, it provides the same facilities as `C_structure`.

The techniques for specifying signatures, external files and type correspondence are the same as for C.

← “[Specifying an external routine signature](#)”, page 838; “[Specifying external files](#)”, page 840; “[Controlling the Eiffel-C type correspondence](#)”, page 846.

C++ Inline semantics

In an external routine *er* of the `C++_external` form, an External_name of the form **alias** `C++_text` denotes the algorithm defined, according to the semantics of the C++ language, by a C++ function that has:

- As its signature, the signature specified by *er*.
- As its body, `C++_text` after replacement of every occurrence of **\$a**, where the lower name of *a* is the lower name of one of the formal arguments of *er*, by *a*.

Extra argument

For a non-static C++ member function or destructor, the corresponding Eiffel feature should include an extra argument of type *POINTER*, at the first position. This argument represents the C++ object to which the function will be applied.

For example, a C++ function



```
void add (int new_int);
```

should have the Eiffel counterpart

```
cpp_add (obj: POINTER; new_int: INTEGER)
  -- Encapsulation of member function add.
  external "[
    "C++
      member IntArray
      signature (IntArray *, int)
      use intarray.h
    ]"
  end
```

This scheme, however, is often inconvenient because it forces the Eiffel side to work on objects in a non-object-oriented way. (The O-O way treats the current object, within a class, as implicit.) A better approach, used by Legacy++, is to make a feature such as *cpp_add* secret, and to export a feature whose signature corresponds to that of the original C++ function, with no extra object argument; that feature will use a secret attribute *object_ptr* to access the object. In the example this will give

```
add (new_int: INTEGER)
  -- Encapsulation of member function add.
  do
    cpp_add (object_ptr, new_int)
  end
```

where *object_ptr* is a secret attribute of type *POINTER*, initialized by the creation procedures of the class. To the Eiffel developer, *add* looks like a normal object-oriented feature, which takes only the expected argument. Further examples appear below.

There is no need for an extra argument in the case of static member functions, constructors and data members.

The next section will illustrate the various available possibilities by showing the code generated, in each case, by the Legacy++ tool.

31.13 WRAPPING C++ CLASSES: LEGACY++

Legacy++ is a tool, not a part of the language specification. Its practical role is, however, sufficiently important to justify a special section in this chapter. This will also provide us with a set of examples covering all the special C++ encapsulation possibilities.

The role of Legacy++

Often you will want to provide an Eiffel encapsulation of **all** the facilities — member functions, static functions, constructors, destructors, data members — of a C++ class. This means producing an Eiffel class that will provide an Eiffel feature for each one of these C++ facilities, using external declarations based on the mechanisms listed in the preceding section.

Rather than writing these external declarations and the class structure manually, you can use Legacy++ to produce the Eiffel class automatically from the C++ class.

Calling Legacy++

Legacy++ is called with an argument denoting a **.h** file that must contain C++ code: one or more classes and structure declarations. It will translate these declarations into Eiffel wrapper classes.

The following options are available:

- **-E**: apply the C preprocessor to the file, so that it will process **#include**, **#define**, **#ifdef** and other preprocessor directives. This is the default.
- **-NE**: do not apply the C preprocessor to the file.
- **-p *directories***: use *directories* as include path.
- **-c *compiler***: use *compiler* as the C++ compiler.
- **-g**: treat the C++ code as being intended for the GNU C++ compiler.

Result of applying Legacy++

Running Legacy++ on a C++ file will produce the corresponding Eiffel classes. Legacy++ processes not only C++ classes but also C++ “structs”; in both cases it will generate an Eiffel class. Among its properties:

- Legacy++ knows about *default specifiers*: **public** for classes, **private** for structs.
- Legacy++; will generate Eiffel features for *member functions* (static or not).

- It will also handle any *constructors* and *destructors* given in the C++ code, yielding the corresponding Eiffel creation procedures. If there is no constructor, it will produce a creation procedure with no arguments and an empty body.
- For any non-static member function or destructor, Legacy++ will generate a *secret feature* with an extra argument representing the object, as explained in the preceding section. It will also produce a public feature with the same number of arguments as the C++ function, relying on a call to the secret feature, as illustrated for *add* and *cpp_add* above.
- The *char* * type is translated into *STRING*. Pointer types, as well as reference types corresponding to classes and types that Legacy++ has processed, will be translated into *POINTER*. Other types will yield the type *UNRESOLVED_TYPE*.

Legacy++ limitations

It is up to you to supply Eiffel equivalents of all the needed types. If Legacy++ encounters the name of a C++ class or type that it does not know — it is neither a predefined type nor a previously translated class — it will use the Eiffel type name *UNRESOLVED_TYPE*. If you do not change that type in the generated class, the Eiffel compiler will report an error.

Legacy++ does not handle inline function declarations and makes no effort to understand the C++ inheritance structure. More generally, given the differences in the semantic models of C++ and Eiffel, Legacy++ can only perform the basic Eiffel wrapping of a C++ class, rather than a full translation. You should always inspect the result and be prepared to adapt it manually. Legacy++'s contribution is to take care of the bulk of the work, in particular the tedious and repetitive parts. The final details are left to the Eiffel software developer.

Legacy++ example

Consider the following C++ class, which has an example of every kind of facility that one may wish to access from the Eiffel side:



```
class IntArray
{
public:
    IntArray (int size);
    ~IntArray ();
    void output ();
    void add (int new_int);
    static char * type ();
protected:
    int *_integers;
};
```

*Warning: this is C++,
not Eiffel.*

Here is the result of applying Legacy++ to that class, which will serve as an illustration of both the C++ interface mechanisms and Legacy++:



```
note
  description:
    "Eiffel encapsulation of C++ class IntArray"
class
  INTARRAY
inherit
  MEMORY
redefine
  dispose
end

create
  make
feature -- Initialization
  make (size: INTEGER)
  -- Create Eiffel and C++ objects.
  do
    object_ptr := cpp_new (size)
  end

feature -- Removal
  dispose
  -- Delete C++ object.
  do
    cpp_delete (object_ptr)
  end
```

```

feature
  output
  -- Call C++ counterpart.
  do
    cpp_output (object_ptr)
  end

  add (new_int: INTEGER)
  -- Call C++ counterpart.
  do
    cpp_add (object_ptr, new_int)
  end

feature {INTARRAY}
  underscore_integers: POINTER
  -- Value of corresponding C++ data member.
  do
    Result := underscore_integers (object_ptr)
  end

feature {NONE} -- Externals
  cpp_new (size: INTEGER): POINTER is
  -- Call single constructor of C++ class.
  external"[
    C++ new IntArray
    signature (EIF_INTEGER_32) use INTARRAY.h
  ]"
  end

  cpp_delete (cpp_obj: POINTER)
  -- Call C++ destructor on C++ object.
  external"[
    C++ delete IntArray
    signature () use INTARRAY.h
  "]"
  end

  cpp_output (cpp_obj: POINTER)
  -- Call C++ member function.
  external "[
    C++ function IntArray
    signature () use INTARRAY.h
  ]"
  alias
    "output"
  end

```

```

cpp_add (cpp_obj: POINTER; new_int: INTEGER)
  -- Call C++ member function.
  external "[
    C++ function IntArray
    signature (EIF_INTEGER_32) use INTARRAY.h
  ]"
  alias
    "add"
  end

cpp_underscore_integers (cpp_obj: POINTER): POINTER
  -- Value of C++ data member
  external "[
    C++ data IntArray
    use INTARRAY.h
  ]"
  alias
    "integers"
  end

feature {NONE} -- Implementation
  object_ptr: POINTER
  -- Access to C++ object
end

```

31.14 USING DYNAMIC LINKED LIBRARIES (DLLS)

Dynamic Link Libraries enable an Eiffel system to take advantage of DLL routines on platforms (such as Windows) supporting the DLL mechanism. A DLL routine is not compiled into your system but kept separate; your system will load the routine the first time it needs to call it. This has two principal advantages:

- You pay only, in memory usage, for what you use. Without DLLs every system must be compiled with every piece of functionality it *might* use even if 98% of executions don't need it. This is a source of size bloat.
- DLLs facilitate software evolution since you can deliver incremental functionality updates through specific DLL replacements, without changing the entire system previously delivered to your users.

Each of these advantages also implies less pleasant counterparts (leading to the phrase “*DLL hell*”): unlike with statically linked systems, a missing component may not be detected until run time (and in certain executions only); a product may install a new DLL that invalidates another product; and you never quite know what your users’ configuration is, which doesn’t facilitate customer support. DLLs are, however, a very popular technique. ISE Eiffel includes a DLL tool for generating DLLs from Eiffel systems.

Eiffel systems also need to *use* DLLs produced elsewhere. Two mechanisms are available for that purpose:

- A DLL sublanguage, similar in spirit to the C and C++ sublanguages reviewed previously, lets you specify DLL routines that you need. Although based on dynamic linking this is a “static” mechanism in that you have to express what you need in your software, before compiling.
- There is also a completely dynamic mechanism, DESC, allowing you to wait until run time to determine what dynamic libraries you need and what routines you want to call.

We now review these two mechanisms in turn.

The static DLL sublanguage

Using the DLL sublanguage you can define an external Eiffel routine relying on a routine from a DLL. You will use a clause **external dll** *file_name* to specify the *file_name* for the dynamic library, and a clause **alias** *name* to specify the name or integer index of the desired routine in that library.

Here is an Eiffel routine encapsulating a function from a DLL:



```

dynamic_external (a, b, c: INTEGER)
  external "[
    "dll
      signature (WORD, DWORD, WORD)
      use herlib.dll
    ]"
  alias
    "35"
  end

```

SEMANTICS

A **dll** subclause requires you to specify a **DLL index or name**, indicating where to find the routine in the DLL. Use the **alias** part for that purpose. Normally, as we have seen, the **alias** part of an **External** declaration gives the native name of the routine (required only if different from the Eiffel name). In the case of a DLL it is also acceptable to provide the routine's index in the library, an integer, such as **35** in the example. There is no ambiguity: an integer alias denotes an index, anything else is taken as a name. This variant also requires the presence of an **External_signature** part.

*The **alias** part also gives the C text of an **inline** routine..*

If your system uses several routines from the same DLL, its execution will only load one instance of the DLL. When the execution terminates, the Eiffel run-time system will free all DLL instances loaded in this way.

Here is the syntax for the DLL variant of the **external** part:

SYNTAX

DLL externals

```

DLL_external  $\triangleq$  ' " ' dll
                [ windows ]
                DLL_identifier
                [DLL_index]
                [External_signature]
                [External_file_use]
                ' " '

DLL_identifier  $\triangleq$  Simple_string
DLL_index  $\triangleq$  Integer

```

Through a **DLL_external** you may define an Eiffel routine whose execution calls an external mechanism from a Dynamic Link Library, not loaded until first use.

The mechanism assumes a dynamic loading facility, such as exist on modern platforms; it is specified to work with any such platform.



External DLL rule

VZDL

A **DLL_external** of **DLL_identifier** *i* is valid if and only if it satisfies the following conditions:

- 1 • When interpreted as a file name according to the conventions of the underlying platform, *i* denotes a file.
- 2 • The file is accessible for reading.
- 3 • The file's content denotes a dynamically loadable module.

External DLL semantics

The routine to be executed (after loading if necessary) in a call to a `DLL_external` is the dynamically loadable routine from the file specified by the `DLL_identifier` and, within that file, by its name and the `DLL_index` if present.



The DLL mechanism specified here is **static** since it requires you to indicate, in the software text, the name of the library and the index (in the form of an integer constant) of the desired routine in that library. One of the advantages of DLLs is the ability to wait until run time to specify both the library and the routine. A corresponding **dynamic** mechanism, complementing the facilities just described, is also available through the DESC library studied later in this chapter.

→ *“DESC: CALLING A DLL ROUTINE DETERMINED AT RUN TIME”, 31.15, page 858.*

The optional **windows** qualifier specifies that the DLL uses the calling conventions of the Windows platform.

31.15 DESC: CALLING A DLL ROUTINE DETERMINED AT RUN TIME

All the mechanisms discussed so far for calling an external routine require that you include the routine’s exact name in the Eiffel text (as the Eiffel routine name if it is the same, after **alias** otherwise), or the routine itself in the C **inline** case. Even the C **dll** mechanism requires you to specify the name of the Dynamic Link Library and the index of the desired routine.

The **Dynamic External Shared Call** mechanism (DESC for short) removes this limitation by letting you wait until run time to determine the name of the external routine to be called in a DLL, or even the name of the DLL itself.

DESC is a library, not a language mechanism, but as important in practice as the purely linguistic mechanisms defined in this chapter.

In line with the general spirit of Eiffel, the DESC takes care of low-level aspects of DLL programming, relieving developers from operations which they would have to perform manually if they were using a language such as C: loading library instances; sharing these instances; freeing the instances when they are not needed any more.

DLLs vary with operating systems. The description in this section applies to Windows.

DESC overview

The DESC mechanism enables you to construct objects representing external routines determined at execution time through their name and libraries, and to call these routines with the appropriate arguments.

Two classes, *DLL* and *DLL_ROUTINE*, supported by an auxiliary class *SHARED_LIBRARY_CONSTANTS*, provide the basis of DESC:

- An instance of class *DLL* describes a Dynamically Linked Library. This class is a descendant of the deferred class *SHARED_LIBRARY*, covering the platform-independent notion of shared library.
- An instance of class *DLL_ROUTINE* describes a routine from a DLL. The class has an attribute of type *DLL* describing the library to which the routine belongs. It has a deferred ancestor *SHARED_LIBRARY_ROUTINE* capturing the platform-independent notion of shared library routine.
- *SHARED_LIBRARY_CONSTANTS* introduces a few declarations useful for dealing with shared libraries and routines, in particular some integer constants describing error codes and type codes. It is an ancestor to both of the preceding classes; application classes using DESC can also inherit from it to gain access to its facilities.

The normal sequence of operations to use the DESC mechanism is:

- 1 • Create a library object (an instance of *DLL*), providing the library's name as argument to the creation procedure.
- 2 • Create a routine object (an instance of *DLL_ROUTINE*), providing the library object, the routine's name or index in the library, and the routine's signature — number of arguments, types of arguments, type of result if any — as arguments to the creation procedure.
- 3 • Apply the procedure *call* to the routine object, passing to *call* an array that contains the actual arguments required by the external routine.

You may repeat each of these steps as often as necessary to use multiple libraries, multiple routines in a library, or multiple calls to a given routine. More details follow.

Creating a library object

To create a DESC object representing a library and load that library, use a declaration such as



```
your_dll: DLL
```

replacing *your_dll* by whatever name you have chosen to denote the library in your software; execute a creation instruction of the form

```
create your_dll.make ("your_lib_name")
```

where *your_lib_name* is the name of the file containing the library.

After this call has been executed, the boolean value *your_dll.meaningful* will be true if and only if the creation has been successful, that is to say, the given name did correspond to an available library, and it was possible to load it.

If *your_dll.meaningful* is false, you can have more details about the error by comparing the value of *your_dll.error_code*, an integer, to those of constant attributes defined in class *SHARED_LIBRARY_CONSTANTS*. As expressed by an invariant of class *DLL*, the value of *meaningful* is true if and only if *error_code* = 0.

Creating a routine object

To create a DESC object representing a routine from a DLL, use a declaration such as



```
your_routine: DLL_ROUTINE
```

replacing *your_routine* by the name you have chosen to denote the routine in your software, and execute a creation instruction of the form



```
create your_routine.make_by_name  
    (your_dll,  
     "your_routine_name",  
     [argtyp1, argtyp2, ...],  
     res_type)
```

or, if you prefer for faster access to identify the routine by an integer index rather than a name:



```
create your_routine.make_by_index
      (your_dll,
       your_routine_index, -- The only differing argument
       [argtyp1, argtyp2, ...],
       res_type)
```

In either form *your_dll* is the library object obtained at the previous step. The preconditions for both *make_by_name* and *make_by_index* include the following clauses on the first argument, known through its formal name *lib* (corresponding to *your_dll* above) in the routine:

```
require
  library_exists: lib /= Void
  meaningful: lib.meaningful
```

After either call, the boolean value *your_routine.meaningful* will be true if and only if the creation has been successful, that is to say, the given name or index did correspond to a routine of the library, and it was possible to open it. If the value is false, you can have more details about the error by comparing the value of *your_routine.error_code*, an integer, to those of constant attributes defined in class *SHARED_LIBRARY_CONSTANTS*. As expressed by a clause of the invariant of class *DLL_ROUTINE*, the value of *meaningful* is true only if *error_code* = 0.

Procedures *make_by_name* and *make_by_index* are usable not only as creation procedures but also as normal exported routines, so that you can later reinitialize the object to represent another external routine. The four arguments play the following roles:

- The first argument, as noted, denotes the library.
- The second argument identifies the desired routine in the library: by its name, of type *STRING*, with *make_by_name*; by its index, of type *INTEGER*, with *make_by_index*.
- The third argument, of type *ARRAY [INTEGER]*, gives the list of type codes for the arguments to the routine. Each type code is an integer associated with one of the possible types to be passed to a DLL routine. Possible type codes appear next.
- The fourth and last argument is a type code for the result.

In the above examples the third argument is declared as a manifest array through the notation [*a1*, *a2*, ...]; here the array items *argtyp1*, *argtyp2*, ... must all be integers giving the type codes of the successive arguments to the routine, taken from the list appearing next. (Use an empty manifest array, [], if the routine has no arguments.)

Type codes

For the type codes used in the array serving as third argument to *make_by_name* and *make_by_index*, and in the fourth argument *res_type*, the class *SHARED_LIBRARY_CONSTANTS* provides a set of constant integer attributes; the easiest way to let a class use them is to make it an heir of that library class. Here is the list of codes:

Type code	Meaning and comments
<i>T_array</i>	Array. What is passed to C is the “special object” containing the actual array elements, directly usable by C. To pass the Eiffel array object, use <i>T_reference</i> . A restriction: the elements of the array may be references, or they may be of a basic type — <i>BOOLEAN</i> , <i>INTEGER</i> etc. — but they may not be of an expanded type other than the basic types.
<i>T_boolean</i>	Boolean value. Passed to C as unsigned character: 0 for false, nonzero for true.
<i>T_character.</i>	Character value.
<i>T_integer</i>	Long integer.
<i>T_no_type</i>	No type. Useful for <i>res_type</i> in the case of a procedure (which has no result type).
<i>T_real</i>	Real number.
<i>T_pointer</i>	Pointer to C structure.
<i>T_reference</i>	Reference to Eiffel object.
<i>T_short_intege r</i>	Short integer. The Eiffel side will use normal <i>INTEGER</i> values for the corresponding actual arguments.
<i>T_string</i>	String. What is passed to C is the C form of the Eiffel string, obtained through the feature <i>to_c</i> of class <i>STRING</i> . To pass the Eiffel string object, use <i>T_reference</i> .

Calling a routine

Having created the object representing the external routine and attached it to entity *your_routine*, you may now call the routine with arbitrary actual arguments through the procedure *call*, a feature of class *DLL_ROUTINE*.

The procedure takes a single argument, of type *ARRAY [ANY]*, containing the successive actual arguments to be passed to the external routine. The easiest technique is to use a manifest array, as in



```
your_routine.call ([-325, 67.2, x, a + b])
```

Accessing the result of a function

If *your_routine* denotes a function (a routine that returns a result), you will be able to access the result by querying the attached instance of *DLL_ROUTINE* through one of the following calls, each corresponding to one of the possible result types:

Typical call	Eiffel type of the result
<i>your_routine.boolean_result</i>	<i>BOOLEAN</i>
<i>your_routine.character_result</i>	<i>CHARACTER</i>
<i>your_routine.integer_result</i>	<i>INTEGER</i>
<i>your_routine.integer_result</i>	<i>INTEGER</i>
<i>your_routine.real_result</i>	<i>REAL</i>
<i>your_routine.reference_result</i> (To use the result, an assignment attempt will usually be necessary.)	<i>ANY</i>
<i>your_routine.string_result</i> (Result converted to Eiffel string format through the feature <i>from_c</i> of class <i>STRING</i> .)	<i>STRING</i>

Consistency requirements and protection against errors

In a call to procedure *call* such as the above, the number of elements in the array and their types must correspond to the signature — number and type of arguments — specified in the third argument of the latest call to *make_by_name* or *make_by_index*.

This requirement is captured by a function *conforms_to_signature*, relying on the function *conforms_to* from the Kernel Library class *ANY*. The third precondition clause of procedure *call* states it:

```
call (args: ARRAY [ANY])
  require
    meaningful: meaningful
    valid_array: args /= Void
    conformant: conforms_to_signature (args)
```

This precondition, combined with queries *meaningful* and *error_code* in classes *DLL* and *DLL_ROUTINE*, provides a certain degree of protection against possible errors. But the Eiffel side does not know anything about the external routine, and so cannot check that the number of actual arguments and their types match the actual signature of that routine. You are responsible for ensuring that the routine gets what it expects.

Similarly, each of the *_result* features has a precondition stating that it must be compatible with the result type set by the latest call to *make_by_name* or *make_by_index*. For example in the case of *boolean_result* the result type must have been set to *T_boolean*. Here too there is no protection against type errors at the Eiffel-C border; double-check your software to make sure that the result types you are positing on the Eiffel side match what the DLL routines actually declare.

Sharing and freeing

One of the effects of creating a library object through a creation instruction of the form *create your_dll.make* ("*your_lib_name*") is, as noted, to load the library of name *your_lib_name*. When you subsequently create routine objects relative to *your_dll*, they will all share the same library instance.

You may, if you wish, load several instances of a given library: simply create several library objects, passing in every case the same string "*your_lib_name*" as actual argument to the *make* creation procedure.

If the same library name is used by an external DLL routine, statically declared through the mechanism studied [earlier in this chapter](#), and by a library object created dynamically by the DESC mechanism as an instance of *DLL*, two different instances will be loaded.

When a DESC library object is no longer accessible and the garbage collector reclaims it, this will automatically (through the procedure *dispose* of class *MEMORY* as redefined for class *DLL*) free the corresponding library instance.

For most uses this automatic freeing will be sufficient. If, however, you want to free a library manually, you can do so through the call `your_dll.free`. As a postcondition of this call, `your_dll.meaningful` will be false, as well as `your_dll.meaningful` for any routine object `your_routine` that was created relative to `your_dll`.

31.16 THE CECIL LIBRARY

The mechanisms studied so far support **call-out**: calling foreign mechanisms from Eiffel. There is a complementary need for a **call-in** mechanism, enabling foreign software to call Eiffel features.

Cecil overview

Call-in and call-out are in fact closely related since an external (call-out) routine may pass, among others, arguments of the Address form, denoting features of the enclosing class. The sole purpose of such arguments is, obviously, to let foreign routines call the associated Eiffel features. ← “PASSING THE ADDRESS OF AN EIFFEL FEATURE”, 31.8, page 833.

More generally, some developers may wish to write foreign routines that create Eiffel objects and apply features to these objects, without necessarily relying on features explicitly passed by the Eiffel side. This last section shows a way to do this from C, using a library of C functions called the C-Eiffel Call-In Library, or **Cecil**. The first C in the acronym is there mostly for historical reasons: you can use Cecil from any foreign language that supports standard argument passing conventions.

Cecil role and status



Most developments do not need to use Cecil or its equivalent, and most developers do not need to learn about it. The ideas are of interest to installations with a heavy use of C or some other foreign language, if they want to integrate Eiffel classes in applications driven by their foreign components. If you are not in this situation, then you most likely should spare yourself the rest of this chapter; but do shed a tear or two for your less fortunate colleagues.

Please send your tax-deductible contributions to the HAVOC fund (Help All Victims Of C!), Box 00, Palma de Majorca.

Call-in mechanisms belong in foreign languages. The Cecil library this section describes, then, is not part of Eiffel as a language, but it is a required component of any Eiffel implementation.

The following Cecil resources should complement the explanations of this section:

- <http://eiffel.com/doc/manuals/library/cecil/> is a complete Cecil manual.

- If you program non-trivial Cecil applications you will benefit from the set of examples at <ftp://ftp.eiffel.com/pub/examples/cecil>; you can retrieve individual examples from that directory, or download all examples, zipped, from <ftp://ftp.eiffel.com/pub/examples/cecil/cecil.zip>. The directory is split into two subdirectories: *unix-examples* and *windows-examples*.

Compiling for Cecil

To use the facilities of an Eiffel system through Cecil you must first compile a “cecilized” form of it. This may require a special compilation or (as with ISE Eiffel) you may simply get the “cecilized” form as a standard output of your compilation with no extra work.

You will of course need to compile your foreign application, a process that is not always as automatic as Eiffel compilation as managed by good Eiffel environments. Even here, however, Eiffel can help: you can specify a Make file in the **external** part of your Ace through a directive of the form

```
external: make: "your_makefile"
```

which causes Eiffel compilation to start C compilation using the provided Make file. (To specify its location, remember that you can use environment variables, such as `$EIFFEL5` denoting the location of the Eiffel installation, in the Ace file.)

As explained next, the foreign software will gain access to the Cecil mechanisms through two include files produced by the Eiffel environment: `eif_cecil.h` and (if execution starts on the foreign side rather than from Eiffel) `eif_setup.h`. You will use the “include” option of your C compiler, normally `-I`, to specify the directory where these files reside.

→ For the location of this directory in ISE Eiffel see [“ISE Eiffel specifics”, page 876](#).

Avoiding abusive optimization

Even with a compiler that generates cecilized code without any special compilation option, you may have to exert some care if the compiler (again such as ISE Eiffel) performs dead-code-removal optimization, to delete the generated code for routines that are not called from within the Eiffel system. Such routines may still be needed by foreign software as part of the cecilized interface. To protect them from over-enthusiastic dead code removal, list them in the **visible** clause of the Ace file, as in



```
system system_name root ... default ... cluster
...
your_cluster: "/home/user/cluster1"
adapt
...
visible
    CLASS1
    CLASS2
        create
            "other_make"
        export
            "feat1", "feat2"
        end
    end
... Other cluster specifications...
end
```

→ See appendix [B](#) about Luce, in particular "[VISIBLE FEATURES](#)", [B.13, page 1034](#)

Here all exported features of *CLASS1* are available to the external software; for *CLASS2*, only *other_make* (for creation) and *feat1* and *feat2* (for normal call) are available.

By default the status of features is deduced from the Eiffel class text: only the publicly available features will be available through the Cecil interface. You can use the **export** clause to override this default, in particular to make a feature is available to the outside world even though it is not used in the Eiffel system and hence subject to dead-code removal.

The creation status is determined in a similar way: by default any procedure listed in Eiffel as a generally available for creation will be accessible through Cecil; you can override this default through the **create** subclass of the **visible** clause.

Note that because a Cecil application will create and initialize an object through two separate calls (unlike the Eiffel instruction *a.make (...)* which does both), the creation and export status are the same for Cecil, so listing a feature under **create** or **export** has the same effect: making it available to foreign software through the Cecil interface.

Basic Cecil conventions

The Cecil library contains macros, functions, types and error codes. All have names beginning with either **eif_** (functions and macros) or **EIF_** (types and error codes); examples are the function **eif_type_id** and the type **EIF_PROCEDURE**, explained below. Their declarations appear in a C “header file”, *eif_cecil.h*, which you may add to a C program through the C preprocessor directive

*Eiffel's emphasis on clarity suggests using **eif_** and **EIF_** as prefixes, but some of the resulting names would be too long for some C compilers.*

```
#include "eif_cecil.h"
```

Warning: this is C, not Eiffel.

A similar mechanism will be available for other supported foreign languages, although the rest of the discussion will assume C or C++.

We now review the various facilities available from *cecil.h*. To avoid any confusion with the format used in the rest of this book for Eiffel software elements, C code will appear as follows (in color):

- Bold font (as elsewhere for Eiffel keywords) for Cecil functions, macros and types, such as **eif_type_id** and **EIF_PROCEDURE**.
- Italic font, for C names representing Eiffel class names or entities, such as *CLASS_NAME*.
- Regular font for ordinary C text, including example variables illustrating function usage, such as *your_id*.

The basic scheme of using Cecil is the following:

- Build an Eiffel system.
- “Cecilize” it: compile it for Cecil use. This may require some specific compilation options, or at least, as noted above, protecting features from dead code removal.
- Write a program in C or some other language that gains access to the resulting facilities through appropriate **include** directives and uses Cecil functions and macros to create Eiffel objects, call features on them, and receive any resulting exceptions.

Initializing the Eiffel 4 run-time

An application using Cecil, involving both Eiffel and foreign elements, may start its execution from either side. If execution starts on the non-Eiffel side — in other words, if the foreign language is in control — it will need, prior to calling any Eiffel facility, to set up the Eiffel run time to ensure that Eiffel mechanisms such as garbage collection and signal handling will work properly. It will also need, before it terminates, to call the run-time termination mechanisms, ensuring in particular that all Eiffel objects are freed and the corresponding *dispose* procedures are called to free any associated system resources.

The runtime setup will typically appear in the foreign application’s main program. Simply add the preprocessor directive

```
#include "eif_setup.h"
```

Warning: this is C, not Eiffel.

To start the Eiffel runtime, use

```
EIF_INITIALIZE (failure_function);
```

Warning: this is C, not Eiffel.

where *failure_function* () is a function to be called in case of failure to initialize. To terminate the Eiffel runtime, collect all objects and call their *dispose* procedures if any, use

```
EIF_DISPOSE_ALL;
```

Warning: this is C, not Eiffel.

`EIF_INITIALIZE` and `EIF_DISPOSE_ALL` are macros defined in `eif_setup.h`. The macros assume that the enclosing function, normally the main program, has the three standard arguments, as in

```
main (int argc, char **argv, char **envp);
```

Warning: this is C, not Eiffel.

Manipulating values of basic Eiffel types

If you pass Eiffel values of basic types (integers, booleans and so on) you will need to make sure that the C side manipulates them properly. For example there is no guarantee that an Eiffel *INTEGER* and a C *int* are the same; for portability and to guarantee numerical precisions the Eiffel-C interface includes the following set of macros defining the C representation of the Eiffel basic types:

```
EIF_BOOLEAN   EIF_CHARACTER   EIF_INTEGER_8
EIF_INTEGER_16 EIF_INTEGER_32 EIF_INTEGER_64
EIF_REAL_32   EIF_REAL        EIF_POINTER
```

These names are those of macros defined in `cecil.h`.

The macro **EIFFEL_TYPE** denotes the C type (actually *int*) covering C representations of Eiffel types; the possible values are the twelve listed, plus **EIF_REFERENCE**, introduced below.

If you have control over the C code, always use the above types to manipulate Eiffel values from C. So with an Eiffel external function



```
c_func (ptr: POINTER; obj: OBJECT): INTEGER is
  external
  "C include %"your_file.h%"
end
```

you may write the C side as

```
EIF_INTEGER_32 c_func(EIF_POINTER ptr, EIF_OBJECT obj)
{... Function body ...}
```

Warning: this is C, not Eiffel.

In other cases, the C function pre-exists and you cannot (or do not want to) change it. In that case you should take care of the proper typing on the Eiffel side, using the **External_signature** facility introduced earlier in this chapter With a function

← See "[Controlling the Eiffel-C type correspondence](#)", page 846.



```
int other_func (void *arg1, char c, FILE *file)
{... Function body ...}
```

Warning: this is C, not Eiffel.

you should write the Eiffel external as

```
other_func (arg1: POINTER; c: CHARACTER; file: POINTER):
  INTEGER
  external
  "C(void *, char, FILE *) : int include %"your_file.h%"
end
```

Omitting the `External_signature` part (the part that lists the C types before the colons) would produce C compilation warnings and possibly errors.

Manipulating Eiffel class types

To call Eiffel features, the foreign software will need to access the classes and types to which they belong. It will know an Eiffel type through a “type-id”, of type `EIF_TYPE_ID`.

To obtain a type-id for a type `TYPENAME` and record it in a C variable `your_id`, use the function `EIF_TYPE_ID`, returning an `EIF_TYPE_ID`:



```
EIF_TYPE_ID your_id;
...
your_id = EIF_TYPE_ID ("TYPENAME");
```

Warning: this is C, not Eiffel.

As usual, you must make sure that the base class of `TYPENAME` is not optimized away by the compiler.

← *“Avoiding abusive optimization”, page 866.*

If the class is generic, include the generic parameters in the `TYPENAME` as in:



```
your_other_id = EIF_TYPE_ID ("ARRAY [INTEGER]");
```

Warning: this is C, not Eiffel.

Given an Eiffel type descriptor `type_id` of `EIF_TYPE_ID`, you can obtain the corresponding Eiffel type name as well as the name of the generating class (the type’s base class). Use `EIF_TYPE` (`tid`) for the type name and `EIF_CLASS` (`tid`) for the class name. In both cases the result is a `char *`, representing a C string.

Accessing an Eiffel object

A foreign function may access Eiffel objects through references passed to it by the Eiffel side in external calls, or returned by calls to `EIF_CREATE` (see below). The corresponding variable must be declared of the Cecil type `EIF_OBJECT`.

A value `your_object` of type `EIF_OBJECT` is not a C pointer to the corresponding object. To obtain such a pointer (for example to pass it to a C function which manipulates objects directly), use the macro `EIF_ACCESS`, which takes an `EIF_OBJECT` and returns a pointer to the object:

*Current C guidelines suggests that `EIF_ACCESS` should return a void *. Warning: this is C, not Eiffel. The result is a null pointer if `your_object` represents a void reference.*

```
some_function (EIF_ACCESS (your_object), ...):
```



The reason for this rule is that an Eiffel implementation supporting garbage collection may move objects around. Then a pointer passed directly to a C function might be obsolete by the time the function tries to access the associated object. Given an **EIF_OBJECT**, **eif_access** will retrieve a correct pointer. If the implementation does not move objects, **eif_access** will do little or no work.

The result type of **eif_access** is of type **EIF_REFERENCE**. A value of this type is a pointer to an Eiffel object; you can pass it to an Eiffel routine, or as the result of a C external. Do not, however, pass an **EIF_REFERENCE** to another C function, since the object might have moved; use **EIF_OBJECT** instead.



What if **your_object** is a variable that does not just allow immediate object processing as above, but retains its value between successive activations of the C side? In the meantime, the Eiffel side might have discarded all references to the corresponding object; but then a garbage collecting implementation must not be allowed to reclaim it! To avoid this, the C side must **adopt** the object, using the function **eif_adopt**. Once C functions do not need to hold the object any more, they may release it through **eif_wean**. Here is the scheme:

```
EIF_OBJECT your_object,...
eif_adopt (your_object);
    ... Then in the same or another C program unit: ...
    some_function (eif_access (your_object), ...);
    ...
eif_wean (your_object);
```

Warning: this is C, not Eiffel.

A call to `EIF_WEAN` actually returns a value: an `EIF_REFERENCE` to the object just “weaned”.

You should use `EIF_ADOPT` for a value of type `EIF_OBJECT`, created by an Eiffel routine and passed as argument to the foreign software. For an `EIF_REFERENCE` value returned by one of the Cecil mechanisms, use `EIF_PROTECT` instead. An example appears next with an `EIF_REFERENCE` denoting an Eiffel string created by `EIF_STRING` (“*SOME TEXT*”). Function `EIF_PROTECT` returns an `EIF_OBJECT`; as with `EIF_ADOPT`, you should `EIF_WEAN` that `EIF_OBJECT` when you do not need it any more.

Creating an Eiffel object

To create an object from outside, use the function `EIF_CREATE`, which takes an `EIF_TYPE_ID` argument and returns an `EIF_OBJECT`. For example:



```
EIF_OBJECT your_array;
...
your_array = EIF_CREATE (EIF_TYPE_ID ("ARRAY [INTEGER]"));
```

Warning: this is C, not Eiffel.

Assuming class `LINKED_LIST` with one generic parameter, this creates a direct instance of `LINKED_LIST [INTEGER]`. Function `EIF_CREATE` calls `EIF_ADOPT`; the C side should call `EIF_WEAN` when and if it does not need the object any more.



As the example shows, `EIF_CREATE` **does not call a creation procedure**. To apply a creation procedure, you will need to include a separate call, using function `EIF_PROCEDURE` as explained below. This departs from Eiffel conventions, which prohibit creating an object without applying a creation procedure if the class has a `Creators` clause. With Cecil, forgetting to call a creation procedure after `EIF_CREATE` may produce an object which violates the class invariant, so you must be particularly vigilant to avoid this error (which cannot occur in Eiffel).

← About creation rules in Eiffel and the `Creators` clause, see chapter 20.

A shortcut is available for the case of string objects. As you will recall, `STRING` is a normal class with its own creation procedures. To avoid going through the creation of a `STRING` object and separate initialization, you can use `EIF_STRING` as in:



```
EIF_REFERENCE your_string;
EIF_OBJECT your_string_object;
my_string = EIF_STRING ("SOME TEXT");
your_string_object = EIF_PROTECT ("my_string");
```

The result of `eif_string` is an **EIF_REFERENCE**; if you are going to use it beyond the immediate context, make sure to **eif_protect** it as shown. When you do not need it any more, call **eif_wean** (`your_string_object`) to let the Eiffel garbage collector reclaim it once the Eiffel side is also done with it.

As a related facility, you can produce an Eiffel array `eif_array` from a C array `c_array` through the macro call



```
eif_array_from_c (eif_array, c_array, n, type_id)
```

where `n`, an integer, is the number of array elements and `type_id`, an integer, represents is the type of the array elements. The argument `eif_array` must be an **EIF_REFERENCE** denoting an array; `c_array` must be of type (`type_id *`), with enough space available to hold the array values. The value of `type_id` must be one of the Eiffel-C interface types defined earlier: **EIF_BOOLEAN** etc. for basic types, **EIF_REFERENCE** for any reference type.

← “*Manipulating values of basic Eiffel types*”, page 870; “*Manipulating Eiffel class types*”, page 871.

You can similarly use `eif_string_from_c` (`eif_string, c_string, n`) to get the C string (`char *`) equivalent of an Eiffel string.

Calling routines

Having gained access to Eiffel objects, the foreign application will want to apply Eiffel routines and attributes to them. To do so it needs pointers to these routines, which it will obtain through one of a set of Cecil functions provided for this purpose. For example, having obtained the type-id `your_array` as shown above, use the following to assign to variable `your_procname` a pointer to the Eiffel procedure whose Eiffel name in class `ARRAY` is `put`:



```
EIF_PROCEDURE your_array_put:  
...  
your_array_put = eif_procedure ("put", your_array);
```

Warning: this is C, not Eiffel.

Function `eif_procedure` is one of a group of functions, each corresponding to a different category of Eiffel routines: procedures, functions returning results of basic types, class types, bit types. Here is the list of these functions, with their argument and result types:

All these routines have the same arguments: a string (`char * in C`), representing a routine name, and a type-id, obtained through `eif_type_id`.

These functions look for a routine of name `rout_name` in the base class of the type corresponding to `type_id`. If such a routine exists, the result will be a pointer to a C function representing it desired routine; you may then call that function on appropriate arguments. For example:

→ See “*Requesting a non-existing routine*”, page 875 below about Warning: this is C, not Eiffel.


```

EIF_PROCEDURE eif_procedure
    (char * rout_name, EIF_TYPE_ID type_id)
EIF_REFERENCE_FUNCTION eif_reference_function
    (char * rout_name, EIF_TYPE_ID type_id)
EIF_INTEGER_32_FUNCTION eif_integer_32_function
    (char * rout_name, EIF_TYPE_ID type_id)
EIF_CHARACTER_FUNCTION eif_character_function
    (char * rout_name, EIF_TYPE_ID type_id)
EIF_REAL_32_FUNCTION eif_real_function
    (char * rout_name, EIF_TYPE_ID type_id)
EIF_REAL_FUNCTION eif_real_function
    (char * rout_name, EIF_TYPE_ID type_id)
EIF_BIT_FUNCTION eif_bit_function
    (char * rout_name, EIF_TYPE_ID type_id)
EIF_BOOLEAN_FUNCTION eif_boolean_function
    (char * rout_name, EIF_TYPE_ID type_id)
EIF_POINTER_FUNCTION eif_pointer_function
    (char * rout_name, EIF_TYPE_ID type_id)

```

```

(your_array_put) (eif_access (your_array), 365, 10)

```

This applies the routine corresponding to *go*, accessible through `your_array_put` as a result of the above call to `eif_procedure`, to the object corresponding to `your_array`, with the actual argument `10`. The corresponding call would have been written in Eiffel as `your_array.put (345, 10)`. In C, do not forget to enclose the name of the function pointer, here `your_array_put`, in parentheses, and to use `eif_access`.

REMARKS

As in Eiffel, the call will use dynamic binding: it will trigger the version of the feature directly adapted to the type of the target object.

Requesting a non-existing routine

The facilities just reviewed — `eif_procedure`, `eif_reference_function` and so on — enable the foreign side to gain access to an Eiffel feature. What if the requested feature does not exist in the class specified? If you stay within Eiffel this case will not arise since the type checking mechanism will detect the error at compile time; but from a foreign language no such static check is possible; the error will only become manifest at run time.

For the outcome in such a case you have a choice between two behaviors, which you can enforce by calling either of two status-setting procedures (whose effect will last until a call to the other):

Warning: this is C, not Eiffel.

The word `POINTER` in `EIF_POINTER_FUNCTION` refers to the Eiffel `POINTER` type (see [31.8](#) above), not to C pointers.

Variants of `eif_integer_32_function` also exist for 8, 16 and 64.

- You can ensure that a request for a non-existent feature will trigger an exception, passed as a signal to the foreign side. This is not the default behavior, but you can obtain it by calling `EIF_ENABLE_VISIBLE_EXCEPTION`.
- By default, functions such as `EIF_PROCEDURE` and consorts return a null value if they can't find the Eiffel feature. You can restore this default behavior by calling `EIF_DISABLE_VISIBLE_EXCEPTION`.

Accessing field objects

The macro `EIF_ATTRIBUTE` enables the foreign side to access fields of objects, corresponding to attributes of the generating classes.

You may use the result of `EIF_ATTRIBUTE` in two different ways: as an expression, or “r-value” in C terminology; or as a `Variable` entity, or “l-value”, which may then be the target of an assignment. Such an assignment will re-attach the corresponding object field.

The macro requires four arguments:

```
EIF_ATTRIBUTE
(EIF_REFERENCE object, char * attrib_name,
 EIFFEL_TYPE type_id, int const * status);
```

Warning: this is C, not Eiffel.

The `object` argument denotes the object of which you want to access a field.; `attrib_name` denotes the name of the attribute in the generating class.

The third argument, `type_id`, serves to cast the result to the appropriate type. It must be one of the Eiffel-C interface types defined earlier: `EIF_BOOLEAN` etc. for basic types, `EIF_REFERENCE` for any reference type. `EIFFEL_TYPE` covers all these type values. In `EIF_REFERENCE` case, do not forget to `EIF_PROTECT` it the result if you will use it further.

← See *“Manipulating values of basic Eiffel types”, page 870, which also introduced `EIFFEL_TYPE`, and “Manipulating Eiffel class types”, page 871.*

The last argument, `status`, is a result code. Possible values are `*status = EIF_CECIL_OK`, indicating success, `EIF_NO_ATTRIBUTE`, indicating that no field exists in the object for the given name, and `EIF_CECIL_ERROR` for other Cecil errors. If you have selected `EIF_ENABLE_VISIBLE_EXCEPTION` as explained above, the last two cases will trigger an exception.

ISE Eiffel specifics

The following comments apply to the use of Cecil with ISE Eiffel and may not be relevant for other implementations.

To will gain access to the Cecil facilities through two include files, both in `$(EIFFEL5)/bench/spec/$(PLATFORM)/include` where `$(EIFFEL5)` is the Eiffel installation directory and `$(PLATFORM)` the platform code (such as `windows`, `linux` etc.):

- To use Cecil in a C file it suffices to include `EIF_eiffel.h`.
- The main program may include `EIF_setup.h` to access facilities for setting up and terminating the Eiffel run-time. This is not necessary if execution starts on the Eiffel side; if, however, a C main program starts execution and needs at some stage to call Eiffel mechanisms it will need these facilities to get everything initialized on the Eiffel side.

The following Lace options will be useful on Windows:

- Use `console_application (yes)` if you want to produce a console application rather than a default (graphical) Windows application.
- Use `C_main ("path_name")` to specify that the main program will be the C file at `path_name`.

ISE Eiffel offers three compilation modes: melted (super-fast incremental recompilation, no C generation), frozen (incremental, C generation), finalized (full C generation, extensive global optimizations). You can use Cecil with all three modes.

In the case of a melted system of name `system_name`, you must copy the file `<system_name.melted>` from the subdirectory `EIFGEN/W_code` of your project directory to the directory from which you will execute your C program. (The execution directory, not the compilation directory). This file will change after each melting; so on Unix it may be more convenient to use instead a symbolic link to it, which also saves space.

A limitation exists in case of a melted system: it is not permitted to use through Cecil any routine that has been melted in the last compilation. This would raise the run-time exception “`$ applied to melted routine`”. The solution is simple: refreeze.

To “cecilize” your system you do not need to use any special Eiffel compilation option. The only extra concern you need to have is, in finalized mode, to protect features from the dead-code removal algorithm, as explained earlier. Compilation produces both C code and a Makefile, in a subdirectory of `EIFGEN` in your project directory: `EIFGEN/W_code` (in melted or frozen mode) or `EIFGEN/F_code` (in finalized mode). To produce a CECIL library, you must, in a DOS console (Windows) or shell (Unix), go to the appropriate `EIFGEN/x_code` directory and run the make utility with the `cecil` option: `make cecil` (Unix), `nmake cecil` (Windows with Visual C++ and compatible compilers).

This generates a Cecil archive whose name derived from the name *system_name* of your Eiffel system: *system_name.lib* (Windows), *libsystem_name.a* (Unix). The archive will include the Eiffel runtime thanks to the **include** directives listed above. Then it suffices to link the archive with the rest of your application through the link command appropriate for your operating system.

On Unix, you should use the **-lm** option to the link command to include the C mathematical library, required by the Eiffel runtime. You may need other libraries too, for example **-lbsd** on Linux, **-lpthread** (Posix threads) on Linux, **-lthread** (Solaris thread library) on Solaris. The linking command might look like this:



```
ld -lm -lbsd your_application.c libsystem_name.a
```

Lexical components

32.1 OVERVIEW

The previous discussions have covered the syntax, validity and semantics of software systems. At the most basic level, the texts of these systems are made of **lexical** components, playing for Eiffel classes the role that words and punctuation play for the sentences of human language. All construct descriptions relied on lexical components — identifiers, reserved words, special symbols ... — but their structure has not been formally defined yet. It is time now to cover this aspect of the language, affecting its most elementary components.

This chapter defines the various kinds of lexical element.



The lexical structure of Eiffel is simple and predictable. For a first approach to Eiffel, the examples found in the rest of this book should provide enough models to enable you to write your own class texts without studying this chapter.

32.2 CHARACTER SETS

Every lexical component is a sequence of characters.



Syntax (non-production): Character, character set

An Eiffel text is a sequence of **characters**. Characters are either:

- All 32-bit, corresponding to Unicode and to the Eiffel type *CHARACTER_32*.
- All 8-bit, corresponding to 8-bit extended ASCII and to the Eiffel type *CHARACTER_8*.

Compilers and other language processing tools must offer an option to select one **character set** from these two. The same or another option determines whether the type *CHARACTER* is equivalent to *CHARACTER_32* or *CHARACTER_8*.

In manifest strings and character constants, characters can be coded either directly, as a single-key entry, or through a multiple-key character code such as `%N` (denoting new-line) or `%/59/`. The details appear below.

32.3 CHARACTER CATEGORIES

The discussion will rely on a classification of characters into letters, digits and other categories:



Letter, `alpha_betic`, numeric, `alpha_numeric`, printable

A **letter** is any character belonging to one of the following categories:

- 1 • Any of the following fifty-two, each a lower-case or upper-case element of the Roman alphabet:

a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- 2 • If the underlying character set is 8-bit extended ASCII, the characters of codes 192 to 255 in that set.
- 3 • If the underlying character set is Unicode, all characters defined as letters in that set.

An **alpha_betic character** is a letter or an underscore `_`.

A **numeric character** is one of the ten characters *0 1 2 3 4 5 6 7 8 9*.

An **alpha_numeric character** is `alpha_betic` or numeric.

A **printable character** is any of the characters listed as printable in the definition of the character set (Unicode or extended ASCII).



In common English usage, “alphabetic” and “alphanumeric” characters do not include the underscore. The spellings “*alpha_betic*” and “*alpha_numeric*” are a reminder that we accept underscores in both identifiers, as in *your_variable*, and numeric constants, as in *8_961_226*.

“Printable” characters exclude such special characters as new line and backspace.

Case 2 of the definition of “letter” refers to the 8-bit extended ASCII character set. Only the 7-bit ASCII character set is universally defined; the 8-bit extension has variants corresponding to alphabets used in various countries. Codes 192 to 255 generally cover letters equipped with *diacritical marks* (accents, umlauts, cedilla). As a result, if you use an 8-bit letter not in the 7-bit character set, for example to define an identifier with a diacritical mark, it may — without any effect on its Eiffel semantics — display differently depending on the “*locale*” settings of your computer.

To avoid such headaches, switch to Unicode (and discover a few new headaches).

32.4 GENERAL FORMAT

At the lexical level, a class text is made of **tokens**, **breaks** and **comments**. Tokens are the meaningful components; breaks play a purely lexical role (separating tokens); comments add informal explanations for the benefit of human readers.

The next sections examine breaks, comments, layout conventions, the influence of letter case, and the various categories of token.

32.5 BREAKS



Break character, break

A **break character** is one of the following characters:

- Blank (also known as space).
- Tab.
- New Line (also known as Line Feed).
- Return (also known as Carriage Return).

A **break** is a sequence of one or more break characters that is not part of a **Character_constant**, of a **Manifest_string** or of a **Simple_string component** of a **Comment**.

Some platforms do not support the concept of a New Line character, but represent texts as sequences of lines. On such a platform, you may apply the rules of this chapter by considering a text as made of the concatenation of all its lines, with a New Line character between consecutive lines.

Breaks separate successive tokens. Any break is as good as any other:

→ As detailed below:
[“TEXT LAYOUT”](#),
 32.7, page 885.



Break semantics

Breaks serve a purely syntactical role, to separate tokens. The effect of a break is independent of its makeup (its precise use of spaces, tabs and newlines). In particular, the separation of a class text into lines has no effect on its semantics.

Because the above definition of “break” excludes break characters appearing in **Character_constant**, **Manifest_string** and **Comment** components, the semantics of these constructs may take such break characters into account.

32.6 COMMENTS



A class text may contain comments, which have no effect on the semantics of the classes in whose texts they appear, but provide explanations for the benefit of readers of these texts.

Any part of a line beginning with two consecutive hyphens `--` and extending to the end of a line is a comment, as in

```
Some other text -- A comment
```

or, without any preceding text

```
-- A comment
```

Some comments are “expected”, others “free”:



Expected, free comment

A comment is **expected** if it appears in a construct as part of the style guidelines for that construct. Otherwise it is **free**.

Free comments appear in any position where you feel you should include some explanations for the reader of your software:

```
your_array.sort -- Sorting algorithm must be stable.
```

Expected comments are more closely connected to the syntax structure. The three main examples are informal assertions, feature clause qualifiers and feature headers. As an example of the first, you may use a **Comment** (possibly with a **Tag_mark**) as an **Assertion_clause** expressing a property which you have not been able to write formally as a **Boolean_expression**; here is an example from **FIXED_QUEUE** in EiffelBase:

```
invariant
    0 <= first_index
    first_index <= last_index
    -- If queue is not empty, items are in positions
    -- first_index, first_index + 1, ..., last_index - 1 (mod capacity)
    . ... More invariant clauses ...
```

Feature comments introduce successive feature categories, each in a separate **Feature_clause**, as in

```
class LINKED_LIST [T] inherit
    ...
feature -- Access
    ... Feature declarations ...
feature -- Measurement
    ... Feature declarations ...
```

← From the sketch of **LINKED_LIST** on page 134.




```
feature {LINKED_LIST} -- Implementation
    ... Feature declarations ...
    ...
end
```

Finally the optional `Header_comment` of a feature appears after its signature and expresses concisely the purpose of the routine, as in:

DEFINITION

```
convert_to_resolution (res_val: REAL)
    -- Convert to world coordinates,
    -- using res_val as resolution.
    ... Rest of Routine omitted ...
```

If a comment is important, it is often advantageous to replace it by a `note` clause, which has an official place in the syntactic structure. The convention in this case is to use `what` as note tag:

DEFINITION

```
convert_to_resolution (res_val: REAL)
    note
        what: "[
            Convert to world coordinates,
            using res_val as resolution.
        ]"
    ... Rest of Routine omitted ...
```

The of Free and Expected comments is the same: a comment is made of one or more line segments, each beginning with two consecutive dash characters -- and extending to the end of the line

With an auxiliary definition

Syntax (non-production): “Blanks or tabs”, new line

A specimen of `Blanks_or_tabs` is any non-empty sequence of characters, each of which is a blank or a tab.

A specimen of `New_line` is a New Line.

the following syntax captures the form of comments:

SYNTAX

Comments

```
Comment  $\triangleq$  "--" {Simple_string Comment_break ...}*
Comment_break  $\triangleq$  New_line [Blanks_or_tabs] "--"
```

where **Simple_string** denotes sequences of characters without a new line.

This syntax implies that two or more successive comment lines, with nothing other than new lines to separate them, form a single comment.

For example, the text extract

```
c :=          -- This is comment text
              -- This is the first comment's continuation
a + b        -- This is a second comment.

              -- This is a third comment.
```

contains three-comments as indicated (with a blank line between the second and the third)..

Syntax (non-production): Free Comment rule

It is permitted to include a free comment between any two successive components of a specimen of a construct defined by a BNF-E production, except if excluded by specific syntax rules.

An example of construct whose specimens may not include comments is **Line_sequence**, defined not by a BNF-E production but by another “non-production” syntax rule: no comments may appear between the successive lines of such a sequence — or, as a consequence, of a **Verbatim_string**.

Similarly, the Alias Syntax rule excludes any characters — and hence comments — between an **Alias_name** and its enclosing quotes.

The **Header_comment** of a routine is formally equivalent to the more explicit **note** form:



Header comment rule

A feature **Header_comment** is an abbreviation for a **Note** clause of the form

note

what: *Explanation*

where *Explanation* is a **Verbatim_string** with [and] as **Open_bracket** and **Close_bracket** and a **Line_sequence** made up of the successive lines (**Simple_string**) of the comment, each deprived of its first characters up to and including the first two consecutive dash characters, and of the space immediately following them if any.

Per the syntax, a comment is a succession of `Simple_string` components, each prefixed by "--" itself optionally preceded, in the second and subsequent lines if any, by a `Blank_or_tabs`. To make up the `Verbatim_string` we remove the `Blank_or_tabs` and dashes; we also remove one immediately following space, to account for the common practice of separating the dashes from the actual comment text, as in

-- A comment.

32.7 TEXT LAYOUT

An Eiffel text is a sequence; each of the elements of the sequence is a break, a comment or a token.

You may always insert a break between two elements without affecting the semantics of the text.

A break is not required between two adjacent elements if one is a comment and the other a token or another comment. Between two successive tokens, a break may be required or not depending on the nature of the tokens.

We may divide tokens into two categories:



Symbol, word

A **symbol** is either a special symbol of the language, such as the semicolon “;” and the “.” of dot notation, or a standard operator such as “+” and “*”.

A **word** is any token that is not a symbol. Examples of words include identifiers, keywords, free operators and non-symbol operators such as **or else**.

→ For the list of symbols see below “SPECIAL SYMBOLS”, [32.11, page 889](#).

Then:



Syntax (non-production): Break rule

It is permitted to write two adjacent tokens without an intervening break if and only if they satisfy one of the following conditions:

- 1 • One is a word and the other is a symbol.
- 2 • They are both symbols, and their concatenation is not a symbol.

Without this rule, adjacent words not separated by a break — as in *ifxthen* — or adjacent symbols would be ambiguous.

Between adjacent words or adjacent symbols a break is required. For example, a break is needed between a keyword and an identifier (both of which are words); in



```
if x then ...
```

the breaks both before and after *x* are required. But the assignment

```
c:=a+b
```

may be written without any break, although the standard style guidelines → Chapter 34. suggest using a one-blank break both around the assignment symbol `:=` and around every operator.

The syntax actually permits few cases of adjacent symbols; the most common is a prefix operator appearing after an infix operator, as in `3 + -5`.

More generally, the physical layout of components should be so designed as to foster the readability of software texts. For example, indentation (using tab characters) highlights the structure of nested components. Since readability will benefit from consistency, this book introduces some recommended style conventions. Appendix A.

32.8 LETTER CASE

The conventions on letter case were introduced at the beginning of this book. Here is the precise rule. ← “TEXTUAL CONVENTIONS”, 2.13, page 102.



Letter Case rule

Letter case is significant for the following constructs: Character_constant and Manifest_string except for special character codes, Comment.

For all other constructs, letter case is not significant: changing a letter to its lower-case or upper-case counterpart does not affect the semantics of a specimen of the construct.

In particular, letter case is not significant for identifiers and for reserved words; remember the notion of “same feature name”, which ignores letter case. ← “Same feature name, same operator, same alias” → Appearing throughout the book and collected in chapter 34.

This policy goes with a precise set of style guidelines enjoining you to use specific conventions for specific constructs, in particular identifiers (class names in all upper case, variable identifiers in all lower case etc.).

32.9 TOKEN CATEGORIES



Tokens are the basic meaningful elements of software texts.

As noted in the description of general conventions at the beginning of this book, tokens are specimens of **terminal constructs**. For example the token *8940* is a specimen of the terminal construct **Integer**. (In contrast, higher-level syntactical structures, such as class texts or routines, are specimens of non-terminal constructs such as **Class** or **Routine**.) Terminal constructs do not appear in left sides of the productions of the grammar; instead, their structure is defined in this chapter.

← *“THE LEXICAL LEVEL”, 2.4, page 87.*

There are two categories of tokens, fixed and variable:

- **Fixed tokens** have a single, frozen form. They include reserved words such as **class** or *Current*, and special symbols such as **:=**. For fixed tokens this book not distinguish between the form of a token and the underlying terminal construct. For example, **class** is the single specimen of a construct which could be called **Class_keyword** but remains implicit; an occurrence of the token in the grammar denotes the construct.
- **Variable tokens** are specimens of terminal constructs such as **Integer**, **Identifier**, **Free_binary**, for which this chapter defines a general structure, within which you can define tokens that fit the needs of your software. For example, the rules for **Integer**, given below, permit specimens made of one or more decimal digits; a token such as *327* satisfies this specification.

The following sections examine reserved words, special symbols, and the various terminal constructs defining variable tokens: **Identifier**, **Integer**, **String**, **Simple_string**, **Real**, **Operator**, **Character**.

32.10 RESERVED WORDS

Reserved word, keyword						
The following names are reserved words of the language.						
agent	alias	all	and	as	assign	attribute
check		class		convert		create
Current		debug		deferred		
do		else		elseif		end
ensure		expanded		export		
external		False		feature		from
frozen		if		implies		
inherit		inspect		invariant		like
local		loop		not		
note		obsolete		old		once
only		or		Precursor		
redefine		rename		require		rescue
Result		retry		select		
separate		then		True		<i>TUPLE</i>
undefine		until		variant		
Void		when		xor		
The reserved words that serve as purely syntactical markers, not carrying a direct semantic value, are called keywords ; they appear in the above list in all lower-case letters.						

The non-keyword reserved words, such as **True**, have a semantics of their own (**True** denotes one of the two boolean values).

The Letter Case rule applies to reserved words, so the decision to write keywords in all lower case is simply a style guideline. Non-keyword reserved words are most closely related to constants and, like constants, have — in the recommended style — a single upper-case letter, the first; *TUPLE* is most closely related to types and is all upper-case.

---- MOVE AND REWRITE The first and simplest tokens are reserved words, listed in an appendix. Each is a sequence of letters, with in two cases — **and then, or else** — an intervening blank (normally just one, but we tolerate more). Formally: → *Appendix L*.

Reserved words are called that way because you may not choose them for your own identifiers. They include **keywords** and **predefined names**: *See below on identifiers.*

Syntax (non-production): Double Reserved Word rule

The reserved words **and then** and **or else** are each made of two components separated by one or more blanks (but no other break characters). Every other reserved word is a sequence of letters with no intervening break character.

- Keywords, such as **class** and **feature**, introduce and delimit the various components of constructs.
- Predefined names come at positions where variable tokens would also be permissible: *Result*, denoting the result of a function, may appear in lieu of a local variable, for example as target of an assignment; *INTEGER* may appear at a position where a type is expected.

The definition of Result as a special kind of local variable appeared in 8.6, page 114..

In accordance with the Letter Case rule, letter case is not significant for reserved words, so that *CLASS*, *result* or even *rEsULt* are permissible forms. According to the style rules, however:

- Keywords appear in lower-case, as with **class**. In a typeset text they should always appear in **bold**, as in this book.
- Predefined names start with a capital letter; the rest is in lower case (as with *Result*), except for types since the general convention for all types is to use all upper-case (as with *INTEGER*). When typeset, they appear in italics.

The following general guidelines presided over the choice of reserved words and should help you to learn reserved words quickly and remember them without hesitation:

- Reserved words are simple and common English words. They are never abbreviations and, with one exception, they are never composite words.

The exception is **elseif**, denoting a simple idea for which there is no one-word name in English.

- For simplicity and consistency, the grammatical form is always the shortest. For a noun it's the singular, even if the plural might seem more natural: the clause introducing the features of a class begins with the keyword **feature**. For a verb it's in the infinitive form, as in **require**, again without an "s".

32.11 SPECIAL SYMBOLS

A small number of one- and two-character strings, called special symbols, have a special role in the syntax of various constructs.



Earlier chapters have introduced these symbols in connection with the syntactic form of various constructs. Here is the complete list:

Special symbol

A **special symbol** is any of the following character sequences:

```
-- : ; , ? ! ' " $ . -> :=
= /= ~ /~ ( ) (| ) [ ] { }
```

The following table gives a reminder of their role and the page where the corresponding syntax productions appear.

Symbol	Name	Role	Pages
--	Double dash	Introduces comments.	883
;	Semicolon	Separates instructions, declarations, assertion clauses...; always optional.	
,	Comma	Separates elements in lists of of entities or expressions.	
:	Colon	Separates the Type_mark in a declaration, a Tag_mark in an Assertion_clause , and a Note_name term in a Notes clause.	
'	Single quote	Encloses manifest constants.	
"	Double quote	Encloses manifest strings.	
%	Percent	Introduces special character codes.	
/	Slash	In a special character code, introduces a character through its code.	
+ -	Plus and minus	Signs of integer and real constants. (Also permitted as prefix and infix operators, appearing in a separate table.)	788
\$	Dollar	Address operator for passing the address of an Eiffel feature or expression to a routine (usually external).	833
%	Percent	Introduces a special character code.	
/	Slash	In a special character, introduces a character by its numerical code.	
.	Dot	Separates target from feature in a feature call or creation call. Separates integer from fractional part in a real number.	
->	Arrow	Introduces the constraint of a constrained formal generic parameter.	
:=	Receives	Assignment operator.	
= /=	Equal, not-equal signs	Equality and non-equality operators.	
~ /~	Tilde, slash-tilde	Object equality and non-equality operators.	
()	Parentheses	Group subexpressions in operator expressions; enclose formal and actual arguments of routines.	
()	Target parentheses	Enclose a constant or non-atomic expression used as target of a call in dot or bracked notation.	
[]	Brackets	Enclose formal and actual generic parameters to classes; enclose items of a manifest tuple; specify that a feature has a Bracket alias.	
{ }	Braces	Enclose types in various contexts: Clients part, Feature_clause or New_export_list , Creation_type .	



The special symbols must be written as given in the above table, with no intervening blanks or other characters. They should be typeset in roman.

32.12 IDENTIFIERS

So much for fixed tokens; on to variable tokens.



An important category of variable tokens is identifiers, describing symbolic names which class texts use to denote various components such as classes, features or entities.

Here are some example identifiers:



```
A
LINKED_LIST
a
an_identifier
feature_1
```

The construct is defined as follows:



Syntax (non-production): Identifier

An **Identifier** is a sequence of one or more alpha numeric characters of which the first is a letter.

According to the earlier definitions of character categories this implies that an identifier must start with a letter (not a digit or an underscore) and continue with zero or more letters, digits and underscores.

← “Letter, alpha betic, numeric, alpha numeric, printable”, page 880.

Identifiers are subject to a number of restrictions to avoid ambiguity; see for example the Entity rule, which prevents you from using the same identifier to declare two entities within a given scope. But these are rules on higher-level constructs, such as **Entity**, relying on identifiers. For themselves, identifiers are only subject to a basic validity constraint on the choice of name:

← “VEEN”, page 513.



Identifier rule

VIID

An **Identifier** is valid if and only if it is not one of the language’s reserved words.

Two of the reserved words, **and then** and **or else**, include a blank and would not be lexically acceptable as identifiers anyway. Their components — **and, or, then, else** — are themselves all reserved.

There is no limit to the length of identifiers, and all characters are significant: to determine whether two identifiers are the same or not, you must take all their characters — but not letter case — into account.

← “Same feature name, same operator, same alias”, page 153; “Letter Case rule”, page

32.13 OPERATORS

Operators appear as **alias** for identifier features when you wish to let your clients call the feature in infix or prefix form, as with

DEFINITION
T

```
no_better_than alias "<=" (other INVESTMENT): BOOLEAN
-- Is other a least as good as current investment?
... Rest of function declaration omitted ...
```

which, can then be called under the form *inv1 <= inv2* as well as the standard dot-notation call *inv1.no_better_than(inv2)*.

← “*OPERATOR FEATURES*”, 5.15, page 154.

There are three kinds of operators: **predefined**, **standard** and **free**.

There are only four *predefined* operators:

DEFINITION
T

Predefined operator

A **predefined operator** is one of:

= /= ~ /~

These operators — all “special symbols” — appear in **Equality** expressions. Their semantics, reference or object equality or inequality, is defined by the language (although you can adapt the effect of *~* and */~* since they follow redefinitions of *is_equal*). As a consequence you may not use them as **Alias** for your own features.

The *standard* operators, used as aliases for features of the basic types (*INTEGER* etc.). The list given in the discussion of features includes boolean operators (such as **not**, **implies**, **and**, **or**) which lexically are keywords, leaving the following as standard operators in the lexical sense:

← Page 154.

Standard operator

A **standard unary operator** is one of:

+ -

A **standard binary operator** is any one of the following one- or two-character symbols:

+ - * / ^ < >
<= >= // \ \ ..

All the standard operators appear as **Operator** aliases for numeric and relational features of the Kernel Library, for example *less_than alias "<"* in *INTEGER* and many other classes. You may also use them as **Alias** in your own classes.

The above *no_better* example illustrated such a use as **Alias**.

Free operators allow you to make up your own operators, for example to support specific notations in mathematics or physics. You can use almost any combination of “operator symbols”, a notion defined very broadly:

DEFINITION

Operator symbol

An **operator symbol** is any non-alpha numeric printable character that satisfies any of the following properties:

- 1 • It does not appear in any of the special symbols.
- 2 • It appears in any of the standard (unary or binary) operators but is neither a dot `.` nor an equal sign `=`.
- 3 • It is a tilde `~`, percent `%`, question mark `?`, or exclamation mark `!`.

Condition 1 avoids ambiguities with special symbols such as quotes. Conditions 2 and 3 override it when needed: we do for example accept as operator symbols `+`, a standard operator, and `\` which appears in a standard operator — but not a dot or an equal sign, which have a set meaning.

Thanks to this notion, the definition of free operators lets you make up your own notations as long as they cause no ambiguity:

DEFINITION

Free operator

A **free operator** is sequence of one or more characters satisfying the following properties:

- 1 • It is not a special symbol, standard operator or predefined operator.
- 2 • Every character in the sequence is an operator symbol.
- 3 • Every subsequence that is not a standard operator or predefined operator is distinct from all special symbols.

A **Free_unary** is a free operator that is distinct from all standard unary operators.

A **Free_binary** is a free operator that is distinct from all standard binary operators.

← From the table in
“SPECIALSYMBOLS”,
32.11, page 889.

Condition [3](#) gives us maximum flexibility without ambiguity; for example:

- You may **not** use `---` as an operator because, its subsequence `--` clashes with the special symbol introducing comments.
- You may similarly **not** use `--` because the full sequence (which of course is a subsequence too) could still be interpreted as making the rest of the line a comment.
- You **may**, however, use a single `-`, or define a free operator such as `-*` which does not cause any such confusion.
- You may **not** use `?`, `!`, `=` or `~`, but you **may** use operators containing these characters, for example `!=`.
- You **may** use a percent character `%` by itself or in connection with other operator symbols. No confusion is possible with character codes such as `%B` and `%/123/`. (If you use a percent character in an `Alias` specification, its occurrences in the `Alias_name` string must be written as `%%` according to the normal rules for special characters in strings. For example you may define a feature `remainder alias "%%"` to indicate that it has `%` as an `Operator` alias. But any use of the operator outside of such a string is written just `%`, for example in the expression `a % b` which in this case would be a shorthand for `a.remainder (b)`.)

Alpha_numeric characters are not permitted. For example, you may not use `+b` as an operator: otherwise `a+b` could be understood as consisting of one identifier and one operator.

Remember that — style guidelines aside — we do **not** want to require breaks between an operator such as `+` and the following identifier such as `b`, so that we can interpret `a+b` as an expression.

← “*Syntax (non-production): Break rule*”, page 885.

The last two parts of the definition separate “free” unary and binary operators from the standard ones. They allow, for example, defining `*` as unary operator in one of your classes — whether or not it also uses it as binary operator — even though, in the basic types `INTEGER`, `REAL` and their sized variants, it figures only as binary.

This rule still leaves you considerable room in choosing free operators to match the needs of just about any application domain, as illustrated by other examples: `**`, `|-`, `<->`, `-|>`, `=>` and many others.

General-purpose libraries such as EiffelBase and EiffelVision make very limited use of free operators; this facility is mostly for specialized application domains that are accustomed to their own notations.

32.14 CHARACTERS



Characters — specimens of construct `Character` — are used in various constructs: `Character_constant` (of which a specimen is a `Character` in single quotes, as `'A'`); `Manifest_string` (zero or more characters in double quotes, as in `"ABC DE!#$"`); `Identifier`.

A character is an element of the character set as defined at the beginning of this chapter: either Unicode or extended ASCII. To define the notion properly let us assume that the device used to enter software texts is a keyboard, offering its users a number of keys, each defined by a code. We must distinguish between *keys*, which simply serve to enter certain codes, and *characters*, the atoms of Eiffel lexical elements.

In the simplest and most common case, of course, you enter a character just by pressing an associated key. For certain characters, however, you may have to press a succession of two or more keys; and some keys do not yield a character at all. For example the `Manifest_string` in

`String_with_backspace: INTEGER is "AAA %B ZZZ"`

includes a non-printable character, Backspace, appearing as `%B`. For this character there is in fact a key on usual keyboards, but pressing it does not yield a character: it simply erases the previous character you have entered. To make Backspace part of your string, you may represent it by the two-key sequence `%B`, as here. Another possibility, using the numerical code for this character, is to enter it as `%/8/`.

The definition of “character” must be general enough to encompass all such cases and address portability problems raised by the different in keyboards found in various countries:



Syntax (non-production): Manifest character

A **manifest character** — specimen of construct `Character` — is one of the following:

- 1 • Any key associated with a printable character, except for the percent key `%`.
- 2 • The sequence `%k`, where *k* is a one-key code taken from the list of special characters.
- 3 • The sequence `%/code/`, where *code* is an unsigned integer in any of the available forms — decimal, binary, octal, hexadecimal — corresponding to a valid character code in the chosen character set.

Appearing on the next page.

Form [1](#) accepts any character on your keyboard, provided it matches the character set you have selected (Unicode or extended ASCII), with the exception of the percent, used as special marker for the other two cases.

Form [2](#) lets you use predefined percent codes, such as `%B` for backspace, for the most commonly needed special characters. The set of supported codes follows.

Form [3](#) allows you to denote any Unicode or Extended ASCII character by its integer code; for example `%/59/` represents a semicolon (the character of code 59). Since listings for character codes — for example in Unicode documentation — often give them in base 16, you may use the [0xNNN convention](#) for hexadecimal integers: the semicolon example can also be expressed as `%/0x3B/`, where `3B` is the hexadecimal code for 59.

→ Introduced later in this chapter: [“INTEGERS”](#), [32.16, page 899](#).

Since the three cases define all the possibilities, a percent sign is illegal in a context expecting a **Character** unless immediately followed by one of the keys of the following table or by `/code/` where `code` is a legal character code. For example `%?` is illegal (no such special character); so is `%0x/FFFFFF/` (not in the Unicode range).

Special characters and their codes

<i>Character</i>	<i>Code</i>	<i>Mnemonic name</i>
@	%A	At-sign
BS	%B	Backspace
^	%C	Circumflex
\$	%D	Dollar
FF	%F	Form feed
\	%H	Backslash
~	%L	Tilde
NL (LF)	%N	Newline
`	%Q	BackQuote
CR	%R	Carriage Return
#	%S	Sharp
HT	%T	Horizontal Tab
NUL	%U	NULL
	%V	Vertical bar
%	%%	Percent
'	%'	Single quote
"	%"	Double quote
[%([Opening bracket
]	%)	Closing bracket
{	%{	Opening brace
}	%}	Closing brace

A few of these codes, such as the last four, are present on many keyboards, but sometimes preempted to represent letters with diacritical marks; using `%()` rather than `[` guarantees that you always get a bracket.

The major application of forms [2](#) and [3](#) is to express a **Manifest_string** containing characters that you cannot type directly into the class text. This includes non-printable characters, such as Backspace (see [String_with_backspace](#) above), and others not supported on all keyboards. If you have an American ASCII keyboard but want to define a **Manifest_string** that output devices supporting the appropriate codes will display as *ambiguïté*, with two letters bearing diacritical marks, you may enter it as the string `"ambigu%/139/t%/130"`.

The codes are also useful in defining character intervals for `Inspect` instructions, as in

```
inspect
  entry                -- Of type CHARACTE
when %/128/ .. %/165/ then
  >...
... Other clauses ...
end
```

Since this convention is of no particular benefit for entering such tokens as identifiers, it is reserved for character and string constants:

Syntax (non-production): Percent variants

The percent forms of `Character` are available for the manifest characters of a `Character_constant` and of the `Simple_string` components of a `Manifest_string`, but not for any other token.

The characters “of” such a constant do not include the single `'` or double `''` quotes, which you must enter as themselves.

The use of the percent character

The semantic specification follows the cases of the syntax definition:

Manifest character semantics

The value of a `Character` is:

- 1 • If it is a printable character `c` other than `%`: `c`.
- 2 • If it is of the form `%k` for a one-key code `k`: the corresponding character as given by the table of special characters.
- 3 • If it is of the form `%/code/`: the character of code `code` in the chosen character set.

32.15 STRINGS



Do not confuse `String` or `Simple_string` with `Manifest_string`, seen in the discussion of expressions. A specimen of `Manifest_string`, a non-terminal construct, is a `Simple_string` in double quotes, as in `"SOME STRING"`. ← [29.8, page 794](#).

In the definition of `String`, a “character” is any legal Eiffel character as defined in the preceding sections. This includes in particular:

- A keyboard key other than `%`.

See [32.14, page 894](#), for the various forms of characters and the use of the percent sign.



Syntax (non-production): String, simple string

A **string** — specimen of construct **String** — is a sequence of zero or more manifest characters.

A **simple string** — specimen of **Simple_string** — is a **String** consisting of at most one line (that is to say, containing no embedded new-line manifest character).

- A special character code such as **%B**
- A character given by its numerical code, such as **%/59/** or **%/0b3B/**.

The semantics is straightforward:

String semantics

The value of a **String** or **Simple_string** is the sequence of the values of its characters.

32.16 INTEGERS

Integer, a lexical construct, describes unsigned integer values. You may express an **Integer** in either the usual decimal notation or in one of three bases other than 10: binary (base 2), octal (base 8), hexadecimal (base 16).

← Do not confuse **Integer** with **Integer_constant**, seen s in “[INTEGER CONSTANTS](#)”, 29.5, page 792. A specimen of **Integer_constant**, a non-terminal construct, is an **Integer** optionally preceded by a sign.

Examples of the most common case, decimal notation are:



```
0
327
3197865
3_197_865
```

Except for underscores, no intervening characters (such as blanks) are permitted between digits.



You can use underscores to improve readability by dividing a long integer into pieces. The recommended convention, as in the last example is to use groups of three digits from the right (so that the leftmost one may be shorter). Underscores have no effect on the value: the last two examples denote the same integer value.

Examples in non-decimal notation, all representing the number twenty-nine, are:

```
0b11101      -- Binary
0c35         -- Octal
0x1D         -- Hexadecimal
```

The convention is clear: the constant starts with the digit **0**, followed by a code for the base (**b** for Binary, **c** for octal, the conventional **x** for hexadecimal), followed by digits meaningful for the appropriate base.

Here too underscores may be used to group digits, as in **0b1_1101**, with no particular style guideline.



As with other lexical elements, letter case is not significant for the **Integer_base** (so that **0X**, for example, is acceptable in lieu of **0x**) and for the hexadecimal digits **A** to **F**. The forms given, such as **0x1D**, indicate the recommended style: lower case for the base and upper case for the digits.

← “*Syntax (non-production): Double Reserved Word rule*”, page 889.

To describe this structure it is best to resort to a syntax production and a validity rule, with the understanding that unlike with ordinary syntax productions (and as indicated in the first clause of the validity rule) the successive elements are not separated by breaks. The syntax is



Integers
$\text{Integer} \triangleq [\text{Integer_base}] \text{Digit_sequence}$
$\text{Integer_base} \triangleq \text{"0"} \text{Integer_base_letter}$
$\text{Integer_base_letter} \triangleq \text{"b"} \mid \text{"c"} \mid \text{"x"} \mid \text{"B"} \mid \text{"C"} \mid \text{"X"}$
$\text{Digit_sequence} \triangleq \text{Digit}^+$
$\text{Digit} \triangleq \text{"0"} \mid \text{"1"} \mid \text{"2"} \mid \text{"3"} \mid \text{"4"} \mid \text{"5"} \mid \text{"6"} \mid \text{"7"} \mid \text{"8"} \mid \text{"9"} \mid \text{"a"} \mid \text{"b"} \mid \text{"c"} \mid \text{"d"} \mid \text{"e"} \mid \text{"f"} \mid \text{"A"} \mid \text{"B"} \mid \text{"C"} \mid \text{"D"} \mid \text{"E"} \mid \text{"F"} \mid \text{"_"} \mid$

To introduce an integer base, use the digit **0** (zero) followed by a letter denoting the base: **b** for binary, **c** for octal, **x** for hexadecimal. Per the Letter Case rule the upper-case versions of these letters are permitted, although lower-case is the recommended style.

Similarly, you may write the hexadecimal digits of the last two lines in lower or upper case. Here upper case is the recommended style, as in **0xA5**.

The associated constraint is:



Integer rule

VIIN

An **Integer** is valid if and only if it satisfies the following conditions:

- 1 • It contains no breaks.
- 2 • Neither the first nor the last **Digit** of the **Digit_sequence** is an underscore “_”.
- 3 • If there is no **Integer_base** (decimal integer), every **Digit** is either one of the decimal digits **0** to **9** (zero to nine) or an underscore.
- 4 • If there is an **Integer_base** of the form **0b** or **0B** (binary integer), every **Digit** is either **0**, **1** or an underscore.
- 5 • If there is an **Integer_base** of the form **0c** or **0C** (octal integer), every **Digit** is either one of the octal digits **0** to **7** or an underscore.

The rule has no requirement for the hexadecimal case, which accepts all the digits permitted by the syntax.

Integer is a purely lexical construct and does not include provision for a sign; the construct Integer_constant denotes possibly signed integers. ← *“INTEGER CONSTANTS”, 29.5, page 792.*

Finally, the semantics:



Integer semantics

The value of an **Integer** is the integer constant denoted in ordinary mathematical notation by the **Digit_sequence**, without its underscores if any, in the corresponding base: binary if the **Integer** starts with **0b** or **0B**, octal if it starts with **0c** or **0C**, hexadecimal if it starts with **0x** or **0X**, decimal otherwise.



This definition always yields a well-defined mathematical value, regardless of the number of digits. It is only at the level of Integer_constant that the value may be flagged as invalid, for example `{NATURAL_8} 256`, or `999 ... 999` with too many digits to be representable as either an `INTEGER_32` or an `INTEGER_64`.

The semantics ignores any underscores, which only serve to separate groups of digits for clarity. With decimal digits, the recommended style, if you include underscores, is to use groups of three from the right.

32.17 REAL NUMBERS

Real numbers – specimens of construct **Real** – define the manifest constants for the basic types **REAL** and its sized variants.

The following are real numbers:



1.0	1.
0.1	.1
2345.632E-7	2345.632e-7

Here is the definition:



Syntax (non-production): Real number

A **real** — specimen of **Real** — is made of the following elements, in the order given:

- An optional decimal **Integer**, giving the integral part.
- A required “.” (dot).
- An optional decimal **Integer**, giving the fractional part.
- An optional exponent, which is the letter *e* or *E* followed by an optional **Sign** (+ or –) and a decimal **Integer**.

No intervening character (blank or otherwise) is permitted between these elements. The integral and fractional parts may not both be absent.

As with Integer, there is no sign; only a Real_constant may introduce the sign.



As with integers, you may use underscores to group the digits for readability. The recommended style uses groups of three in both the integral and decimal parts, as in `45_093_373.567_21`. If you include an exponent, *E*, rather than *e*, is the recommended form.

The integral part, fractional part and exponent, all specimens of **Integer**, must be expressed in decimal (no binary, octal or hexa).

The denoted value is the expected one:



Real semantics

The value of a **Real** is the real number that would be expressed in ordinary mathematical notation as $i.f10^e$, where *i* is the integral part, *f* the fractional part and *e* the exponent (or, in each case, zero if the corresponding part is absent).

As with integers, this semantics yields the *exact* mathematical value. Approximation to the supported floating-point type — *REAL*, *REAL_32* or *REAL_64* — occurs only when you use the *Real* as a Real constant. ← *“REALCONSTANTS”*, [29.6, page 792](#).

Concurrency *(not done)*

33.1 OVERVIEW

Style guidelines *(not done)*

34.1 OVERVIEW

To facilitate the exchange of Eiffel software, it is preferable to follow a standardized programming style. This appendix describes a set of guidelines which help in this effort. Clearly, these rules are not part of the language definition.

Many of the rules are rather low-level, dealing with such mundane questions as how to phrase comments, where to put blanks adjacent to parentheses, and whether to use verbs or nouns for routine names. Modest as some of these concerns may seem, they are not to be neglected. Adherence to a uniform style for the more superficial aspects of software texts may indeed be of great benefit to both readers and writers of Eiffel software:

In the process of getting acquainted with previously written classes, you will feel more comfortable if they follow a commonly agreed style, enabling you to understand the details more accurately, and to move on without delay to the deeper aspects of the classes under review.

When you write new classes, or modify existing classes, the existence of simple, well-defined guidelines helps you avoid wasting your time hesitating on minor issues.

Far from stifling their creativity, then, the style discipline described in this chapter encourages software developers to apply it to the true challenges of quality software engineering: design of elegant, modular system architectures; selection of appropriate data structures; and use of the best possible algorithms.

34.2 LETTER CASE

Letter case is not significant for entity, feature and type names. The recommended style observes the following conventions.

Any identifier that may be used as a type, or part of a type, should be written all in upper case. This includes:

- Class names, in all possible uses.
- Formal generic parameters, such as *G* in *LIST [G]*.
- Basic types (*BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, their sized variants and *POINTER*), *ARRAY* and *STRING*.

Constant attributes should be written with an initial capital letter, with the rest in lower case, as in

Area: REAL is 43_512/.g57;
Red, Green, Blue, Yellow: *INTEGER*

The same convention, initial upper-case letter, also applies to Void, the entity of type NONE representing void references, and to the two predefined entities *Current* and *Result*.

All other features (variable attributes and routines) and local variables should use all-lower-case names.

34.3 CHOICE OF NAMES

Names for features and entities should be clear and informative. Do not use abbreviations, except possibly for formal routine arguments, which are only used in a restricted context.

Complex names should use the underscore character to connect various components, as in

put_right

The use of internal upper-case letters for the same purpose, as in *putAtRight*, contradicts the standard conventions of English and most other languages and is not part of the recommended style.

.FS

Apart from proper names of the form *MacName* or *McName*, the only common use of internal upper-case letters seems to be for composite proper names of French origin as spelled in North American English. This convention, however, is unknown in actual French.

If two related names have some elements in common, make them differ at the beginning, rather than at the end; for example, use *x_position* and *y_position* rather than *position_x* and *position_y*. This will decrease the probability of confusion.

Clarity does not imply length. Although names may be as long as needed to avoid ambiguity, you should resist the temptation to overqualify. In particular, feature names should not include an identification of the enclosing class. For example, a feature for updating a customer's invoice in a class *INVOICE* should be called *update*, not *update_invoice* or *invoice_update*.

The design of the Basic Eiffel Libraries has gone even further in the direction of simplifying and standardizing feature names. This means in particular that the Data Structure Library makes little use of the specific terminology traditionally applied by the computer science literature to each individual kind of data structure. For example, you will **not** find features called

push, pop, top for stacks.

add, remove, oldest, latest for queues.

enter, entry for arrays.

insert, value, search for lists and hash tables.

Such names, widely used in textbooks about algorithms and data structures, highlight the differences between the various structures rather than their common properties.

In contrast, the Eiffel libraries are based on a taxonomy of data structures, grouped into well-structured families such as “dispensers”, “chains” and “tables”. The taxonomy is directly implemented into the library through multiple inheritance from separate hierarchies of deferred classes.

For an in-depth discussion of these issues and other aspects of library design, see “Eiffel: The Libraries” and the chapter entitled “Lessons from the Design of the Eiffel Libraries” in “An Eiffel Collection”. The references are in the bibliography of appendix =====.

Feature names are in line with this approach; they reflect the deeper common properties rather than the superficial differences. Some of the most important universal names are: *make* (Basic initialization operation; should be creation procedure) *item* (Basic access operation. *p*) *count* (Number of significant items in a structure.) *put* (Basic operation to insert or replace an item.) *force* (Like *put*, but will always succeed when it can. For example, it may resize the structure if full.) *remove* (Basic operation for removing an item.) *wipe_out* (Basic operation for removing all items.) *empty* (Test for absence of significant items. Should return the same value

as *count = 0*.) *full* (Test for lack of space for more items.) *to_external* (Function providing a pointer to actual data structure, for example the sequence of values making up an array or string, useful for transmission to external routines. May have language-specific variants such as *to_c* or *to_fortran*. All such functions should have a result of type *NONE* to preclude any feature application on the Eiffel side.)

from_external

(Inverse of *to_external*: procedure to reinitialize a data structure such as a string from an external form. May have language-specific variants such as *from_c*.)

Although such names as *item*, *put* and *remove* for stacks (replacing the traditional *top*, *push* and *pop*) may be a shock to some users accustomed to the more traditional terminology, this unifying move was felt inevitable if client users are to master easily a large number of powerful reusable classes describing many data structures variants.

The inevitable differences in signatures and specifications should not be compounded by differences in names which (in a typed language where incorrect calls will be detected automatically) only stand in the way of understanding.

If you are familiar with these conventions, you will easily recognize the purpose of the major routines when you explore a class that follows them, and you will be able to find out quickly whether the class suits your needs.

Thanks to their systematic presence in the Basic Libraries, these names have acquired a status which is next in importance to that of the language keywords. Make sure you use them whenever they are applicable.

34.4 GRAMMATICAL CATEGORIES FOR FEATURE NAMES

Since procedures are commands to perform actions, their names should be drawn from **verbs** in the imperative mode: *put*, *write*, *remove* etc.

In contrast, functions and attributes (which are indistinguishable by clients, except through the presence or absence of arguments) describe access to information. A function or attribute of a type other than *BOOLEAN* should usually use a **noun**, possibly qualified by an adjective, as in *item* or *last_transaction*. Sometimes the noun may be implicit; then only the adjective or adjectives remain, as in *last_read*, which really stands for *last_item_read*.

The names of boolean functions or attributes should be of either of two forms:

The name may suggest a question, usually with the prefix *is_*, as in *is_leaf* for a boolean feature used to determine whether a node is a leaf.

Adjectives are also appropriate in some cases, as in *opened* for a boolean feature used to determine whether a file is open.

“opened” rather than “open” because the later might be confused with a verb, indicating a command to open the file.

Of the two possible names for a boolean feature, the one chosen should suggest the property which is **false** in the default case. This is because the default initialization rules will initialize a boolean attribute to false, so that there will be no need for the creation procedure to include a specific initialization. For example, if files are to remain closed until some call explicitly opens them, use an attribute *opened* rather than *closed*. Then the creation procedure of the class does not need to do anything special for this attribute.

.IA

Using a verb such as *get* for a function is usually inappropriate. Functions should not “do” something, but return information in a non-destructive way.

See “Object-Oriented Software Construction” about side effects in functions.

For example, a sequential read operation which advances the input cursor may be implemented as the combination of a procedure *get*, changing the state, and an attribute or function *last_item*, returning the last element read. A call to *get* updates the value of *last_item*, but calling *last_item* several times in a row repeatedly yields the same result.

The input routines from the Kernel Library class STANDARD_FILES, discussed in chapter =====, follow these rules: a procedure such as ‘readint’ will read an element, and an attribute such as ‘lastint’ will give access to the last value read.

34.5 GROUPING FEATURES

Classes introducing many features should group them into logical categories. The syntax encourages this by allowing a class to have more than one Feature_clause, each beginning with a Header_comment. (A Header_comment has the same form as a free-comment, but appears as an official although optional component of some construct in the syntax. The constructs which take header comments are Feature_clause, Creation_clause and Routine. The next section will examine header comments of routines.)

Syntax productions: Feature_clause, *page* =====; Creation_clause, *page* =====; Routine, *page* =====.

The presentation of features sketched a class text organized in this way: the Data Structure Library class *LINKED_LIST*, with feature groups introduced by

See =====, page =====.

- Number of elements
- Special elements
- Cursor movement
- Chaining
- Representation

Such header comments should be short, simple phrases characterizing a set of logically related features.

In some cases it may be more convenient to read a class text in alphabetical order of feature names. Part D of this book indeed used this convention for most of its presentations of library classes in flat-short form. In the class texts themselves, however, grouping by feature categories is usually better; then a good language processing tool (for example through options of the **short** and **flat** commands) will be able to produce alphabetical output from a class organized by category — the reverse being of course impossible.

See =====, page =====, about **short**, and =====, page =====, about **flat**.

34.6 HEADER COMMENTS

Every routine should begin with a Header_comment. Here is an example:

```

distance_to_origin: REAL
-- Distance to point (0, 0)
local
origin: POINT
do
create origin/.gset_to_origin;
Result := distance (origin)
end

```

Header comments should be informative, clear, and concise. In general, brevity is one of the essential qualities of comments in programs; over-long comments tend to obscure the program text rather than help the reader. The following principles should help achieving brevity.

Avoid repeating information which is obvious from the immediately adjacent program text. For example, the header comment for a routine beginning with

```
tangent_to (c: CIRCLE; p: POINT): LINE
```

should not be

-- Tangent to circle c through point p

but just

-- Tangent to c through p

as it is clear from the function header that c is a circle and p is a point.

For the same reason, the header comment should not usually include restrictions on using the routine (such as “Call only on non-void argument”) since such restrictions are the business of the Precondition clause, which will give a more complete and more precise view.

Avoid noise words and phrases. An example is “Return the...” in explaining the purpose of functions. In the above cases, writing “Return the distance to point (0, 0)” or “Return the tangent to...” does not bring any useful information as the reader knows a function must return something. Another example of a noise phrase is “This routine computes...”, or “This routine performs...”. Instead of

-- This routine updates the display according to the user’s last input

write

-- Update display according to last user input.

Every header comment should begin with an upper-case letter.

Do not use abbreviations in header comments. The purpose of a comment is to explain; a reader may not know the meaning of an abbreviation.

Header comments should have the following syntactical form, which parallel rules given above for routine names:

The header comment for a procedure should be a sentence in the imperative, as in the last example. The sentence should end with a period.

The header comment for a non-boolean function should be a nominal phrase, such as “Tangent...” above. A final period is not necessary in this case, unless the comment contains more than one sentence.

The header comment for a boolean function should be a question, ending with a question mark, as in “Is current node a leaf?”.

Header comments should be consistent. If a function of a class has the comment “Length of string”, a routine of the same class should not say “Update width of string” if it acts on the same attribute.

In general, comments should be of a level of abstraction higher than the code that they document. In the case of header comments, the comment should concentrate on the “what” of the routine rather than the “how” of the algorithm used.

Finally, remember that much of the important semantic information about the effect of a routine may be captured more precisely and concisely through the Precondition and Postcondition clause than through natural language explanations.

34.7 OTHER COMMENTS

Although this does not appear in the syntax, a class should also begin with a comment. The class comment should be brief and come before the beginning of the class text proper. For a class describing a set of objects, the comment should characterize these objects in the plural, as in

```
-- Binary search trees, pointer representation
```

Other parts of class texts may also include free-comments, used to explain potentially unclear components. They should be indented to the right of the normal text so as not to interfere with the understanding of the software text proper.

Classes and routines should have ending comments repeating their names. These comments are in fact optional parts of the syntax.

See =====, page ===== for comments ending classes and =====, page ===== for comments ending routines.

It is not necessary to label the end of a control structure by a closing comment (such as in “**end** -- *if*”). The nesting depth of control structures should remain small in well-written Eiffel texts, not requiring any supplementary help for matching the beginning and end of each structure.

34.8 EIFFEL NAMES IN COMMENTS

In the examples above in the rest of this book, the name of a feature or other entity appearing in a comment is shown in italics to avoid any confusion with common words. For example a header comment could be of the form:

```
-- Record element under key
```

where *element* and *key* are formal arguments of the enclosing routine.

The corresponding convention in actual software texts (where font variation is not a possibility) is to enclose such an Eiffel name in single quotes (one opening quote ‘, one closing quote ’). So the actual form of the above comment in its class text should be:

-- Record *element* under *key*

Language processing tools which produce typesettable forms of classes should recognize this convention and use italics for Eiffel names quoted in comments. (As seen below, italics is the recommended convention in typeset output.)

34.9 LAYOUT

The recommended layout of Eiffel software texts results from the general form of the syntax, which is essentially an “operator grammar”, meaning that any text is a succession of alternating “operators” and “operands”. An operator is a fixed language symbol, such as a keyword (**do** etc.) or a separator (semicolon, comma etc.); an operand is a user-chosen symbol (identifier or constant).

As a consequence, the text should follow a “comb-like” structure where every syntactical component either fits on a line together with a preceding operator, or is indented just by itself on one or more lines, as in a comb whose branches normally begin and end with operators:
.pF ""Comb-like layout"

For an example, depending on the size of its components *a*, *b* and *c*, the same Conditional may be written, among other possibilities, as

if *c* **then** *a* **else** *b* **end**

or

if
c
then
a
else
b
end

or

if *c* **then**
a
else *b* **end**

For indentation, you should always use tabs, not spaces. (The only exception is if you are using a text editor that doesn't handle tabs well. Most modern tools, however, have no such problem.)

The same principle applies to classes and routines. The following extract from the ARRAY class of the Kernel Library illustrates the standard indentation conventions for the different clauses. Ellipses (...) indicate omitted features.

```
.t1
-- One-dimensional arrays
```

```
.t1
note
```

```
.t1
names: array;
access: index;
representation: array;
size: fixed, resizable
```

```
.t1
class ARRAY [T] creation
```

```
.t1
make
```

```
.t1
```

```
.t1
```

```
inherit
```

```
.t1
INDEXABLE [T, INTEGER];
```

```
.t1
INDIRECT [T];
```

```
.t1
BASIC_ROUT
```

```
.t1
```

```
.t1
```

feature

```
.t1
```

```
make (minindex, maxindex: INTEGER)
```

```
-- If minindex <= maxindex, allocate array with bounds
```

```
-- lower and upper; otherwise create empty array.
```

```
do
```

```
upper := -1;
```

```
-- lower initialized to 0 by default, so invariant holds
```

```
if minindex <= maxindex then
```

```
lower := minindex; upper := maxindex;
```

```
actual_lower := lower; actual_upper := upper;
```

```
allocate (maxindex - minindex + 1)
```

```
end
```

```
ensure
```

```
empty_if_impossible: minindex > maxindex implies count = 0;
```

```
consistent_size: minindex <= maxindex implies
```

```
(lower = minindex and upper = maxindex and
```

```
count = upper - lower + 1)
```

```
end;
```

```
.t1
```

```
.t1
```

```
lower: INTEGER;
```

```
-- Minimum current legal index
```

```
.t1
```

```
upper: INTEGER;
```

```
-- Maximum current legal index
```

```
.t2
```

```

    item (i: INTEGER): T
-- Entry of index i, if within bounds
require
index_large_enough: lower <= i;
index_small_enough: i <= upper
do
Result := ext_item (area, i — lower )
end;

```

```
.t2
```

```

    force (v: T; i: INTEGER)
-- Replace i-th entry by v.
-- Always applicable: resize if i not in current bounds.
local
extra_block_size: INTEGER
do
extra_block_size :=
max (Block_threshold, Extra_percentage star count %eidiv% Hundred);
if i < actual_lower then
resize (i — extra_block_size, upper);
lower := i
elseif i > actual_upper then
resize (lower, i + extra_block_size);
upper := i
else
lower := min (i, lower);
upper := max (i, upper)
end;
put (v, i);
ensure
inserted: item (i) = v;
higher_count: count >= old count
end;

```

```
...
```

```
.t1
feature {NONE} -- Representation details
```

```
.t1
```

```
actual_upper: INTEGER;
```

```
-- Actual upper bound
```

```
.t1
```

```
...
```

```
.t2
```

```
feature -- Obsolete features
```

```
.t1
```

```
enter_force (i: INTEGER, v: T)
```

```
obsolete "Use 'force (value, index)'"
```

```
do
```

```
force (v, i)
```

```
ensure
```

```
inserted: item (i) = v;
```

```
end
```

```
...
```

```
invariant
```

```
consistent_size: count = upper — lower + 1;
```

```
non_negative_size: count >= 0
```

```
end
```

Note in particular the indentation used for routine header comments.

The indentation step is the “tab” character. Blank characters should never be used for indentation.

34.10 OPTIONAL SEMICOLONS

Closely related to layout is the question of optional semicolons. For most repetition constructs which use the semicolon as separator, semicolons are optional (except between two adjacent specimens if the second one begins with an opening parenthesis).

The recommended style is to **omit the semicolons** between items appearing on successive lines. Extensive experimentation has shown that the semicolons impair readability since they add no information and detract from the meaningful components of the software text.

If successive items appear on the same line — as is sometimes useful for short multiple declarations or instructions — the semicolon should of course be included, since mere spaces are not visible enough to delimit the successive elements clearly.

Semicolons are optional between feature clauses (page =====), parent clauses (page =====), declarations of local variables (page =====), assertion clauses (page =====) and instructions (page =====).

34.11 LEXICAL CONVENTIONS

Lexical conventions follow the practice of ordinary text, both in language components and in comments. In particular:

There should be a blank before an opening parenthesis, and after a closing parenthesis, but none after an opening parenthesis or before a closing one. The same rule applies to square brackets.

A comma should always be followed by a blank, never preceded by one.

In a comment, a period should be followed by a blank, never preceded by one.

As an exception to non-software textual practice, the dot (period, full stop) used for qualified feature calls is neither preceded nor followed by a blank, as in *this_window/.gdisplay*. As this example indicates, it is preferable, for typeset texts, to use a very small bullet (appearing in this book and in the output of the **short** command when it is meant for typesetting), more visible than a dot.

Three further conventions govern the use of blanks:

An Assignment or Assignment_attempt symbol (:= or ?=) should be preceded and followed by one blank.

These are single symbols: it would be invalid to insert a blank before the = character.

Arithmetic operators should also have a blank to the left and one to the right. When typesetting an Eiffel text with asterisks in expressions, make sure they appear properly as star; by default, typesetting systems will often print an asterisk as * (appearing too high above the line).

As mentioned on page =====, however, a form with intervening blanks would be valid.

34.12 FONTS

When Eiffel texts are typeset, as in this book, the following font conventions should be observed.

Keywords should appear in bold italics:

class

Type, class, feature and entity names (including predefined types and entities) should appear in italics:

INTEGER
ARRAY
put
x
Result

Comments, especially header comments of routines, should appear in roman font, except for identifiers from the software — denoting classes, features, arguments — which should appear in italics:

-- Change lending rate to highest of *rate1* and *rate2*.

As mentioned above, these names should appear in single quotes in the corresponding source texts.

34.13 GUIDELINES FOR ANNOTATING CLASSES

The **Notes** clause which optionally begins a class text may be used to record information about the class, for use by class browsing and retrieval tools. Such tools are important in the Eiffel approach to software construction, based on the reuse of industrial quality software modules.

The very idea of a **Notes** clause assumes a degree of standardization of annotation conventions. This section introduces some important guidelines.

It is important first to put the overall purpose of the **Notes** clause in perspective. The general principle of documenting Eiffel software is that as much of the documentation as possible should be within the class texts themselves. Documentation and browsing tools should use these texts as their primary source of information.

See “Eiffel: The Environment” about documentation and browsing tools.

Some properties of class designs, however, are of a higher level of abstraction than what is usually expressed in the class text proper. They include annotation categories, descriptions of design and implementation decisions, references to algorithms or data structures as published in the literature etc. The **Notes** clause is meant for such information. It should record it in a standardized format for use by documentation, archival and retrieval tools. Such tools should enable users to retrieve archived classes using query languages that express queries based on *<index, value>* pairs.

The following guidelines were used in the Basic Eiffel Libraries and are recommended for other software as well.

- Keep the **Notes** clauses short (2 to 10 entries is typical).
- Avoid repeating information which is in the rest of the class text.

- Use a set of standardized indices for properties that apply to many structures (such as choice of representation).

Such standardized indices are suggested below.

For values, define a set of standardized possibilities for the common cases.

Include positive information only. For example, a *representation* index is used to describe the choice of representation (linked, array, ...). A deferred class does not have a representation. For such a class the clause should not contain the entry *representation: none* but simply no entry with the index *representation*. A reasonable query language will make it possible to use a query pair of the form *<representation, NONE>*.

Here are a few of the standard index terms and typical values.

An entry of index *names* records alternative names for a structure. Although a class has only one official name, the abstraction it implements may be commonly known under other names. For example, a “list” is also called a “sequence”.

An entry of index *access* records the mode of access of the data structures. The standard values include the following; more than one value may be listed.

- *fixed* (only one element is accessible at any given time, as in a stack or queue).
- *fifo* (first-in-first-out policy).
- *lifo* (last-in-first-out).
- *index* (access by an integer index).
- *key* (access by a non-integer key)
- *cursor* (access through a client-controlled cursor, as with the list classes).
- *membership* (availability of a membership test).
- *min, max* (availability of operations to access the minimum or the maximum).

An entry of index *size* indicates a size limitation. Among common values:

- *fixed* means the size of the structure is fixed at creation time and cannot be changed later (there are few such cases in the library).
- *resizable* means that an initial size is chosen but the structure may be resized (possibly at some cost) if it outgrows that size. For extendible structures without size restrictions this entry should not be present.
- An entry of index *representation* indicates a choice of representation. Value *array* indicates representation by contiguous, direct-access memory areas. Value *linked* indicates a linked structure.

- An entry of index *contents* is appropriate for container data structures, used to keep objects. It indicates the nature of the contents. Possible values include *generic* (for generic classes), *integer_c*, *real_c*, *boolean_c*, *character_c* (for classes representing containers of objects of basic types).

The notion of container data structure was presented in =====, page =====, and =====, page =====.

For example, the *ARRAY_LIST* class describes lists implemented by one or more arrays, chained to each other. The clause in this case is:

note

names: block_list;

representation: array, linked; -- In this case it is both!

access: fixed, cursor;

size: resizable;

contents: generic

PART III: KERNEL LIBRARY CLASSES

This third part of the book presents a number of important facilities available to Eiffel developers not through the language but through a few classes known as the Eiffel Library Kernel or ELKS for short, where the S stands for Standard since these classes are standardized (the draft standard is appendix [A](#)).

The Kernel Library comprises the classes, closely related to the language, that are useful to most Eiffel applications. Its role is not to provide a wide-ranging repertoire of algorithms and data structures. The facilities covered by the Kernel Library, detailed in the following chapters, include:

- Universal classes (chapter [35](#)), defining features inherited by all Eiffel classes, from *clone* and *equal* to *print* and *type*.
- Arrays and strings (chapter [36](#))
- Tuples (chapter [13](#))
- Exception handling facilities (chapter [37](#))
- Persistence mechanisms (chapter [37](#))
- Basic types such as *INTEGER* and the like (chapter [30](#)).
- ELKS, the Eiffel Kernel Library Standard (appendix [A](#)).

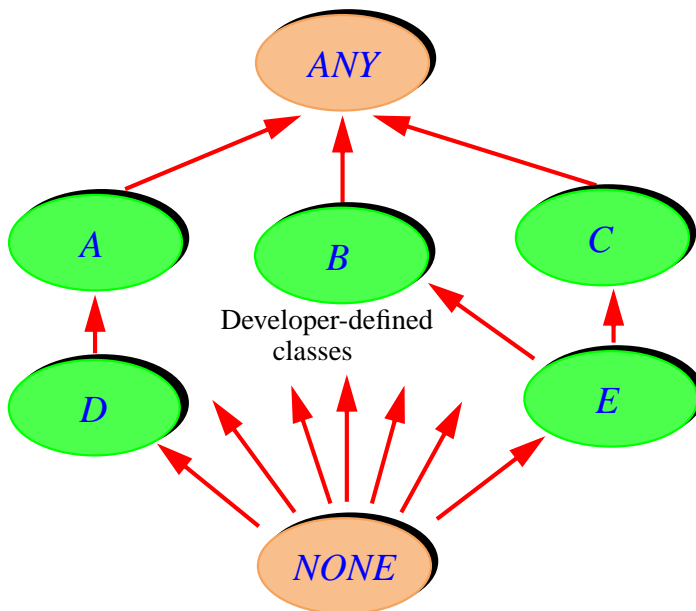
The last of these, appendix [A](#), is different from the other chapters of this book: it is meant to be used not only as part of the book but also as an **independent standard document**. This explains why it uses a legalistic style with limited attempts at pedagogical help for the reader. This should not cause any problem since you can read detailed presentations of the classes in chapters [35](#) to [30](#). You should find appendix [A](#) useful as reference material. Its gist is the formal specification — through flatshort forms — of the ELKS classes.

Universal features and class

ANY (in progress)

35.1 OVERVIEW

Class *ANY* from the Kernel Library is known as a "universal" class since it is an ancestor of any class that you may ever write.



*Universal
Class
Structure*



As you will remember from the [discussion](#) of inheritance, the rule is that any class which does not include its own Inheritance clause is considered to have an implicit clause of the form

← "*ANY*", 6.5, page 172.

`inherit ANY`

appeared on page 85.

Because of this rule, *ANY* serves as both the most general type, to which all types conform, and as the most general set of features, since all developer-defined classes inherit its features.

35.2 INPUT AND OUTPUT FEATURES



This section and the following ones provide an overview of the facilities offered, beginning with input and output facilities. The full flat-short form of the class is given at the end of this chapter.

Feature *io*, of type *STANDARD_FILES*, gives access to standard input and output facilities. It is appropriate for simple input and output operations. For example, *io.input* is the standard input file and *io.new_line* will print a line feed on the standard output.

STANDARD_FILES
and other input-output
facilities are covered in
chapter 38.

You may think of *io* as an attribute, whose value is a standard input-output environment, made available to any class that may need it. As noted, however, the standard %ANY% has no variable attributes to avoid imposing a penalty on every run-time object. Feature *io* is in fact implemented as a once function.

Function *out*, returning a *STRING*, yields a simple external representation of any object. For non-void *x* of any type, the string *out.(x)* is a printable representation of *x*. Because *x* is an argument of the call rather than its target, the function returns a result, an empty string, for void *x*.

Function *out* works for all types: basic types (such as *INTEGER*), reference types, *Class_type_expanded*. For basic types, it gives the standard representation; for example, *x.out*, for integer *x*, is the representation of *x* as a string of decimal digits. For a non-void reference, or a value of *Class_type_expanded*, *x.out* is by default the concatenation of the *out* forms of the object attached to *x*. You may redefine *out* in a class, to produce a specific external representation of the instances of the class.

A variant of *out* is *tagged_out* which, for reference types, normally produces a more readable representation where each field is tagged by the corresponding attribute name. This is only intended as a debugging option; as a consequence, the exact format is not guaranteed, and may under certain language processing tools be reduced to that of *out*.

Procedure *print* is a universal output mechanism. The call *print.(x)* achieves the same as *io.putstring(x.out)*, printing a string representation of *x* on the standard output.

35.3 DUPLICATION AND COMPARISON ROUTINES

A group of routines (*copy*, *clone*, *deep_copy*, *equal* and others) provides facilities for copying, duplicating and comparing objects.

The chapter on object duplication and comparison explored the properties of these routines in detail. *See chapter 21 about object duplication and comparison.*

35.4 OBJECT PROPERTIES

A few features give access to general properties of the current object.

Feature *object_id*, implemented as a function or attribute, yields a *STRING* result which is guaranteed to be a different string for distinct complex objects. The string may be computed from the address of the memory location used to store the object, but this is not part of the feature's specification, which only requires uniqueness of the result for each object. The value of *x.object_id* is only meaningful for *x* of reference type; if *object_id* is used in Unqualified_call form, its value is only meaningful if the target of the current call is non-expanded.

--- TALK HERE ABOUT “type” ---

Function *generator* returns a string, the name of the current object's generating class – that is to say, the base class of the type of which the object is a direct instance. *The notion of generator was defined in 19.2, page =====.*

Sometimes you may need to determine at run time whether the type of a certain object conforms to the type of another. The boolean function *conforms_to* provides a way to do this under the form

x.conforms_to (y)



The discussion of object test introduced two complementary techniques for the same goal: using a succession of assignment attempts, or function *dynamic_type* from the Kernel Library class *INTERNAL*. That discussion also explained in detail why such an operation is seldom required, and seldom appropriate, in the Eiffel method of software development. *See .*

35.5 PLATFORM-DEPENDENT FEATURES

Features introduced in class *PLATFORM* give platform-dependent values:

- The integer constants *Character_bits*, *Integer_bits*, *Real_bits*, *Double_bits* and *Pointer_bits* indicate the number of bits used to represent instances of the basic expanded types *CHARACTER*, *INTEGER*, *REAL* and *POINTER*.
- The integer constant *Maximum_character_code*, which may not be less than 128, gives the highest supported character code. Valid character codes are between 1 and this value.
- For any non-void entity *a*, the value of *a.bit_size* is the number of bits used to store *a*. *See ===== about bit_size.*

35.6 OTHER UNIVERSAL FEATURES

(Needs to define *do_nothing* — see reference in the discussion of delayed calls.)

Arrays and strings *(not done)*

36.1 OVERVIEW

Arrays and strings are homogeneous sequences of values – characters for strings, values conforming to an arbitrary type for arrays – accessible through contiguous integer indices.

The basic operations on arrays and strings are not special language constructs; instead, two Kernel Library classes, *ARRAY* and *STRING*, describe the corresponding objects and provide features for access and modification. Although for most purposes you may use these two classes as any other library classes, one property sets them apart: the language offers notations for **manifest** values of type *STRING* and *ARRAY* *.[T]* (manifest strings of the form "*some text*" and manifest arrays of the form *<<element, element, ...>>*). This means that language processing tools must know about these classes.

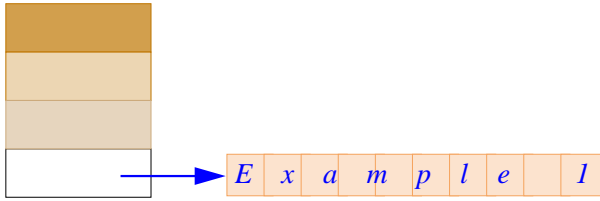
This chapter presents the features of classes *ARRAY* and *STRING* and explains how to manipulate the corresponding objects.

36.2 REPRESENTATION

The two classes use the same representation technique. Both are non-expanded classes, so that the value of an entity of type *STRING* or *ARRAY* *.[T]* for some *T* is a reference to an object. That object is **not**, however, the actual sequence of values (array or string), but a descriptor, which contains information about the sequence and its properties (such as its length and its bounds), and provides access to the sequence itself.

A possible representation is shown below for strings; here the descriptor contains, among other fields, a reference to the sequence of characters, --- - UPDATE --- which is a special object. A special object is not a direct instance of a type but simply a sequence of values used only for implementation purposes, and accessible through a descriptor. For an array the values in the sequence may be of a type other than *CHARACTER*.

s: *STRING*



The representation shown here is an illustration, not necessarily the exact physical form of arrays. In particular:

An implementation may choose to store no other information in the descriptor than the reference to the special object, any other information (the shaded areas on the figure) being kept in the special object itself.

- Some implementation constraints may require an extra level of indirection.
- At the other extreme, a compiler may even be able to avoid any indirection for a certain array (perhaps because it is never resized). This is permitted as long as the semantic effect of operations on arrays is the one described in this chapter.



What you must remember in practice is that an entity of type *STRING* or *ARRAY*.[*T*] does not necessarily give you a direct handle on the actual sequence of values making up a string or array. These representation issues are of little consequence as long as you only access these values through the features of classes *STRING* and *ARRAY*, as given by this chapter in flat-short form. You need to be careful, however, when writing an Assignment or Equality test on such structures, since these are defined as operations on the descriptors rather than the sequences themselves. To obtain operations on the sequences, use the *copy*, *clone* and *equal* routines for which, as explained below, redefinitions in %*STRING*% and *ARRAY* yield the expected semantics.

36.3 RESIZING

As noted at the beginning of this chapter, each element in a string or array has an associated integer index, and the legal indices form a contiguous interval. In practice, this almost universally means that sequence elements are stored contiguously in memory. Contiguous memory storage is not absolute requirement, but the best means known for the only end that really matters to users of arrays, the guarantee that the time needed to access or replace an element known by its index is constant, and small.

The price to pay for this guarantee is the difficulty of changing the index interval – adding elements before, between or after existing ones. If a high percentage of the operations performed on a data structure are such out-of-bounds additions, then array or string is probably not the appropriate representation; many classes of the Data Structure Library provide more flexible data structures, at some cost in access time, storage occupation, or both.

See "Eiffel: The Libraries" about the Data Structure Library.

To make fast access and replacement possible, then, an Eiffel array or string is a bounded data structure, the bounds being defined at any time by the index interval. But bounded does not mean fixed-size. You may resize an array or string, either explicitly (through procedure *resize*, available in both classes) or implicitly, by assigning a value to an array element outside of the current index interval, using a feature which includes a provision for automatic resizing.

As an example of implicit resizing, *STRING* has a feature *append*, which concatenates a copy of a string at the end of another string, with no length restriction; *append* will automatically resize the target string if the operation causes it to grow beyond its originally assigned capacity. Similarly, *ARRAY* has two features for assigning a value to an element given by its index: one, *put*, requires (as part of its precondition) an index that belongs to the current index interval, but the other, *force*, has no such restriction and will automatically resize the array if needed.

In practice, you should keep the following two considerations in mind when using explicit or implicit resizing:

- Resizing is likely to be much more expensive than the basic array and string operations of accessing or replacing an element within the current index interval. In fact the obvious implementation of resizing to a higher size allocates a fresh array or string and copies the old values, which implies an access and replacement operation on *every* previously allocated element. This means that the ratio of resizing operations to non-resizing ones (basic access and replacement) should normally remain low. For highly dynamic data structures, linked representations supported by classes of the Data Structure Library such as *LINKED_LIST* are usually preferable to arrays.

The 'resize' procedures keep existing items, and hence cannot be used to make some of the structure's storage space reclaimable. To shrink an array, use 'remake'. For strings, use 'shrink', with the string itself as first argument. See the class specifications below.



- For arrays, the knowledge that a certain array has a fixed *lower* bound may allow compilers to generate faster code for access and replacement operations, especially if that bound is syntactically a constant (rather than a variable attribute or a local routine entity). A good compiler may be able to detect that a certain array will always keep its original lower bound (because it is never resized, or resizing affects its upper bound only). In this case the penalty for changing the lower bound may be higher than just the cost of resizing since it means that even basic access and replacement, although still constant-time, are less efficient than for an array which has a constant lower bound.

36.4 BASIC ARRAY HANDLING

Class *ARRAY* represents arrays; it is a generic class, with one generic parameter representing the type of the array elements. The arrays it describes are one-dimensional; for multi-dimensional arrays, the Data Structure library offers further classes such as *ARRAY2*, but you may also use *ARRAY* in a nested form, as in *ARRAY [ARRAY [T]]*.

To create an array with bounds *m* and *n* (two integer expressions), that is to say, with an initial index interval consisting of all the values *i* such that $m \sim \leq \sim i$ and $i \sim \leq \sim n$, use the creation procedure *make*, as in

```

up_to_date, delinquent: ARRAY [ACCOUNT];
m, n: INTEGER;
...
m := ...; n := ...;
create up_to_date.make (1, 300);
create delinquent.make (m-1, n+1)

```

The current lower and upper bounds are accessible through features *lower* and *upper*, the number of available elements through *count*. An invariant clause states that *count* is $upper \sim - \sim lower \sim + \sim 1$.

The two basic operations on an array are accessing and replacing an element known by its index:

- Access uses function *item*. To obtain the *i*-th element of *up_to_date*, write the expression

```
up_to_date.item (i).
```

- Replacement uses procedure *put*. To replace by *a* the *i*-th value of *delinquent*, write the instruction

```
delinquent dot put .(a ,, i)
```

Each of these functions has a precondition stating that the index *i* must have a value between the bounds *lower* and *higher*. (This means in particular that by enabling precondition checking on class *ARRAY* you can get an implementation to check the legality of your array accesses.)

On run-time assertion checking see [9.13, page 253](#).

In contrast with *put*, procedure *force* has no such precondition. The call

```
delinquent.force (a, i)
```

will put value *a* at index *i*, even if the value of *i* is not part of the current index interval. This may cause resizing. To request resizing explicitly, you may also use a call of the form

```
delinquent.resize (l, 2k n)
```

The semantics of this call is to add to the index interval all the integer values between the arguments given (here *l* and *2kn*), without losing any previously entered array element. In other words, the call does not necessarily ensure that the new bounds are *l* and *2kn*, but simply that they accommodate these values and everything in-between. (The bounds may even remain unchanged if the existing index interval already includes the argument values.)

The access function *item* has a bracket alias ---- COMPLETE ----- with an infix alias : **infix** "@". This means that you may express *up_to_date.item* (*i*), if you prefer, as

```
up_to_date @ i
```

Since "@", like any free operator, has the highest possible precedence, you must use parentheses if the index is a non-atomic expression, as in

```
up_to_date @ (i - l)
```

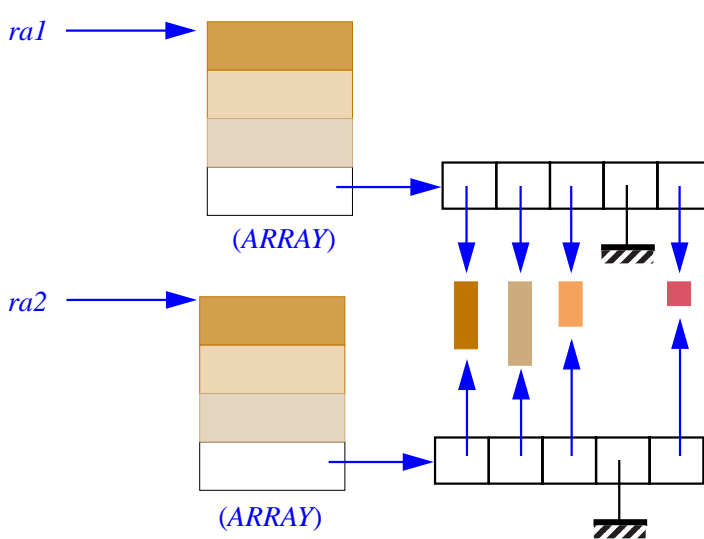
This presence of two equally acceptable names for a single feature is a unique occurrence in the Eiffel libraries – an exception to the style rule which enjoins developers to choose a single name for each feature, and stick to it. The reason for this rare departure from a general guideline is a concession to tradition. Although *item* is the recommended name for the basic access feature for all data structure classes (and is used consistently for this purpose in the Data Structure Library and others), many developers prefer a compact notation for the special and frequent case of array access.

See the table on page =====about operator precedence. The definition of free operators is on page =====.

The style rule was given in the discussion of synonym features: [5.18, page 159](#). See also appendix A about style guidelines. Remember that redefining a feature through one of its name does not redefine the synonym.

36.5 COPYING AND COMPARING ARRAYS

Class *ARRAY* redefines *copy* and *is_equal* from *ANY*, so that for arrays these routines will copy or compare not just the array descriptors but the actual sequences of values.



Array Duplication

Recall that redefining procedure *copy* also implies a new semantics for function *clone*, and that redefining *is_equal* also implies a new semantics for *equal*, which is called under the form *equal* (*a*, *b*) and returns the value of

See [21.4, page 575](#), and [21.7, page 582](#).

(*a* = *Void* and *b* = *Void*) or
 (*a* /= *Void* and *b* /= *Void* and then
a.is_equal (*b*))

For example, if the value of *ra1* is an array as illustrated on the top portion of the figure, an instruction of the form

ra2.copy (*ra1*)

A call to 'copy' requires that its target be non-void.

will assign to *ra2* an identical array, as illustrated by the bottom part. This is still a comparatively shallow copy: if the array elements are references (as on the figure), the references are copied but not the objects to which they are attached (represented by the shaded rectangles).

Function *clone* also duplicates the array, and duplicates the descriptor as well, returning a result attached to the new descriptor.

In contrast with the semantics of *copy* and *clone*, an Assignment on array entities has the semantics of direct reattachment for entities of reference types, which means that

See the table of page 809 about the semantics of direct reattachment.

```
ra2 := ra1
```

will attach *ra2* to the array descriptor attached to *ra1*. Any later array operation on *ra2*, for example the call

```
ra2.put (some_value, some_index)
```

will have the same effect as the corresponding operation of *ra1*. The same behavior results from actual-formal association in a call

```
r (... , ra2, ...)
```

where *ra1* is the corresponding formal argument in %r%.

As with *copy* and *clone*, the version of *equal* for *ARRAY*, using the redefined version of *is_equal*, compares not the descriptors but the actual arrays, recursively applying the appropriate version of *equal* to each successive pair of elements. (In *equal* (*ra1*, *ra2*) where *ra1* and *ra2* are of type *ARRAY* [*T*], the comparison applied to element pairs is the version of *equal* for *T*.)

On the relation between 'equal' and 'is_equal', see 21.6, page 580.

To copy array descriptors rather than arrays, use the routines *standard_clone*, *standard_copy*, *standard_equal*. To obtain deep operations, which will recursively duplicate or compare not just the array values but (if these values are references) the data structures to which they are attached, use *deep_clone*, *deep_copy*, *deep_equal*.

36.6 MANIFEST ARRAYS

You may obtain an array simply by giving its values through a manifest array expression. The expression

29.9, page 809, covered manifest arrays.

```
<< e1, e2, ...en >>
```

denotes an array of *n* elements, the values of expressions *e1* ,, *e2* ,, "..." ~ *en*. If every *ei* conforms to a type *T*, then the manifest array expression conforms to *ARRAY* [*T*]

36.7 STRINGS

Class *STRING* describes character strings.

In principle, a string could be implemented as an object of type *ARRAY* .[*CHARACTER*]. Having a special class makes it possible to use a more compact internal representation and to support many specific string operations (such as appending another string or extracting substrings) which are not as interesting for arbitrary arrays.

To create a string, use the creation procedure *make*:

```
text1.text2: STRING; n: INTEGER
...
create text1.make (n)
```

This will dynamically allocate a string *text1* with room for *n* characters. The argument to *make* gives the initial length of the string. This is **not**, however, a hard-wired limit: if further operations result in *text1* growing beyond *n* characters, the string will automatically be resized.

As with arrays, you can also initialize a string by giving its contents: use a Manifest_string of the form

See [29.8](#) and , starting on page 391, on manifest strings.

```
example: STRING is "This string literal contains 42 characters"
```

Remember, when using an assignment on strings, that it will be an assignment of string descriptors, not a copy. To obtain a fresh copy of a string, use one of the forms

```
text2.copy (text1)
text2 := clone (text1)
```

relying on the redefined version of *copy* (also used by *clone*), which duplicates the character sequence, not just the string descriptor.

See chapter [21](#) about 'copy', 'clone', 'equal' and 'is_equal'.

Similarly, a test of the form

```
if text1 = text2 then ...
```

will compare string descriptors, which is not what you will want most of the time; to compare the actual strings, use

```
if equal (text1, text2) then ...
```

The next section shows the many operations available on strings, such as concatenation, character or substring extraction, comparison. The comparison operations (which have infix aliases "<", "<=", ">=", ">") use lexical ordering, based on numerical character codes (ASCII or an extended version). These functions exist in deferred form in class *COMPARABLE*, a parent of *STRING*.

See [A.6.3, page 977](#), about *COMPARABLE*.

Function *adapt* is useful if you declare a descendant of *STRING*, say *SPECIFIC_STRING*, and want to initialize an entity *s* of type *SPECIFIC_STRING* by giving its characters explicitly. You may not assign a manifest string such as "*Some Text*" to *s* because of the conformance rules, but you may use

```
s := adapt ("Some Text")
```

since the formal argument of *adapt* is of type **like** *Current*; this anchored declaration ensures automatic adaptation to the type of a descendant See [11.10, page 339](#), on anchored declarations.

Persistence *(not done)*

37.1 OVERVIEW

When you execute a system which creates objects, you will sometimes find it necessary to keep some of these objects in secondary storage for later retrieval by the same or another session.

Mechanisms permitting this belong to the libraries and supporting environment rather than to the language. The issue is sufficiently important, however, to justify a presentation in this book. The solutions presented below are not the only possible ones, but they have proved useful in practice.



The material presented in this chapter is not part of the specification of Eiffel.

37.2 CLASSES FOR PERSISTENCE

The relevant facilities come from two classes of the Support Library: *STORABLE* and *ENVIRONMENT*.

STORABLE covers the more elementary situations. It suffices when all that is needed is to store away an entire object structure, starting at a certain object attached to %x%. Then an instruction such as

```
x.store_by_name ("some_file")
```

will produce and write onto the file of name *some_file* an external form of the entire object structure starting at the object attached to *x*. Later, a system may retrieve the structure by executing

```
retrieve_by_name ("some_file")
```

On the Support Library see "Eiffel: The Libraries" (reference in appendix C).

The facilities provided by *ENVIRONMENT* are more elaborate. An instance of this class is a set of objects. If you open an environment, all objects created thereafter belong to that environment, and you may give them keys for individual identification. You can then store an external representation of the environment in a file through a *store* operation, or request that the environment be stored automatically on session termination. The environment can later be retrieved, and its identified objects accessed individually through their keys.

ENVIRONMENT also introduces features for querying the current state of the execution, for example to count the number of instances of a certain type.

This chapter presents classes *STORABLE* and *ENVIRONMENT*.

37.3 OBJECTS AND THEIR DEPENDENTS

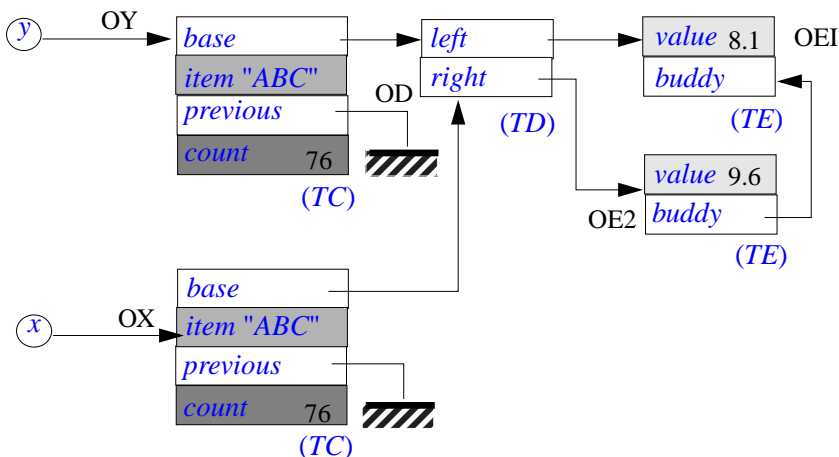
Whether you use *STORABLE* or *ENVIRONMENT*, persistence raises an important practical issue: when an object is stored, what happens to the references it contains?

In a class *C* any attribute declared of a reference type, such as

attrib: SOME_REFERENCE_TYPE

represents a field which, in any run-time instance of *C*, contains a reference to an instance of *SOME_REFERENCE_TYPE* (or a void reference).

Such references may give an object direct and indirect **dependents**. On the above figure, for example, the object marked OX has one direct dependent, OD; the whole set of its dependents, direct or indirect, includes OX itself, OD, OE1 and OE2.



Direct and indirect dependents

This is derived from the figure illustrating object copying on page =====.

More generally:

The **direct dependents** of an object O , at some time during the execution of a system, are the objects attached to the reference fields of O . The **dependents** of an object are the object itself and (recursively) the dependents of its direct dependents.

Whenever an object is stored by one of the operations described in this chapter, its reference fields would become meaningless for later retrieval unless the operation also stores the dependents of the object, direct or indirect. This imposes a universal rule on the routines of both *STORABLE* and *ENVIRONMENT*:

Persistence Completeness rule

Whenever a routine of class *STORABLE* or *ENVIRONMENT* stores an object into an external file, it stores with it the dependents of that object. Whenever one of the associated retrieval routines retrieves a previously stored object, it also retrieves all its dependents.

In other words, whenever you make a certain object persistent, you implicitly make all of its dependents persistent as well.

The persistence facilities properly handle shared references (references to the same object from several sources) and cyclic dependencies (direct or indirect dependencies from an object to itself).

37.4 RETRIEVAL, TYPING, AND THE ASSIGNMENT ATTEMPT

How do persistence facilities combine with static type checking?

During the execution of a given system, every object of interest is accessible through one or more entities such as class attributes and routine arguments. Since the language is typed, the type of any such entity is known in the corresponding class texts from the entity's declaration; it indicates the type of the attached objects, or at least an ancestor of this type, which may be viewed as an approximation of it.

Typing was discussed in chapter 25.

These compulsory declarations, combined with the type rules for polymorphic assignment and feature application, make it possible to guarantee that no operation will be attempted on an object unless it is indeed applicable to all objects of the corresponding type.

When objects are stored away in persistent memory, however, they lose their connection with the text of the software system that created them. If the objects are later retrieved during a session of the same or another system, their internal format is no longer guaranteed to match the declared types of the entities attached to them in the new system.

The problem arises when it comes to declaring a type for library features returning objects from persistent storage: attribute *retrieved* in *STORABLE*, a reference to the retrieved structure, and function *item* in class *ENVIRONMENT*, accessing an object of an environment through its key.

These are features of general-purpose library classes and must be applicable to access retrieved objects of any type. The obvious question is then: under what types should these features be declared in classes *STORABLE* and *ENVIRONMENT*?

The only possible answer is the least committing one: because these two classes must be universally applicable, *item* and *retrieved* can only yield a result of type *ANY* – the most general type, to which all types conform.

If we stopped here, no useful operation would be possible on the retrieved objects, since class *ANY* only provides general-purpose operations applicable to all types – such as copy, clone, equality comparison or output in a default format. But when you store objects of a given type – for example instances of a class *BANK_ACCOUNT* – it is because you expect that at retrieval time you will use them according to this type, applying the corresponding features – such as *deposit*, *withdraw* or *balance*. The features of *ANY* would never suffice.

← “*ANY*”, 6.5, page 172; see also chapter 35 for more details.

Yet you will still want the benefits of type checking: if for some reason a retrieved object is not an instance of *BANK_ACCOUNT*, you cannot accept it blindly and start applying *BANK_ACCOUNT* features to it, with all the possible consequent damages.

The solution, as you will undoubtedly have guessed, is provided by the *Assignment_attempt* instruction. Assume that *her_account* is an entity of type *ACCOUNT*. Then if *retrieved_object* is an expression of type *ANY*, built from features of *STORABLE* or *ENVIRONMENT* and denoting an object retrieved from persistent storage, the instruction

Assignment_attempt was introduced in chapter 22.

her_account ?= *retrieved_object*

makes *her_account* void if *retrieved_object* is void, or attached to an object of a type which does not conform to *her_account*'s declared type, *ACCOUNT*; otherwise, that is to say if the instruction's source *retrieved_object* is attached to an object of the expected type, the instruction also attaches the target *her_account* to the same object.

Here is a more complete form of the example, showing a typical scheme for retrieving persistent objects in a type-safe way.



```

old_accounts: HASH_TABLE [BANK_ACCOUNT, STRING];
her_account: BANK_ACCOUNT;

...

retrieve_by_name ("Old_account_file");
old_accounts ?= retrieved;
if old_accounts+ Void then
    -- File Old_account_file doesn't contain what we
    thought it did!
    ...
else
    -- Deal normally with the hash-table old_accounts:
    her_account := old_accounts.item ("Sarah")
    -- Here item is the search function for hash tables.
    ...
end

```

The use of `Assignment_attempt` achieves the combination of flexibility and safety required to support persistent objects in a statically typed context.

37.5 STORING AND RETRIEVING AN ENTIRE STRUCTURE

The features of class `STORABLE` make it possible to store an object structure consisting of an object and all its dependents, and retrieve it later.

There are two storing procedures, called under the form

```

x.store_by_name ("file_name")
x.store_by_file (file_object)

```

The first expects a file specified by its name (a string), and the second expects an instance of the Support Library class `FILE`.

In all cases, the base class of the type of `x` must be a descendant of class `STORABLE` for these operations to be applicable to `x`. Only `x` must satisfy this constraint: there is no particular requirement on the types of the dependents of the object attached to `x`.

See "Eiffel: The Library-Alternatively, the client relation could be used: if `st` is of type `STORABLE` and has been created, then the forms `st.x.store_by_name` etc. will work.

Retrieval is provided by two procedures corresponding to the two conventions for referring to files:

```
retrieve_by_name ("file_name")
```

```
retrieve_by_file (file_object)
```

In accordance with the methodological advice against functions with side effects, these features are procedures, not functions; they retrieve a structure and make it available through an attribute

See "Object-Oriented Software Construction".

```
retrieved: like Current
```

```
-- Last object retrieved by one of  
the retrieval procedures.
```

37.6 CLASS STORABLE

The precise specification of *STORABLE*'s exported features, as given by

the flat-short form of the class, follows.

-- Facilities for storing and retrieving object structures

-- in binary format Classes needing these facilities

-- should inherit from this class.

class interface *STORABLE* **exported features**

retrieve_by_file (*f*: *FILE*)

-- Retrieve an object structure from external

-- representation previously stored in file *f*.

require

good_file: *f* /= *Void* **and then** *f.exists*

retrieve_by_name (*filename*: *STRING*)

-- Retrieve an object structure from external representation

-- previously stored in file of name *filename*.

require

file_name_not_void: *filename* /= *Void*

retrieved: **like** *Current*

-- Last object retrieved by one of the retrieval procedures

storable_error: *BOOLEAN*

store_by_file (*f*: *FILE*)

-- Produce an external representation of the entire

-- object structure reachable from current object.

-- Write this representation onto file *f*.

require

good_file: *f* /= *Void* **and then** *f.exists*

store_by_name (*filename*: *STRING*)

37.7 ENVIRONMENTS

When the aim is simply to store away a snapshot of the current object structure, or part of it, the facilities of *STORABLE* are sufficient. Environments provide a more flexible and selective approach, with a number of concrete advantages:

- The objects belonging to a stored environment may be individually identified by keys. You may then retrieve them selectively through these keys. (In contrast, the procedures of *STORABLE* produce external structures where only one object, the root, is known individually.)
- An environment may be stored not just through an explicit call to a *store* procedure, but also automatically on session termination, if you request it.
- Environments are normal objects and may be manipulated as such. There is no need to inherit from a special class such as *STORABLE*.
- Environments also provide useful information on the objects of a session, independently from applications to persistence. For example it is possible to query an environment about the number of objects it contains, or the number of objects of a certain type.

You may access an environment through an entity of type *ENVIRONMENT*. For example:

```
env: ENVIRONMENT
```

To create an environment, use the single creation procedure of the class, *make*, which takes no argument:

```
create env.make
```

An environment – instance of class *ENVIRONMENT* – simply represents a set of objects. It is always complete under dependency: in other words, if an object belongs to an environment, all of its dependents, direct or indirect, also belong to the environment.

The various operations on environments, described below, may produce error conditions if the conditions for their application are not met. Class *ENVIRONMENT* contains an integer attribute *error* which every operation sets to the value *No_error* (if all went well) or to one of the error codes described below. The boolean function *ok* has the same result as

```
error = No_error
```

37.8 OPENING AND CLOSING ENVIRONMENTS

Once an environment has been created, you may open it. This means that all objects created from then on (until the environment is closed, or another is opened) will belong to this environment; so will all of their dependents, direct or indirect.

Only one environment may be open at a time; opening an environment closes the previously opened one, if any.

To open or close an environment *env*, use

```
env.open
env.close
```

37.9 RECORDING AND ACCESSING OBJECTS IN AN ENVIRONMENT

You may identify objects individually in an environment through keys. The keys determine what objects will be stored along with the environment, and make it possible to retrieve stored objects individually.

Because the keys are kept along with the objects when the environment is stored, they must be of a type available to all systems. For this reason, keys are restricted to being character strings.

To associate key "*KEY1*" with the object attached to *x* in an environment *env*, use procedure *put*, as in



```
x: SOME_TYPE;
...
create x ... (...);
env.put (x, "KEY1")
```

The first formal argument of procedure *put* is declared of type *ANY*, so that any type will be acceptable for *x*.

After the successful execution of such a call, the object is said to be **recorded** under the given key in the given environment, and the key is said to be **in use** for that environment.

A key may be used for only one object in an environment; if you call *put* with a key that is already in use, the existing object recordings will not be changed and *error* will be set to *Conflict*. To force the new recording (and dissociate the key from any previous object recorded under it), use procedure *force* instead of *put*.

To obtain the object recorded under a certain key, use the function *item*, which returns a result of type *ANY*. As discussed above, you will need an *Assignment_attempt* to access the objects under their true type, as in



```

x: SOME_TYPE;
...
x ?= enn.item ("KEY1");
if x = Void then
... The object recorded under "KEY1" is not of the expected
type
... SOME_TYPE (or the key is not used in the environment)
else
check x / = Void end;
... Here the algorithm may deal with x normally
end

```

To determine whether a key is in use in an environment, call the boolean-valued function *has*, as in



```

key_used := env.has ("KEY1")

```

To change the key under which an object is recorded, use procedure *change_key*, as in

```

env.change_key ('old_key', "new_key")

```

If "*old_key*" was not in use, then *error* will be set to *Not_found*.

37.10 THE OBJECTS OF AN ENVIRONMENT

The preceding discussion yields the definitions of what objects *belong to* an environment, and which ones among these are *persistent*.

One of the main uses of recording objects is indeed to make them persistent in the following sense:



Persistent objects

The **persistent objects** of an environment are all the objects recorded under some key in the environment, and their dependents.

As the name indicates, the objects defined as persistent will be kept by the storing procedures seen in the next section. But the persistent objects of an environment are not the only ones that belong to it. More generally:



Objects belonging to an environment

The objects **belonging** to an environment *env* are defined as follows.

- 1 • Any persistent object of *env* belongs to *env*.
- 2 • Any object created while *env* is open belongs to *env*.
- 3 • Any dependent of an object belonging to *env* belongs to *env*.
- 4 • No object belongs to *env* other than through rules 1, 2 and 3.

37.11 REQUESTING INFORMATION ABOUT ENVIRONMENTS

Function *count* of class *ENVIRONMENT* makes it possible to query an environment about the objects that belong to it, making environments useful even without any application to the storage and retrieval of persistent objects. Function *persistent_count* has the same specification as *count* except that it only takes into account an environment's persistent object.

A call to *count* will yield the number of instances of a certain type belonging to an environment. For example,



```
employee_count := env.count (some_employee)
```

will assign to *employee_count* (assumed to be an integer *Variable*) the number of objects which belong to *env* and conform to the type of the object attached to *some_employee* (or 0 if *some_employee* is void).

The argument of *count* is an expression (here the entity *some_employee*) whose value is used only for the type of the attached object. So if you want the number of instances of the non-generic class *EMPLOYEE* in *env* you can use the above instruction preceded by

```
some_employee: EMPLOYEE;
```

```
...
```

```
create some_employee
```



Recall that **instances** of a reference type include not just direct instances but also instances of any conforming type. So the above will count not just the direct instances of *EMPLOYEE* but also those of any descendants.

As a consequence, you may use *count* to find out about the total number of objects in an environment: just use as argument a direct instance of ANY.

See chapter 19 on instances and direct instances.

Function *count* does not count sub-objects, only "outermost" objects. For example *count* applied to the figure which served to illustrate the notion of sub-object will take only two objects into consideration (O1 and O2).

See the figure page ==, belonging to the discussion of complex objects and their sub-objects in 19.6.

37.12 STORING ENVIRONMENTS

You may store the persistent objects of an environment into a file, and retrieve the objects of a previously stored environment.

For both storage and retrieval, a file must have been associated with the environment; it will be used as target for storage, and as source for retrieval.

Use procedure *set_file* to associate a file with an environment. For example:



```
env.set_file (f)
```

Here *f* must be attached to an instance of class *FILE* from the Kernel Library. There must be a file associated with *f*, and it must have been opened in the appropriate read or write mode.

To store the persistent objects of an environment into the file that has been associated with it, you may use a call of the form:

Class FILE is described in "Eiffel: The Libraries".

```
env.store
```

This is an **explicit** store operation. It is also possible to prescribe an **automatic** store. By executing one or both of the calls

```
env.store_on_end
env.store_on_failure
```

you ensure that session termination (normal termination in the first case, abnormal termination in the second) will automatically result in all the environment's objects being stored in the associated file.

This is a way to guarantee that all the objects of a session, or a selection of these objects (as captured by an environment) will be available for the next session.

In some cases, you may want to use different files for normal and abnormal termination. Only one file may be associated with a given environment at any given time (as the result of the last call to *set_file*). But you may have two environments sharing the same objects. If both *env1* and *env2* are created environments, executing the call

```
env1.share (env2)
```

The effect of env2.share (env1) is identical.

ensures that *env1* and *env2* will contain exactly the same objects under the same keys. The other properties of these environments, such as the associated files, remain separate, so that you may obtain the effect of different normal and abnormal external storage as follows:



```
normal_env, failure_env: ENVIRONMENT;
...
create normal_env.make; create failure_env.make;
failure_env.share (normal_env);
normal_env.set_file (normal_file);
failure_env.set_file (failure_file);
normal_env.store_on_end;
failure_env.store_on_failure;
...
normal_env.open;
...
-- Both environments will contain the same persistent objects.
-- On normal termination, these objects will be stored in
normal_file;
-- On abnormal termination, they will be stored in failure_file.
```

37.13 RETRIEVING AN ENVIRONMENT

To make the persistent objects of a previously stored environment accessible in the same or another session, use procedure *retrieve*. For example:



```
env.retrieve
```

This will load the recorded objects from the file associated with *env*. If this operation does not succeed, *error* will be set to *Not_retrieved*.

Function *item* is then available to retrieve individual objects from the environment, using the keys under which they were recorded. The method was shown above. *See page =====on how to use 'item' to access retrieved objects.*

37.14 AN ENVIRONMENT EXAMPLE

The following example shows a typical use of environments. *C1* and *C2* are arbitrary reference types.

A first session creates a number of objects and records some of them

under some keys, ensuring that they will persist with the environment.



```
env: ENVIRONMENT;  
a: C1;  
b: C2;  
ext_file: FILE;  
-- Create an environment  
create env.make;  
-- Create a file to be associated  
with the environment  
create ext_file.make ("SESSION1");  
ext_file.open_write;  
-- Associate file with environment  
env.set_file (ext_file);  
-- Request that environment be  
automatically stored  
-- on normal termination.  
env.store_on_end;  
-- Request that environment be  
automatically stored  
-- on abnormal termination.  
env.store_on_failure;  
...  
-- Make the objects associated  
with a and b,  
-- as well as any of their  
dependents,  
-- persistent in env, with  
appropriate keys.  
env.put (a, "a_key");  
env.put (b, "b_key");  
-- The following instructions may  
of course modify  
-- the objects associated with a, b  
and  
-- their dependents  
...
```

... The environment may be stored explicitly through
 ... *env.store*
 ... If not, it will be stored automatically on session termination.

The same session (if an explicit *store* has been executed) or another (executed any time later) may now retrieve the stored objects:

```
env: ENVIRONMENT;
a: C1;
b: C2;
ext_file: FILE;
create env.make;
create ext_file.make ("SESSION1");
ext_file.open_read;
env.set_file (ext_file);
  -- Load the previously stored environment.
env.retrieve;
  -- Individual objects may now be accessed through their keys.
  -- Their dependents, if any, have also been loaded.
a ?= env.item ("a_key");
b ?= env.item ("b_key")
...

```

37.15 CLASS *ENVIRONMENT*

The precise specification of *ENVIRONMENT*'s exported features, as given by the flat-short form of the class, follows.

```
class interface ENVIRONMENT
creation procedures
make
  -- Create a new environment.
```

exported features

change_key (*old_key*, *new_key*:
STRING)

-- Record under *new_key* the
object previously

-- recorded under *old_key*; if no
such object, set

-- *error* to *Not_found*.

require

keys_not_void: *old_key* \= *Void*
and *new_key* /= *Void*

ensure

ok **or** (*error* = *Not_found*) **or**
(*error* = *conflict*)

close

-- Make current environment be no
longer active; if

-- it was.

ensure

closed: **not** *is_open*

Conflict: *INTEGER* is 1

count (*obj*: *ANY*): *INTEGER*

-- Number of objects in the
environment whose type conforms

-- to the type of the object attached
to *obj*.

-- 0 if the value of *obj* is void.

current_keys: *ARRAY* [*STRING*]

-- Array of keys in use, starting
from 1

error: *INTEGER*

-- Code of last error produced by a
routine of the class

force (*obj*: *ANY*; *key*: *STRING*)

-- Record *obj* under *key* in this
environment:

-- Make *obj* and its dependents
persistent.

-- If *key* is already in use, lose the

has (*key*: *STRING*): *BOOLEAN*

-- Is *key* in use?

require

key_not_void: *key* /= *Void*

is_open: *BOOLEAN*

-- Is current environment open?

item (*key*: *STRING*): *ANY*

-- Object recorded under *key*.

-- If no such *key*, void value and *error* set to *Not_found*.

require

key_not_void: *key* /= *Void*

ensure

(*Result* = *Void*) **implies** (**not** *has* (*key*))

No_error: *INTEGER is 2*

no_store_on_end

-- Disable automatic storage on normal termination.

-- This is the default.

no_store_on_failure

-- Disable automatic storage on abnormal termination.

-- This is the default.

Not_found: *INTEGER is 3*

Not_retrieved: *INTEGER is 4*

Not_stored: *INTEGER is 5*

ok: *BOOLEAN*

-- Did last retrieve, store, put or remove operation succeed?

ensure

Result = (*error* = *no_error*)

open

-- Close previous environment if any, and make current

-- environment the active one:

-- all objects created from now on,

ensure*open: is_open**persistent_count* (*t: STRING*):
INTEGER

-- Number of persistent objects in the environment whose type conforms

-- to the type of the object attached to *obj*.

-- 0 if the value of *obj* is void

put (obj: ANY; key: STRING)

-- Record *obj* under *key* in this environment:

-- Make *obj* and its dependents persistent.

-- If *key* is already in use, set *error* to *Conflict*.

require*key_not_void: key /= Void***ensure***ok or (error = conflict)**remove (key: STRING)*

-- Dissociate *key* from object recorded under.

-- If no such object, set error to *Not_found*.

require*key_not_void: key /= Void***ensure***ok or (error = Not_found)*

retrieve

-- Load the environment's persistent objects

-- from the associated external file.

ensure*ok or (error = Not_retrieved)**set_file (f: FILE)*

-- Make *f* the file where environment will be stored

```
share (other: like Current)
    -- Make Current and other share
    the same persistent objects.
store
    -- Store the environment's
    persistent objects
    -- into the associated external file.
ensure
    ok or (error = Not_stored)
store_on_end
    -- Enable automatic storage on
    normal termination.
    -- This is not the default.
store_on_failure
    -- Enable automatic storage on
    abnormal termination.
    -- This is not the default.
end interface    -- class
ENVIRONMENT
```

Input and output *(not done)*

38.1 OVERVIEW

Class *STANDARD_FILES* from the Kernel Library offers a set of simple but useful input and output facilities.

Another class, *FILE*, provides much more extensive file handling operations, and the original implementation of *STANDARD_FILES* relied on *FILE*. By its very nature, however, *FILE* depends on the operating system, and its original version is closely patterned after Unix file handling mechanisms. For this reason, no further description of *FILE* appears in this book; "Eiffel: The Libraries" presents *FILE* in detail.

See the bibliography of appendix C for the exact reference to "Eiffel: The Libraries".

This chapter explains how to perform simple input and output using the facilities of *ANY* and *STANDARD_FILES*.

38.2 PURPOSE OF THE CLASS

If all you need is to print a value using the standard output format, you will not even need *STANDARD_FILES*: procedure *print* from class *ANY*, automatically present in every class, provides this facility; for example:

See 35.2, page 928, about 'print' and 'out'.



```
print ("Today's temperature is");
print (temperature);
print ("%N")
```

%N is the new-line character. See [32.14, page 894](#), particularly the table of special characters on page 423.



A call *print* (*x*) outputs on the standard output file the value of *out* (*x*), where function *out* yields a printable version of any object. You may redefine this function in a class to yield a specific form of output.

Class *STANDARD_FILES* provides some further output mechanisms, still elementary but more varied, as well as basic input features.

The most common way to use the facilities of *STANDARD_FILES* is through feature *io*, present in class *ANY* and hence in all developer-defined classes (unless you remove it explicitly). Feature *io* is a once function of type *STANDARD_FILES*. Any class may perform simple input and output by calling *STANDARD_FILES* features on *io*, as in the following variant of the above extract:



```
io.putstring ("Today's temperature is");
io.putstring (out (temperature));
io.new_line
```

Among the features of *STANDARD_FILES* are output procedures such as *putstring*, *putint* and *putreal*, which apply to the standard output, and input features such as *readint* and *lastint*, used according to the conventions explained in the next section and applying to the standard input.

The class also offers features *input*, *output* and *error*, all of type *FILE*, giving access to the standard input, standard output and standard error files. These features are implemented as "once" functions; the first call to any of them opens the corresponding files.



Because of the presence of feature *io* in class *ANY*, any class *C* is a direct client of *STANDARD_FILES*. You may prefer to avoid dot notation for calls to input or output features, as in *io.some_feature*, by making *C* an heir to *STANDARD_FILES*; this enables you to write such calls as just *some_feature*. Some object-oriented purists shun such uses of inheritance, but it's really no more than a matter of taste.

38.3 INPUT TECHNIQUES



The input features of *STANDARD_FILES* observe an important style guideline of the Eiffel method: avoiding functions with side effects. This means that to read an input element you must usually execute two calls:

- A procedure call, such as *io.read_integer* or *io.read_real*, to advance the input cursor past the element.
- A call to a function or attribute, as in *n := io.lastint r n := io.last_real*, to return the value of the element read, with a result of the appropriate type (*INTEGER* and *REAL* in the examples).

Successive calls to a feature such as *io.last_integer* or *io.last_real* will yield the same value if they are not separated by calls to cursor-advancing procedures.

See "Object-Oriented Software Construction" on side effects in functions.

The most obvious implementation of STANDARD_FILES uses attributes rather than functions for 'lastint' and its acolytes ('lastreal', 'last_real', 'laststring', 'lastchar').

To use procedures such as *read_integer* and *read_real*, you must know in advance the types of the input elements to be read. In some cases, of course, you do not have this information. Other mechanisms are available for reading elements and determining their types on the fly; they are not part of the Kernel Library, however, but are provided by the lexical analysis classes of the Lexical Library.

See "Eiffel: The Libraries" about the Lexical Library.

38.4 CLASS *STANDARD_FILES*

Here is the flat-short form of *STANDARD_FILES*.

```
-- Standard input and output.  
class interface STANDARD_FILES  
exported features  
error: FILE  
    -- Standard error file  
input: FILE  
    -- Standard input file  
lastchar: CHARACTER  
    -- Last character read by readchar  
lastint: INTEGER  
    -- Last integer read by readint  
lastreal: REAL  
    -- Last real read by readreal  
laststring: STRING  
    -- Last string read by readstring  
new_line  
    -- Write line feed at end of default  
    output.  
next_line  
    -- Move to next input line on  
    standard input.  
output: FILE  
    -- Standard output file  
putbool (b: BOOLEAN)  
    -- Write b at end of default output.  
putchar (c: CHARACTER)  
    -- Write c at end of default output.  
putint (i: INTEGER)  
    -- Write i at end of default output.
```

```
putreal (r: REAL)
    -- Write r at end of default output.
putstring (s: STRING)
    -- Write s at end of default output.
readchar
    -- Read a new character from
    standard input.
readint
    -- Read a new integer from
    standard input.
readline
    -- Read a line from standard input.
readreal
    -- Read a new real from standard
    input.
readstring (nb_char: INTEGER)
    -- Read a string of at most nb_char
    bound
    -- characters from standard input.
readword
    -- Read a new word from standard
    input.
set_error_default
    -- Use standard error as default
    output.
set_output_default
    -- Use standard output as default
    output.
end    interface    --    class
STANDARD_FILES
```


A

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

ELKS: The Eiffel Library Kernel Standard

A.1 OVERVIEW

[This Overview is not part of the Standard.]

A.1.1 Purpose

To favor the interoperability between implementations of Eiffel, it is necessary, along with a precise definition of the language, to have a well-defined set of libraries covering needs that are likely to arise in most applications. This library is known as the Kernel Library.

A.1.2 Application

The present document defines a standard for the Kernel Library. If an Eiffel implementation satisfies this Standard — under the precise definition of *Kernel Compatibility* given in section [A.3.2](#) — it will be able to handle properly any Eiffel system whose use of the Kernel Library only assumes the library properties defined in this Standard.

A.1.3 Process

The Eiffel Library standardization process is based on a dynamic view which, in the spirit of Eiffel's own "feature obsolescence" mechanism, recognizes the need to support evolution while preserving the technology investment of Eiffel users. One of the consequences of this dynamic view is to define *vintages* corresponding to successive improvements of the Standard. The present document describes **Vintage 2005**, valid for the calendar years 2005-2006.

A.1.4 Copyright status

This Standard is appendix [A](#) of the book *Eiffel: The Language* by Bertrand Meyer (Prentice Hall, 2002) and the copyright belongs to the author. Electronic or paper reproduction of this Standard is permitted provided the reproduction includes the **entire** text of the Standard, including the present copyright notice and the mention that the latest version, up-to-date with any error corrections, may be found at <http://eiffel.com>.

A.2 CONTENTS OF THIS STANDARD

A.2.1 Definition: this Standard

The Eiffel Kernel Library Standard, denoted in the present document by the phrase "this Standard", is made up of the contents of sections [A.2](#) to [A.6](#) of the present appendix, with the exception of elements appearing in black between square brackets [...] which are comments.

[Section [A.1](#), and elements playing a pure typesetting role such as page headers, are not part of this Standard.]

A.2.2 Scope of this Standard

This Standard defines a number of library-related conditions that an Eiffel implementation must satisfy. These conditions affect a set of classes known as the kernel library. An implementation that satisfies the conditions described in this Standard will be said to be **kernel-compatible**, a phrase that is abbreviated in this Standard as just "compatible".

[In other contexts it may be preferable to use the full phrase, since the compatibility of an Eiffel implementation also involves other aspects, such as language compatibility.]

[The terms "compatibility" and "compatible" may be felt to be less clear than "conformance" and "conformant". The former are used here, however, since talking about conformance might cause confusions with the Eiffel notion of a type conforming to another.]

A.2.3 Other documents

The phrase *Eiffel: The Language* as used in this Standard refers to the third edition of the book *Eiffel: The Language*, Prentice Hall, 2000, ISBN 0-13-xxx-xxx-x.

For the purposes of this Standard, the definition of the Eiffel language is the definition given by *Eiffel: The Language*.

In case of contradictions between the library specifications given in this Standard and those of the other chapters of *Eiffel: The Language*, this Standard shall take precedence.

A.3 COMPATIBILITY CONDITIONS

A.3.1 Definitions

A.3.1.1 Required Classes

In this Standard, the phrase “Required Classes” denotes a set of classes whose names are those listed in section [A.4](#).

A.3.1.2 Required Flatshort Form

In this Standard, the phrase “Required Flatshort Forms” denotes the flatshort forms given for the Required Classes in section [A.4](#).

A.3.1.3 Flatshort Compatibility

In this Standard, a class is said to be Flatshort-Compatible with one of the short forms given in this Standard if it satisfies the conditions given in section [A.3](#) of this Standard.

A.3.1.4 Required Ancestry Links

In this Standard, the expression “Required Ancestry Links” denotes the inheritance links specified in section [A.5](#) of this Standard.

[The term “Ancestry” is used rather than “Inheritance” because the required links may be implemented by indirect rather than direct inheritance.]

A.3.2 Kernel compatibility

An Eiffel implementation will be said to be kernel-compatible if and only if it includes a set of classes satisfying the following five conditions:

A.3.2.1 • For each of the Required Classes, the implementation includes a class with the same name.

A.3.2.1.1 • All the Required Ancestry Links are present between these classes.

A.3.2.1.2 • The flatshort form of each one of these classes is Flatshort-Compatible with the corresponding Required Flatshort Form.

A.3.2.1.3 • All the dependents of the Required Classes in the implementation are also included in the implementation.

A.3.2.1.4 • None of the features appearing in the Required Flatshort Forms appears in a **Rename** clause of any of the implementation’s Required Classes.

[These conditions allow a kernel-compatible implementation to include inheritance links other than the ones described in this Standard; condition [A.3.2.1.3](#) indicates that for any such link the additional proper ancestors must also be provided by the implementors, since the dependents of a class include its parents.]

[Condition [A.3.2.1.3](#) guarantees that if a feature name appears in this Standard both in the Flatshort form of a Required Class and in the flatshort form of one of its proper ancestors, it corresponds to the same feature or to a redefinition of it.]

A.3.3 Flatshort Conventions

A.3.3.1 Definition

In the process of assessing for Flatshort Compatibility a class *C* from a candidate implementation, the following ten conventions, which have been applied to the Required Flatshort Forms as they appear in this Standard, shall be applied:

A.3.3.1.1 • No feature shall be included unless it is generally available (as defined in *Eiffel: The Language*, page [211](#)) or is a general creation procedure (as defined in *Eiffel: The Language*, page [550](#)).

A.3.3.1.2 • The **Creation** clause of the flatshort specification shall include the full specification of all general creation procedures of *C*.

A.3.3.1.3 • Any feature of *C* not inherited from *ANY* shall be included in one of the **Feature** clauses.

[As a consequence of the last two rules the specification of a creation procedure that is also generally exported will appear twice: in the **Creation** clause and in a **Feature** clause. Also note that the “features of a class” include inherited as well as immediate features, so that all features inherited from an ancestor other than *ANY* must appear in the flatshort form.]

A.3.3.1.4 • A feature *f* from *ANY* shall be included if and only if *C* redeclares *f*.

A.3.3.1.5 • The header comment of any inherited feature coming from a Required Class *A* and having the same name in *C* as in *A* shall end with a line of the form:

-- (From *A*.)

A.3.3.1.6 • The header comment of any inherited feature coming from a Required Class *A* and having a name in *C* different from its name *x* in *A* shall end with a line of the form:

-- (From *x* in *A*.)

[The comments defined in the last two rules are applicable whether or not *C* redeclares the feature.]

A.3.3.1.7 • If deferred, *C* shall appear as **deferred class**.

A.3.3.1.8 • Any deferred feature of *C* shall be marked as **deferred**.

A.3.3.1.9 • In case of precondition redeclaration, the successive preconditions shall appear as a single **Precondition** clause, separated by semicolons.

A.3.3.1.10 • In case of postcondition redeclaration, the successive preconditions shall appear as a single **Postcondition** clause, separated by **and then**.

A.3.4 Flatshort Compatibility

A.3.4.1 Definition

A class appearing in an Eiffel implementation is said to be Flatshort-Compatible with a class of the same name listed in this Standard if and only if any difference that may exist between its flatshort form *ic* and the flatshort form *sc* of the corresponding class as it appears in section [A.6](#), where both flatshort forms follow the conventions of section [A.3.3](#), belongs to one of the following eleven categories:

A.3.4.1.1 • A feature that appears in *ic* but not in *sc*, whose **Header_comment** includes, as its last line, the mention:

-- (Feature not in Kernel Library Standard.)

A.3.4.1.2 • An invariant clause that appears in *ic* but not in *sc*.

A.3.4.1.3 • For a feature that appears in both *ic* and *sc*, a postcondition clause that appears in *ic* but not in *sc*.

A.3.4.1.4 • For a feature that appears in both *ic* and *sc*, a precondition in *sc* that implies the precondition in *ic*, where the implication is readily provable using rules of mathematical logic.

A.3.4.1.5 • For a feature that appears in both *ic* and *sc*, a postcondition or invariant clause in *ic* that implies the corresponding clause in *sc*, where the implication is readily provable using rules of mathematical logic.

A.3.4.1.6 • A difference between the **Tag_mark** of an **Assertion_clause** in *ic* and its counterpart in *sc*.

A.3.4.1.7 • For a feature that appears in both *ic* and *sc*, an argument type in *sc* that is different from the corresponding type in *ic* but conforms to it.

A.3.4.1.8 • For a feature that appears in both *ic* and *sc*, a result type in *ic* that is different from the corresponding type in *sc* but conforms to it.

A.3.4.1.9 • For a feature that appears in both *ic* and *sc*, a line that appears in the **Header_comment** of *ic* but not in that of *sc*.

A.3.4.1.10 • A **Note_entry** that appears in *ic* but not in *sc*.

A.3.4.1.11 • A difference regarding the order in which a feature appears in *ic* and *sc*, the **Feature_clause** to which it belongs, the **Header_comment** of such a **Feature_clause**, or the presence in *ic* of a **Feature_clause** that has no counterpart in *sc*.

[As a consequence of section [A.3.4.1.11](#), the division of classes into one **Feature_clause** or more, and the labels of these clauses, appear in this document for the sole purpose of readability and ease of reference, but are not part of this Standard.]

[The goal pursued by the preceding definition is to make sure that an Eiffel system that follows this Standard will be correctly processed by any compatible implementation, without limiting the implementors' freedom to provide more ambitious facilities.]

A.4 REQUIRED CLASSES

The Required Classes are the following thirty classes [ordered from the general to the specific, as in section [A.6](#)]:

A.4.1 • **ANY** [flatshort form in section [A.6.1](#)].

A.4.2 • **TYPE** [flatshort form in section [A.6.2](#)].

A.4.3 • **PART_COMPARABLE** [flatshort form in section [A.6.3](#)].

A.4.4 • **COMPARABLE** [flatshort form in section [A.6.4](#)].

A.4.5 • **HASHABLE** [flatshort form in section [A.6.5](#)].

A.4.6 • **NUMERIC** [flatshort form in section [A.6.6](#)].

A.4.7 • **INTERVAL** [flatshort form in section [A.6.7](#)].

A.4.8 • **BOOLEAN** [flatshort form in section].

A.4.9 • **CHARACTER** [flat short form in section [A.6.9](#)].

A.4.10 • **INTEGER_GENERAL** [flatshort form in [A.6.10](#)].

A.4.11 • **INTEGER** [flatshort form in section [A.6.11](#)].

A.4.12 • **INTEGER_8** [flatshort form in section [A.6.12](#)].

A.4.13 • **INTEGER_16** [flatshort form in section [A.6.13](#)].

A.4.14 • **INTEGER_64** [flatshort form in section [A.6.14](#)].

A.4.15 • **REAL_GENERAL** [flatshort form in [A.6.15](#)].

A.4.16 • **REAL** [flatshort form in section [A.6.16](#)].

A.4.17 • **POINTER** [flatshort form in section [A.6.18](#)].

A.4.18 • **ARRAY** [flatshort form in section [A.6.19](#)].

A.4.19 • **ANONYMOUS** [flatshort form in section [A.6.20](#)].

A.4.20 • **STRING** [flatshort form in section [A.6.21](#)].

A.4.21 • **STD_FILES** [flatshort form in section [A.6.22](#)].

A.4.22 • **FILE** [flatshort form in section [A.6.23](#)].

- A.4.23 • *STORABLE* [flatshort form in section [A.6.24](#)].
- A.4.24 • *MEMORY* [flatshort form in section [A.6.25](#)].
- A.4.25 • *EXCEPTIONS* [flatshort form in section [A.6.26](#)].
- A.4.26 • *ARGUMENTS* [flatshort form in section [A.6.27](#)].
- A.4.27 • *PLATFORM* [flatshort form in section [A.6.28](#)].
- A.4.28 • *ONCE_MANAGER* [flatshort form in section [A.6.29](#)].
- A.4.29 • *ROUTINE* [flatshort form in section [A.6.30](#)].
- A.4.30 • *PROCEDURE* [flatshort form in section [A.6.31](#)].
- A.4.31 • *FUNCTION* [flatshort form in section [A.6.32](#)].
- A.4.32 • *PREDICATE* [flatshort form in section [A.6.33](#)].

[The classes appear in this section and section [A.6](#) in the following order: universal classes; deferred classes for basic classes; basic types; arrays and strings; agent and introspection.]

A.5 REQUIRED ANCESTRY LINKS

The following constitute the required ancestry links [ordered alphabetically, after the first rule, by the name of the applicable descendant class]:

- A.5.1 • Every Required Class is a descendant of *ANY*.
- A.5.2 • *COMPARABLE* is a proper descendant of *PART_COMPARABLE*.
- A.5.3 • *TYPE* is a proper descendant of *PART_COMPARABLE*.
- A.5.4 • *BOOLEAN* is a proper descendant of *HASHABLE*.
- A.5.5 • *CHARACTER* is a proper descendant of *COMPARABLE*.
- A.5.6 • *CHARACTER* is a proper descendant of *HASHABLE*.
- A.5.7 • *FILE* is a proper descendant of *MEMORY*.
- A.5.8 • *FUNCTION* [*BASE*, *OPEN_ARGS*, *RESULT_TYPE*] is a proper descendant of *ROUTINE* [*BASE*, *OPEN_ARGS*].
- A.5.9 • *INTEGER* is a proper descendant of *INTEGER_GENERAL*.
- A.5.10 • *INTEGER_8* is a proper descendant of *INTEGER_GENERAL*.
- A.5.11 • *INTEGER_16* is a proper descendant of *INTEGER_GENERAL*.
- A.5.12 • *INTEGER_64* is a proper descendant of *INTEGER_GENERAL*.

A.5.13 • *INTEGER_GENERAL* is a proper descendant of *COMPARABLE*.

A.5.14 • *INTEGER_GENERAL* is a proper descendant of *HASHABLE*.

A.5.15 • *INTEGER_GENERAL* is a proper descendant of *NUMERIC*.

A.5.16 • *POINTER* is a proper descendant of *HASHABLE*.

A.5.17 • *PREDICATE* [*BASE*, *OPEN_ARGS*] is a proper descendant of *FUNCTION* [*BASE*, *OPEN_ARGS*, *BOOLEAN*].

A.5.18 • *PROCEDURE* [*BASE*, *OPEN_ARGS*] is a proper descendant of *ROUTINE* [*BASE*, *OPEN_ARGS*].

A.5.19 • *REAL_GENERAL* is a proper descendant of *COMPARABLE*.

A.5.20 • *REAL_GENERAL* is a proper descendant of *HASHABLE*.

A.5.21 • *REAL_GENERAL* is a proper descendant of *COMPARABLE*.

A.5.22 • *REAL* is a proper descendant of *REAL_GENERAL*.

A.5.23 • *STRING* is a proper descendant of *COMPARABLE*.

A.5.24 • *STRING* is a proper descendant of *HASHABLE*.

A.5.25 • *STRING* is a proper descendant of *HASHABLE*.

A.5.26 • *STRING* is a proper descendant of *HASHABLE*.

["Proper descendant" is a transitive relation, so that for example *INTEGER_8* is a descendant of *COMPARABLE* as a result of [A.5.10](#) and [A.5.13](#).]

A.6 SHORT FORMS OF REQUIRED CLASSES

The following pages (sections [A.6.1](#) to [A.6.33](#)) contain the short forms of the required classes as defined in preceding sections.

A.6.1 CLASS ANY

note

description: "[
Platform-independent universal properties. This
class is an ancestor to all developer-written classes.
]"

class interface

ANY

feature -- Access

type: TYPE [**like** Current]
-- Generating type of current object
-- (type of which it is a direct instance)

onces: ONCE_MANAGER
-- Handle on the state of the system's once routines

feature -- Comparison

is_equal (other: **like** Current): BOOLEAN
-- Is other attached to an object considered equal
-- to current object?
-- The object comparison operator ~ relies on this function.

ensure

same_type: Result **implies** same_type (other)
symmetric: Result = other.is_equal (Current)
consistent: default_is_equal (other) **implies** Result

frozen default_is_equal (other: ? **like** Current):
BOOLEAN

-- Is other attached to an object of the same type as
-- current object, and field-by-field identical to it?

ensure

only_if_same_type: Result **implies** same_type (other)
symmetric: Result **implies** other.default_is_equal
(Current)
consistent: Result **implies** is_equal (other)

frozen is_deep_equal (other: ANY): BOOLEAN

-- Are some and other attached to isomorphic
-- structures made of objects considered equal?

ensure

shallow_implies_deep: is_equal (other) **implies**
Result
same_type: Result **implies** some.same_type
(other)
symmetric: Result **implies** deep_equal (other,
some)

frozen default_is_deep_equal (other: ? ANY):
BOOLEAN

-- Are some and other attached to isomorphic
-- structures made of field-by-field equal objects?

ensure

shallow_implies_deep: default_is_equal (other)
implies Result
only_if_same_type: Result **implies** same_type (other)
symmetric: Result **implies** other.is_deep_equal
(Current)

feature {NONE} -- Duplication

frozen cloned: **like** Current
-- New object equal to current one.

ensure

equal: Result ~ Current)

copy (other: **like** Current)

-- Update current object using fields of object
-- attached to other, so as to yield equal objects.

ensure

equal: Current ~ other

frozen default_cloned: **like** Current

-- New object field-by-field identical to current object

ensure

identical_result: default_is_equal (Result)

frozen default_copy (other: **like** Current)

-- Copy every field of other onto corresponding field
-- of current object.

require

type_identity: same_type (other)

ensure

made_identical: default_is_equal (other)

frozen deep_cloned: **like** Current

-- New object structure recursively duplicated from
-- current object

ensure

deep_equal: deep_is_equal (Result)

feature -- Basic operations

default_rescue

-- Handle exception if no Rescue clause.
-- (Default: do nothing.)

frozen do_nothing

-- Execute a null action.

feature -- Output

io: STD_FILES

-- Handle to standard file setup

out: STRING

-- New string containing terse printable
-- representation of current object

invariant

reflexive_default_equality: default_is_equal (Current)

reflexive_equality: Current ~ Current

end

A.6.2 CLASS TYPE

note

description: "[
 Objects describing types conforming to *G*.
]"

class interface

TYPE [*G*]

feature -- Access

adapted alias "[*x*]" (*x*: *G*): *G*
 -- Value of *x*, adapted if necessary to type *G*
 -- through conformance or conversion

ensure

consistent: *Result* = *x*

class_name: *STRING*

-- Human-readable form of name of base class
 -- (newly created result for every call)

default: *G*

-- Default value of this type

ensure

consistent: *Result.type* ~ *Current*

hash_code: *INTEGER*

-- Hash code value

ensure

good_hash_value: *Result* >= 0

name: *STRING*

-- Human-readable form of this type's name
 -- (newly created result for every call)

up_to alias ".." (*other*: *TYPE* [*ANY*]):

INTERVAL [*TYPE* [*ANY*]]

-- Interval containing all types *t* in system such that
 -- *Current* <= *t* and *t* <= *other*

feature -- Comparison

conforms_to alias "<" (*other*: *TYPE* [*ANY*]):

BOOLEAN

-- Does current type conform to *other*?

is_equal (*other*: *TYPE* [*ANY*]): *BOOLEAN*

-- Is current type identical to *other*?
 -- The object comparison operator ~ relies on this function.

ensure

conformance_both_ways:

Result = *conforms_to* (*other*) **and**
other.conforms_to (*Current*)

yes_if_both_empty_regardless_of_bounds:

is_empty **and** *other.is_empty* **imply** *Result*

end

A.6.3 CLASS *PART_COMPARABLE***note**

description: "[
 Objects that may be compared according to a partial
 order relation
]"

math: "The model is a partial order relation."

comment: [
 "The basic operation is "<" (less than); others are
 defined in terms of this operation and *is_equal*.
]"

deferred class interface

PART_COMPARABLE

feature -- Access

up_to **alias** ".." (*other*: *PART_COMPARABLE*):
INTERVAL [*PART_COMPARABLE*]
 -- Interval containing all values *t*, if any, such that
 -- *Current* <= *t* and *t* <= *other*

feature -- Comparison

is_comparable " (*other*: **like** *Current*): *BOOLEAN*
 -- Do current object and *other* figure in the relation?

deferred**ensure**

definition: *Result* = (*Current* < *other*) **or**
 (*Current* ~ *other*) **or** (*Current* > *other*)
 symmetric: *Result* = *other.is_comparable* (*Current*)

is_less **alias** "<" (*other*: **like** *Current*): *BOOLEAN*
 -- Is current object less than *other*?

deferred**ensure**

asymmetric: *Result* **implies not** (*other* < *Current*)
 only_if_comparable: *Result* **implies** *is_comparable*
 (*other*)

is_less_equal **alias** "<=" (*other*: **like** *Current*):
BOOLEAN

-- Is current object less than or equal to *other*?

ensure

definition: *Result* = (*Current* < *other*) **or**
 (*Current* ~ *other*)
 only_if_comparable: *Result* **implies** *is_comparable*
 (*other*)

is_greater_equal **alias** ">=" (*other*: **like** *Current*):
BOOLEAN

-- Is current object greater than or equal to *other*?

ensure

definition: *Result* = (*other* <= *Current*)

is_greater **alias** ">" (*other*: **like** *Current*): *BOOLEAN*
 -- Is current object greater than *other*?

ensure

definition: *Result* = (*other* < *Current*)
 only_if_comparable: *Result* **implies** *is_comparable*
 (*other*)

is_equal (*other*: **like** *Current*): *BOOLEAN*

-- Is *other* attached to an object considered equal
 -- to current object?
 -- The object comparison operator ~ relies on this function.

ensure

symmetric: *Result* **implies** *other.is_equal* (*Current*)
 consistent: *default_is_equal* (*other*) **implies** *Result*

max (*other*: **like** *Current*): **like** *Current*
 -- The greater of current object and *other*

require

comparable: *is_comparable* (*other*)

ensure

current_if_not_smaller: (*Current* >= *other*) **implies**
 (*Result* = *Current*)
other_if_smaller: (*Current* < *other*) **implies** (*Result*
 = *other*)

min (*other*: **like** *Current*): **like** *Current*
 -- The smaller of current object and *other*

require

comparable: *is_comparable* (*other*)

ensure

current_if_not_greater: (*Current* <= *other*) **implies**
 (*Result* = *Current*)
other_if_greater: (*Current* > *other*) **implies** (*Result*
 = *other*)

three_way_comparison (*other*: **like** *Current*): *INTEGER*
 -- If current object equal to *other*, 0;
 -- if smaller, -1; if greater, 1.

require

comparable: *is_comparable* (*other*)

ensure

equal_zero: (*Result* = 0) = (*Current* ~ *other*)
smaller_negative: (*Result* = -1) = (*Current* < *other*)
greater_positive: (*Result* = 1) = (*Current* > *other*)

invariant

irreflexive_comparison: **not** (*Current* < *Current*)

end

A.6.4 CLASS COMPARABLE

note

description: "[
 Objects such that any two can be compared through
 to a total order relation
]"

math: "The model is a total order relation."

comment: [
 "The basic operation is "<" (less than); others are
 defined in terms of this operation and *is_equal*.
]"

deferred class interface

COMPARABLE

feature -- Access

up_to **alias** ".." (*other: COMPARABLE*):
INTERVAL [*COMPARABLE*]
 -- Interval containing all values *t*, if any, such that
 -- *Current* <= *t* and *t* <= *other*
 -- Empty if *Current* > *other*

feature -- Comparison

is_comparable " (*other: like Current*): *BOOLEAN*
 -- Do current object and *other* figure in the relation?
 -- (From *PART_COMPARABLE*); here lways true
 -- for a total order)

ensure

total_order: *Result* = *True*

is_less **alias** "<" (*other: like Current*): *BOOLEAN*
 -- Is current object less than *other*?

deferred**ensure**

asymmetric: *Result* **implies not** (*other* < *Current*)

is_less_equal **alias** "<=" (*other: like Current*):
BOOLEAN

-- Is current object less than or equal to *other*?

ensure

definition: *Result* = ((*Current* < *other*) **or**
 (*Current* ~ *other*))

is_greater_equal **alias** ">=" (*other: like Current*):
BOOLEAN

-- Is current object greater than or equal to *other*?

ensure

definition: *Result* = (*other* <= *Current*)

is_greater **alias** ">" (*other: like Current*): *BOOLEAN*
 -- Is current object greater than *other*?

ensure

definition: *Result* = (*other* < *Current*)

is_equal (*other: like Current*): *BOOLEAN*

-- Is *other* attached to an object considered equal
 -- to current object?
 -- The object comparison operator ~ relies on this function.

ensure

symmetric: *Result* **implies** *other.is_equal* (*Current*)
 consistent: *default_is_equal* (*other*) **implies** *Result*
 trichotomy: *Result* = (**not** (*Current* < *other*) **and not**
 (*other* < *Current*))

max (*other: like Current*): **like** *Current*

-- The greater of current object and *other*

ensure

current_if_not_smaller: (*Current* >= *other*) **implies**
 (*Result* = *Current*)

other_if_smaller: (*Current* < *other*) **implies** (*Result*
 = *other*)

min (*other: like Current*): **like** *Current*

-- The smaller of current object and *other*

ensure

current_if_not_greater: (*Current* <= *other*) **implies**
 (*Result* = *Current*)

other_if_greater: (*Current* > *other*) **implies** (*Result*
 = *other*)

three_way_comparison (*other: like Current*): *INTEGER*

-- If current object equal to *other*, 0;
 -- if smaller, -1; if greater, 1.

ensure

equal_zero: (*Result* = 0) = (*Current* ~ *other*)

smaller_negative: (*Result* = -1) = (*Current* < *other*)

greater_positive: (*Result* = 1) = (*Current* > *other*)

invariant

irreflexive_comparison: **not** (*Current* < *Current*)

end

A.6.5 CLASS *HASHABLE*

note

description: "[
 Values that may be hashed into an integer index, for
 use as keys in hash tables
]"

deferred class interface

HASHABLE

feature -- Access

hash_code: *INTEGER*
 -- Hash code value

deferred**ensure**

good_hash_value: *Result* >= 0

end

A.6.6 CLASS *NUMERIC*

note

description: "[
Objects to which numerical operations are applicable
]"

math: "The model is a commutative ring."

deferred class interface

NUMERIC

feature -- Access

one: **like** *Current*
-- Neutral element for "*" and "/"

deferred

zero: **like** *Current*
-- Neutral element for "+" and "-"

deferred

feature -- Status report

divisible (other: **like** *Current*): *BOOLEAN*
-- May current object be divided by other?

deferred

exponentiable (other: *NUMERIC*): *BOOLEAN*
-- May current object be elevated to the power other?

deferred

feature -- Basic operations

plus **alias** "+" (other: **like** *Current*): **like** *Current*
-- Sum with other (commutative).

deferred

ensure

commutative: *equal* (*Result*, *other* + *Current*)

minus **alias** "-" (other: **like** *Current*): **like** *Current*
-- Result of subtracting other

deferred

ensure

consistent: *Result* + *other* = *Current*

product **alias** "*" (other: **like** *Current*): **like** *Current*
-- Product by other

deferred

divided **alias** "/" (other: **like** *Current*): **like** *Current*
-- Division by other

require

good_divisor: *divisible* (other)

deferred

power **alias** "^" (other: *NUMERIC*): *NUMERIC*
-- Current object to the power other

require

good_exponent: *exponentiable* (other)

deferred

identity **alias** "+": **like** *Current*
-- Unary plus

deferred

negated **alias** "-": **like** *Current*
-- Unary minus

deferred

invariant

neutral_addition: *equal* (*Current* + *zero*, *Current*)

self_subtraction: *equal* (*Current* - *Current*, *zero*)

neutral_multiplication: *equal* (*Current* * *one*, *Current*)

self_division: *divisible* (*Current*) **implies** *equal*
(*Current* / *Current*, *one*)

end

A.6.7 CLASS INTERVAL

note

description: "[
 Sets of values, from a partially or totally
 ordered set *G*, all between two given bounds
]"

class interface

INTERVAL [*G* → *PART_COMPARABLE*]

create

make (*l*, *u*: *G*)
 -- Set bounds to *l* and *u*; make interval empty if $l > u$.

require

comparable: *l.is_comparable* (*u*)

ensure

lower_set: *lower* = *l*
 lower_set: *upper* = *u*

feature -- Initialization

make (*l*, *u*: *G*)
 -- Set bounds to *l* and *u*; make interval empty if $l > u$.

require

comparable: *l.is_comparable* (*u*)

ensure

lower_set: *lower* = *l*
 lower_set: *upper* = *u*

feature -- Access

lower: *G*
 -- Lower bound

upper: *G*
 -- Upper bound

feature -- Comparison

is_comparable " (*other*: **like** *Current*): *BOOLEAN*
 -- Is either one of current interval and *other*
 -- strictly contained in the other?

ensure

definition: *Result* = (*Current* < *other*) **or**
 ((*Current* ~ *other*) **or** (*Current* > *other*)

is_subinterval **alias** "<" (*other*: **like** *Current*):
BOOLEAN

-- Is current interval strictly included in *other*?

deferred**ensure**

definition: *Result* = *lower* > *other.lower* **and** *upper*
 < *other.upper*

is_superinterval **alias** ">" (*other*: **like** *Current*):
BOOLEAN

-- Does current interval strictly include *other*?

ensure

definition: *Result* = (*other* < *Current*)

... **OTHER COMPARISON FEATURES**
AS IN CLASS PART_COMPARABLE ...

feature -- Status report

is_empty: *BOOLEAN*
 -- Does interval contain no values?

invariant

consistent: *lower.is_comparable* (*upper*)
 empty_if_no_values: *is_empty* = (*lower* > *upper*)

end

A.6.8 CLASS *BOOLEAN*

note

description: "Truth values with boolean operations"

expanded class interface

BOOLEAN

feature -- Access

hash_code: *INTEGER*
 -- Hash code value
 -- (From *HASHABLE*.)

ensure

good_hash_value: *Result* >= 0

feature -- Basic operations

conjoined **alias** "and" (other: *BOOLEAN*):
BOOLEAN
 -- Boolean conjunction with other

ensure

de_morgan: *Result* = **not (not Current or (not other))**
 commutative: *Result* = (other **and** Current)
 consistent_with_semi_strict: *Result* **implies**
 (Current **and then** other)

conjoined_semistrict **alias** "and then" (other:
BOOLEAN): *BOOLEAN*
 -- Boolean semi-strict conjunction with other

ensure

de_morgan: *Result* = **not (not Current or else (not other))**

implication **alias** "implies" (other: *BOOLEAN*):
BOOLEAN

-- Boolean implication of other
 -- (semi-strict)

ensure

definition: *Result* = (**not Current or else other**)

negated **alias** "not": *BOOLEAN*

-- Negation.

disjuncted **alias** "or" (other: *BOOLEAN*): *BOOLEAN*
 -- Boolean disjunction with other

ensure

de_morgan: *Result* = **not (not Current and (not other))**
 commutative: *Result* = (other **or** Current)
 consistent_with_semi_strict: *Result* **implies**
 (Current **or else** other)

disjuncted_semistrict **alias** "or else" (other:
BOOLEAN): *BOOLEAN*

-- Boolean semi-strict disjunction with other

ensure

de_morgan: *Result* = **not (not Current and then (not other))**

disjuncted_exclusive **alias** "xor" (other: *BOOLEAN*):
BOOLEAN

-- Boolean exclusive or with other

ensure

definition: *Result* = ((Current **or** other) **and not**
 (Current **and** other))

feature -- Output

out: *STRING*
 -- Printable representation of boolean

invariant

involutive_negation: Current ~ (**not (not Current)**)
 non_contradiction: **not** (Current **and** (**not Current**))
 excluded_middle: Current **or** (**not Current**)

end

A.6.9 CLASS CHARACTER

note

description: "[
 Characters, with comparison operations and an
 ASCII code
]"

expanded class interface

CHARACTER

feature -- Access

code: INTEGER

-- Associated integer value

hash_code: INTEGER

-- Hash code value

-- (From HASHABLE.)

ensure

good_hash_value: Result >= 0

up_to **alias** ".." (other: CHARACTER) :

INTERVAL [CHARACTER]

-- Interval containing all characters *c*, if any, such that

-- *Current* <= *c* and *c* <= *other*

-- Empty if *Current* > *other*

feature -- Comparison

is_less **alias** "<" (other: **like** Current): BOOLEAN

-- Is *other* greater than current character?

-- (From COMPARABLE.)

ensure

asymmetric: Result **implies not** (other < Current)

is_less_equal **alias** "<=" (other: **like** Current):

BOOLEAN

-- Is current character less than or equal to *other*?

-- (From COMPARABLE.)

ensure

definition: Result = (Current < other) **or**
 (Current ~ other)

is_greater_equal **alias** ">=" (other: **like** Current):

BOOLEAN

-- Is current object greater than or equal to *other*?

-- (From COMPARABLE.)

ensure

definition: Result = (other <= Current)

is_greater **alias** ">" (other: **like** Current): BOOLEAN

-- Is current object greater than *other*?

-- (From COMPARABLE.)

ensure

definition: Result = (other < Current)

max (other: **like** Current): **like** Current

-- The greater of current object and *other*

-- (From COMPARABLE.)

ensure

current_if_not_smaller: (Current >= other) **implies**
 (Result = Current)

other_if_smaller: (Current < other) **implies** (Result
 = other)

min (other: **like** Current): **like** Current

-- The smaller of current object and *other*

-- (From COMPARABLE.)

ensure

current_if_not_greater: (Current <= other) **implies**
 (Result = Current)

other_if_greater: (Current > other) **implies** (Result
 = other)

three_way_comparison (other: **like** Current):

INTEGER

-- If current object equal to *other*, 0;

-- if smaller, -1; if greater, 1.

-- (From COMPARABLE.)

ensure

equal_zero: (Result = 0) = (Current ~ other)

smaller: (Result = -1) = Current < other

greater_positive: (Result = 1) = Current > other

feature -- Output

out: STRING

-- Printable representation of character

-- (From ANY.)

invariant

irreflexive_comparison: **not** (Current < Current)

end

A.6.10 CLASS INTEGER_GENERAL

note

description: "Integer values of set size"

class interface

INTEGER_GENERAL

create

make (*b*: INTEGER)

- Initialize with bit size *b*.
- (No effect on expanded targets.)

require

positive: $b > 0$

ensure

bit_size_set: $bit_size = b$

default_create

- Initialize with default bit size: 32.

ensure

bit_size_set: $bit_size = Default_bit_size$

from_integer **convert** (*other*: INTEGER_GENERAL)

- Initialize from *other*; do not lose any precision.

ensure

bit_size_set: $bit_size = Default_bit_size$

feature -- Access

bit_size: INTEGER

- Number of bits in representation

Default_bit_size: INTEGER

- Number of bits in representation

hash_code: INTEGER

- Hash code value
- (From HASHABLE.)

ensure

good_hash_value: $Result \geq 0$

one: **like** Current

- Neutral element for "*" and "/"
- (From NUMERIC.)

ensure

value: $Result = 1$

sign: INTEGER

- Sign value (0, -1 or 1)

ensure

three_way: $Result = three_way_comparison$ (*zero*)

up_to **alias** ".." (*other*: INTEGER_GENERAL):

INTERVAL [INTEGER_GENERAL]

- Interval containing all integers *i*, if any, such that
- $Current \leq i$ and $i \leq other$
- Empty if $Current > other$

zero: **like** Current

- Neutral element for "+" and "-"
- (From NUMERIC.)

ensure

value: $Result = 0$

feature -- Comparison

is_less **alias** "<" (*other*: **like** Current): BOOLEAN

- Is *other* greater than current integer?
- (From COMPARABLE.)

ensure

asymmetric: $Result$ **implies not** (*other* < Current)

is_less_equal **alias** "<=" (*other*: **like** Current): BOOLEAN

- Is current object less than or equal to *other*?
- (From COMPARABLE.)

ensure

definition: $Result = (Current < other) \text{ or } (Current \sim other)$

is_greater_equal **alias** ">=" (*other*: **like** Current): BOOLEAN

- Is current object greater than or equal to *other*?
- (From COMPARABLE.)

ensure

definition: $Result = (other \leq Current)$

is_greater **alias** ">" (*other*: **like** Current): BOOLEAN

- Is current object greater than *other*?
- (From COMPARABLE.)

ensure

definition: $Result = (other < Current)$

max (*other*: **like** Current): **like** Current

- The greater of current object and *other*
- (From COMPARABLE.)

ensure

current_if_not_smaller: ($Current \geq other$) **implies** ($Result = Current$)

other_if_smaller: ($Current < other$) **implies** ($Result = other$)

min (*other*: **like** Current): **like** Current

- The smaller of current object and *other*
- (From COMPARABLE.)

ensure

current_if_not_greater: ($Current \leq other$) **implies** ($Result = Current$)

other_if_greater: ($Current > other$) **implies** ($Result = other$)

three_way_comparison (other: **like** *Current*):

INTEGER

-- If current object equal to *other*, 0;

-- if smaller, -1; if greater, 1.

-- (From *COMPARABLE*.)

ensure

equal_zero: $(Result = 0) = (Current \sim other)$

smaller: $(Result = 1) = Current < other$

greater_positive: $(Result = -1) = Current > other$

feature -- Status report

divisible (other: **like** *Current*): *BOOLEAN*

-- May current object be divided by *other*?

-- (From *NUMERIC*.)

ensure

value: $Result = (other \neq 0)$

exponentiable (other: *NUMERIC*): *BOOLEAN*

-- May current object be elevated to the power *other*?

-- (From *NUMERIC*.)

ensure

safe_values: $(other.conforms_to (Current) \text{ or } (other.conforms_to (0, 0) \text{ and } (Current \geq 0)))$
implies *Result*

bit_one (*n*: *INTEGER*): *BOOLEAN*

-- Is *n*-th bit (from left, in binary representation)

-- a one?

require

at_most_size: $n \leq bit_size$

at_least_one: $n \geq 1$

feature --Element change

bit_shift (*n*: *INTEGER*): **like** *Current*

-- Bit-shift *n* positions, to right if positive,

-- left otherwise.

require

at_most_size: $n \leq bit_size$

at_least_minus_size: $n \geq -size$

bit_shift_left (*n*: *INTEGER*): **like** *Current*

-- Bit-shift *n* positions to left.

require

non_negative: $n \geq 0$

at_most_size: $n \leq bit_size$

bit_shift_right (*n*: *INTEGER*): **like** *Current*

-- Bit-shift *n* positions to right.

require

non_negative: $n \geq 0$

at_most_size: $n \leq bit_size$

feature -- Basic operations

abs: **like** *Current*

-- Absolute value

ensure

non_negative: $Result \geq 0$

same_absolute_value: $(Result = Current) \text{ or } (Result = -Current)$

product **alias** "*" (other: **like** *Current*): **like** *Current*

-- Product by *other*

-- (From *NUMERIC*.)

plus **alias** "+" (other: **like** *Current*): **like** *Current*

-- Sum with *other*

-- (From *NUMERIC*.)

ensure

commutative: $equal (Result, other + Current)$

minus **alias** "-" (other: **like** *Current*): **like** *Current*

-- Result of subtracting *other*

-- (From *NUMERIC*.)

ensure

consistent: $Result + other = Current$

divided **alias** "/" (other: **like** *Current*): *REAL*

-- Division by *other*

require

good_divisor: *divisible* (*other*)

quotient **alias** "//" (other: **like** *Current*): **like** *Current*

-- Integer division of *Current* by *other*

-- (From "/" in *NUMERIC*.)

require

good_divisor: *divisible* (*other*)

ensure

result_exists: *divisible* (*other*)

remainder **alias** "%" (other: **like** *Current*): **like** *Current*

-- Remainder of integer division of *Current* by *other*

require

good_divisor: *divisible* (*other*)

power **alias** "^" (other: *NUMERIC*): *REAL*

-- Integer power of *Current* by *other*

-- (From *NUMERIC*.)

require

good_exponent: *exponentiable* (*other*)

identity **alias** "+": **like** *Current*

-- Unary plus

-- (From *NUMERIC*.)

negated **alias** "-": **like** *Current*

-- Unary minus

-- (From *NUMERIC*.)

bit_and (*i*: **like** *Current*): **like** *Current*

-- Bitwise and with *i*.

bit_or (*i*: **like** *Current*): **like** *Current*

-- Bitwise or with *i*.

bit_xor (*i*: **like** *Current*): **like** *Current*

-- Bitwise exclusive or with *i*.

bit_not: **like** *Current*

-- One's complement.

feature -- Output

out: *STRING*

-- Printable representation of current object

-- (From *ANY*.)

invariant

bit_size_positive: *bit_size* > 0

default_bit_size_positive: *default_bit_size* > 0

irreflexive_comparison: **not** (*Current* < *Current*)

neutral_addition: *equal* (*Current* + *zero*, *Current*)

self_subtraction: *equal* (*Current* - *Current*, *zero*)

neutral_multiplication: *equal* (*Current* * *one*, *Current*)

self_division: *divisible* (*Current*) **implies** *equal*
(*Current* / *Current*, *one*)

sign_times_abs: *equal* (*sign** *abs*, *Current*)

end

A.6.11 CLASS *INTEGER*

note

description: "32-bit integer values"

expanded class interface

INTEGER

create

default_create

-- Initialize with default bit size: 32.

ensure

bit_size_set: *bit_size* = 32

from_integer convert (*b*: *INTEGER_GENERAL*)

-- Initialize from *other*, losing leftmost part if
-- *other* is of smaller bit size.

ensure

bit_size_set: *bit_size* = *Default_bit_size*

feature

... SAME FEATURE SPECIFICATIONS

AS CLASS *INTEGER_GENERAL* ...

invariant

... SAME INVARIANT CLAUSES

AS CLASS *INTEGER_GENERAL*, PLUS:

bit_size_definition: *bit_size* = 32

end

A.6.12 CLASS *INTEGER_8*

note

description: "8-bit integer values"

expanded class interface

INTEGER_8

create

default_create

-- Initialize with default bit size: 8.

ensure

bit_size_set: bit_size = 8

from_integer (other: INTEGER_GENERAL)

-- Initialize from *other*, losing leftmost part if
-- *other* is of smaller bit size.

ensure

bit_size_set: bit_size = Default_bit_size

feature

... SAME FEATURE SPECIFICATIONS

AS CLASS *INTEGER_GENERAL* ...

invariant

... SAME INVARIANT CLAUSES

AS CLASS *INTEGER_GENERAL*, PLUS:

bit_size_definition: bit_size = 8

end

A.6.13 CLASS INTEGER_16

note

description: "16-bit integer values"

expanded class interface

INTEGER_16

create

default_create

-- Initialize with default bit size: 16.

ensure

bit_size_set: *bit_size* = 16

from_integer **convert** (*other*: *INTEGER_GENERAL*)

-- Initialize from *other*, losing leftmost part if

-- *other* is of smaller bit size.

ensure

bit_size_set: *bit_size* = *Default_bit_size*

feature

... SAME FEATURE SPECIFICATIONS

AS CLASS *INTEGER_GENERAL* ...

invariant

... SAME INVARIANT CLAUSES

AS CLASS *INTEGER_GENERAL*, PLUS:

bit_size_definition: *bit_size* = 16

end

A.6.14 CLASS *INTEGER_64*

note

description: "64-bit integer values"

expanded class interface

INTEGER_64

create

default_create

-- Initialize with default bit size: 64.

ensure

bit_size_set: bit_size = 64

from_integer convert (*other: INTEGER_GENERAL*)

-- Initialize from *other*, losing leftmost part if

-- *other* is of smaller bit size.

ensure

bit_size_set: bit_size = Default_bit_size

feature

... SAME FEATURE SPECIFICATIONS

AS CLASS *INTEGER_GENERAL* ...

invariant

... SAME INVARIANT CLAUSES

AS CLASS *INTEGER_GENERAL*, PLUS:

bit_size_definition: bit_size = 64

end

A.6.15 CLASS REAL_GENERAL

note

description: "Real values, single precision"

expanded class interface*REAL***feature** -- Access*hash_code*: *INTEGER*-- Hash code value
-- (From *HASHABLE*.)**ensure**good_hash_value: *Result* ≥ 0 *one*: **like** *Current*-- Neutral element for "*" and "/"
-- (From *NUMERIC*.)**ensure**value: *Result* = 1.0*sign*: *INTEGER*

-- Sign value (0, -1 or 1)

ensurethree_way: *Result* = *three_way_comparison* (*zero*)*up_to* **alias** ".." (*other*: *REAL_GENERAL*):*INTERVAL* [*IREAL_GENERAL*]-- Interval containing all reals *r*, if any, such that
-- *Current* $\leq r$ and $r \leq other$
Empty if *Current* $> other$ *zero*: **like** *Current*-- Neutral element for "+" and "-"
-- (From *NUMERIC*.)**ensure**value: *Result* = 0.0**feature** -- Comparison*is_less* **alias** "<" (*other*: **like** *Current*): *BOOLEAN*-- Is *other* greater than current real?
-- (From *COMPARABLE*.)**ensure**asymmetric: *Result* **implies not** (*other* $<$ *Current*)*is_less_equal* **alias** " \leq " (*other*: **like** *Current*):
BOOLEAN-- Is current object less than or equal to *other*?
-- (From *COMPARABLE*.)**ensure**definition: *Result* = (*Current* $<$ *other*) **or**
(*Current* \sim *other*)*is_greater_equal* **alias** " \geq " (*other*: **like** *Current*):
BOOLEAN-- Is current object greater than or equal to *other*?
-- (From *COMPARABLE*.)**ensure**definition: *Result* = (*other* \leq *Current*)*is_greater* **alias** ">" (*other*: **like** *Current*): *BOOLEAN*-- Is current object greater than *other*?
-- (From *COMPARABLE*.)**ensure**definition: *Result* = (*other* $<$ *Current*)*max* (*other*: **like** *Current*): **like** *Current*-- The greater of current object and *other*
-- (From *COMPARABLE*.)**ensure**current_if_not_smaller: (*Current* $\geq other$) **implies**
(*Result* = *Current*)other_if_smaller: (*Current* $< other$) **implies** (*Result*
= *other*)*min* (*other*: **like** *Current*): **like** *Current*-- The smaller of current object and *other*
-- (From *COMPARABLE*.)**ensure**current_if_not_greater: (*Current* $\leq other$) **implies**
(*Result* = *Current*)other_if_greater: (*Current* $> other$) **implies** (*Result*
= *other*)*three_way_comparison* (*other*: **like** *Current*):
INTEGER-- If current object equal to *other*, 0;
-- if smaller, -1; if greater, 1.
-- (From *COMPARABLE*.)**ensure**equal_zero: (*Result* = 0) = (*Current* $\sim other$)smaller: (*Result* = -1) = *Current* $<$ *other*greater_positive: (*Result* = 1) = *Current* $>$ *other***feature** -- Status report*divisible* (*other*: **like** *Current*): *BOOLEAN*-- May current object be divided by *other*?
-- (From *NUMERIC*.)**ensure**not_exact_zero: *Result* **implies** (*other* $\neq 0.0$)*exponentiable* (*other*: *NUMERIC*): *BOOLEAN*-- May current object be elevated to the power *other*?
-- (From *NUMERIC*.)**ensure**safe_values: (*other.conforms_to* (0) **or**
(*other.conforms_to* (*Current*) **and** (*Current* \geq
0.0))) **implies** *Result*

feature -- Conversion*ceiling: INTEGER*

- Smallest integral value no smaller than
- current object

ensure

result_no_smaller: Result >= Current
close_enough: Result - Current < one

floor: INTEGER

- Greatest integral value no greater than
- current object

ensure

result_no_greater: Result <= Current
close_enough: Current - Result < one

rounded: INTEGER

- Rounded integral value

ensure

*definition: Result = sign * ((abs + 0.5).floor)*

truncated_to_integer: INTEGER

- Integer part (same sign, largest absolute
- value no greater than current object's)

feature -- Basic operations*abs: like Current*

- Absolute value

ensure

non_negative: Result >= 0
same_absolute_value: (Result = Current) or (Result = -Current)

product alias "" (other: like Current): like Current*

- Product by other
- (From NUMERIC.)

plus alias "+" (other: like Current): like Current

- Sum with other
- (From NUMERIC.)

ensure

commutative: equal (Result, other + Current)

minus alias "-" (other: like Current): like Current

- Result of subtracting other
- (From NUMERIC.)

ensure

consistent: Result + other = Current

divided alias "/" (other: like Current): like Current

- Division by other
- (From NUMERIC.)

require

good_divisor: divisible (other)

power alias "^" (other: NUMERIC): REAL

- Current real to the power other
- (From NUMERIC.)

require

good_exponent: exponentiable (other)

identity alias "+": like Current

- Unary plus
- (From NUMERIC.)

negated alias "-": like Current

- Unary minus
- (From NUMERIC.)

feature -- Output*out: STRING*

- Printable representation of real value
- (From ANY.)

invariant*irreflexive_comparison: not (Current < Current)**neutral_addition: equal (Current + zero, Current)**self_subtraction: equal (Current - Current, zero)**neutral_multiplication: equal (Current * one, Current)**self_division: divisible (Current) implies equal (Current / Current, one)**sign_times_abs: equal (sign*abs, Current)***end**

A.6.16 CLASS *REAL*

note

description: "32-bit real values"

expanded class interface

REAL

feature

... SAME FEATURE SPECIFICATIONS

AS CLASS *REAL GENERAL* ...

end

A.6.17 CLASS TYPED_POINTER

A.6.18 CLASS *POINTER*

note

description: "[
References to objects meant to be exchanged with
non-Eiffel software
]"

expanded class interface

POINTER

feature -- Access

hash_code: *INTEGER*
-- Hash code value
-- (From *HASHABLE*.)

ensure

good_hash_value: *Result* >= 0

feature -- Basic operations

plus **alias** "+" (*offset*: *INTEGER*): *POINTER*
-- Pointer to address at current position plus
-- *offset* bytes

feature -- Output

out: *STRING*
-- Printable representation of pointer value
-- (From *ANY*.)

end

A.6.19 CLASS ARRAY

note

description: "[
 Sequences of values, all of the same type or of a
 conforming one, accessible through integer indices
 in a contiguous interval
]"

class interface

ARRAY [G]

create

make (*minindex*, *maxindex*: INTEGER)
 -- Allocate array; set index interval to
 -- *minindex* .. *maxindex*; set all values to default.
 -- (Make array empty if *minindex* > *maxindex*.)

ensure

empty_if_bounds_dont_fit: (*minindex* > *maxindex*)
implies (*count* = 0)
bounds_set: (*minindex* <= *maxindex*) **implies**
 ((*lower* = *minindex*) **and** (*upper* = *maxindex*))

from_interval (*int*: INTERVAL [INTEGER])
 -- Allocate array; set index interval to *int*;
 -- set all values to default.
 -- (Make array empty if interval is empty.)

ensure

empty_if_bounds_dont_fit: (*int.is_empty*) **implies**
 (*count* = 0)
bounds_set: **not** (*int.is_empty*) **implies**
 ((*lower* = *int.lower*) **and** (*upper* = *int.upper*))

feature -- Access

item **alias** "[]" **assign** "put" (*i*: INTEGER): G
 -- Entry at index *i*

require

good_key: *valid_index* (*i*)

feature -- Measurement

bounds: INTERVAL [INTEGER]
 -- Integer interval for indices

count: INTEGER
 -- Number of available indices

lower: INTEGER
 -- Minimum index

upper: INTEGER
 -- Maximum index

feature -- Status report

valid_index (*i*: INTEGER): BOOLEAN
 -- Is *i* within the bounds of the array?

feature -- Element change

force (*v*: **like** *item*; *i*: INTEGER)
 -- Assign item *v* to *i*-th entry.
 -- Always applicable: resize the array if *i* falls out of
 -- currently defined bounds; preserve existing items.

ensure

inserted: *item* (*i*) = *v*
higher_count: *count* >= **old** *count*

put (*v*: **like** *item*; *i*: INTEGER)
 -- Replace *i*-th entry, if in index interval, by *v*.

require

good_key: *valid_index* (*i*)

ensure

inserted: *item* (*i*) = *v*

feature -- Resizing

resize (*minindex*, *maxindex*: INTEGER)
 -- Rearrange array so that it can accommodate
 -- indices down to *minindex* and up to *maxindex*.
 -- Do not lose any previously entered item.

require

good_indices: *minindex* <= *maxindex*

invariant

consistent_size: *count* = *upper* - *lower* + 1
non_negative_count: *count* >= 0
interval_consistent: *bounds* ~ *lower* .. *upper*

end

A.6.20 CLASS *ANONYMOUS*

note

description: "[
 Tuples: finite sequences of values, each of a specified
 type
]"

class interface*ANONYMOUS***feature** -- Access*item: ANY*-- *i*-th element of tuple**require**good_key: *valid_index* (*i*)*hash_code: INTEGER*

-- Hash code value

-- (From *HASHABLE*.)**ensure**good_hash_value: *Result* >= 0**feature** -- Measurement*count: INTEGER*

-- Minimum member of items in tuple

feature -- Status report*valid_index* (*i: INTEGER*): *BOOLEAN*-- Is *i* within the bounds of the array?**ensure**ok_if_between_one_and_count:
 ((*i* >= 1) and (*i* <= *count*)) **implies***Result***feature** -- Element change*put* (*v: ANY; i: INTEGER*)-- Replace *i*-th item by *v*.**require**good_key: *valid_index* (*i*)**ensure**replaced: *item* (*i*) = *v***end**

A.6.21 CLASS *STRING*

note

description: "[
Sequences of characters, accessible through integer
indices in a contiguous range.
]"

class interface

STRING

create

frozen *make* (*n*: *INTEGER*)

-- Allocate space for at least *n* characters.

require

non_negative_size: *n* >= 0

ensure

empty_string: *count* = 0

from_string (*s*: *STRING*)

-- Initialize from the characters of *s*.
-- (Useful in proper descendants of class *STRING*,
-- to initialize a string-like object from a manifest string.)

feature -- Initialization

from_c (*c_string*: *POINTER*)

-- Reset contents of string from contents of *c_string*,
-- a string created by some external C function.

frozen *remake* (*n*: *INTEGER*)

-- Allocate space for at least *n* characters.

require

non_negative_size: *n* >= 0

ensure

empty_string: *count* = 0

from_string (*s*: *STRING*)

-- Initialize from the characters of *s*.
-- (Useful in proper descendants of class *STRING*,
-- to initialize a string-like object from a manifest string.)

feature -- Access

hash_code: *INTEGER*

-- Hash code value
-- (From *HASHABLE*.)

ensure

good_hash_value: *Result* >= 0

index_of (*c*: *CHARACTER*; *start*: *INTEGER*):
INTEGER

-- Position of first occurrence of *c* at or after *start*;
-- 0 if none.

require

start_large_enough: *start* >= 1

start_small_enough: *start* <= *count*

ensure

non_negative_result: *Result* >= 0

at_this_position: *Result* > 0 **implies** *item* (*Result*) = *c*

-- none_before: **For every** *i* in *start*..*Result*, *item* (*i*) /= *c*

-- zero_iff_absent:

-- (*Result* = 0) = **For every** *i* in 1..*count*, *item* (*i*) /= *c*

item **alias** "[]" (*i*: *INTEGER*): *CHARACTER*

-- Character at position *i*

require

good_key: *valid_index* (*i*)

substring_index (*other*: *STRING*; *start*: *INTEGER*):
INTEGER

-- Position of first occurrence of *other* at or after *start*;
-- 0 if none.

up_to **alias** ". ." (*other*: *STRING*):
INTERVAL [*STRING*]

-- Interval containing all strings *s*, if any, such that
-- *Current* <= *s* and *s* <= *other*
-- Empty if *Current* > *other*

feature -- Measurement

count: *INTEGER*

-- Actual number of characters making up the string

occurrences (*c*: *CHARACTER*): *INTEGER*

-- Number of times *c* appears in the string

ensure

non_negative_occurrences: *Result* >= 0

feature -- Comparison

is_equal (*other*: **like** *Current*): *BOOLEAN*

-- Is string made of same character sequence as *other*?
-- The object comparison operator ~ relies on this function.

is_less **alias** "<" (*other*: *STRING*): *BOOLEAN*

-- Is string lexicographically lower than *other*?
-- (From *COMPARABLE*.)

ensure

asymmetric: *Result* **implies not** (*other* < *Current*)

is_less_equal **alias** " \leq " (*other*: **like** *Current*):

BOOLEAN

-- Is current object less than or equal to *other*?

-- (From *COMPARABLE*.)

ensure

definition: *Result* = (*Current* < *other*) **or** (*Current* ~ *other*)

is_greater_equal **alias** " \geq " (*other*: **like** *Current*):

BOOLEAN

-- Is current object greater than or equal to *other*?

-- (From *COMPARABLE*.)

ensure

definition: *Result* = (*other* <= *Current*)

is_greater **alias** ">" (*other*: **like** *Current*): *BOOLEAN*

-- Is current object greater than *other*?

-- (From *COMPARABLE*.)

ensure

definition: *Result* = (*other* < *Current*)

max (*other*: **like** *Current*): **like** *Current*)

-- The greater of current object and *other*

-- (From *COMPARABLE*.)

ensure

current_if_not_smaller: (*Current* >= *other*) **implies** (*Result* = *Current*)

other_if_smaller: (*Current* < *other*) **implies** (*Result* = *other*)

min (*other*: **like** *Current*): **like** *Current*)

-- The smaller of current object and *other*

-- (From *COMPARABLE*.)

ensure

current_if_not_greater: (*Current* <= *other*) **implies** (*Result* = *Current*)

other_if_greater: (*Current* > *other*) **implies** (*Result* = *other*)

three_way_comparison (*other*: **like** *Current*):

INTEGER)

-- If current object equal to *other*, 0;

-- if smaller, -1; if greater, 1.

-- (From *COMPARABLE*.)

ensure

equal_zero: (*Result* = 0) = (*Current* ~ *other*)

smaller: (*Result* = -1) = *Current* < *other*

greater_positive: (*Result* = 1) = *Current* > *other*

feature -- Status report

is_empty: *BOOLEAN*

-- Does string contain no characters?

valid_index (*i*: *INTEGER*): *BOOLEAN*

-- Is *i* within the bounds of the string?

feature -- Element change

append_boolean (*b*: *BOOLEAN*)

-- Append the string representation of *b* at end.

append_character (*c*: *CHARACTER*)

-- Append *c* at end.

ensure

item_inserted: *item* (*count*) = *c*

one_more_occurrence: *occurrences* (*c*) = **old** (*occurrences* (*c*)) + 1

item_inserted: *has* (*c*)

append_integer (*i*: *INTEGER*)

-- Append the string representation of *i* at end.

append_real (*r*: *REAL*)

-- Append the string representation of *r* at end.

append_string (*s*: *STRING*)

-- Append a copy of *s* at end.

ensure

new_count: *count* = **old** *count* + *s*.*count*

-- appended: For every *i* in 1..*s*.*count*,

-- *item* (*old* *count* + *i*) = *s*.*item* (*i*)

fill (*c*: *CHARACTER*)

-- Replace every character with *c*.

ensure

-- *allblank*: For every *i* in 1..*count*, *item* (*i*) = *Blank*

head (*n*: *INTEGER*)

-- Remove all characters except for the first *n*;

-- do nothing if *n* >= *count*.

require

non_negative_argument: *n* >= 0

ensure

new_count: *count* = *n*.*min* (**old** *count*)

-- *first_kept*: For every *i* in 1..*n*, *item* (*i*) = *old* *item* (*i*)

insert (*s*: **like** *Current*; *i*: *INTEGER*)

-- Add *s* to the left of position *i*.

require

index_small_enough: *i* <= *count*

index_large_enough: *i* > 0

ensure

new_count: *count* = **old** *count* + *s*.*count*

insert_character (*c*: *CHARACTER*; *i*: *INTEGER*)

-- Add *c* to the left of position *i*.

ensure

new_count: *count* = **old** *count* + 1

left_adjust

-- Remove leading white space.

ensure

new_count: (*count* /= 0) **implies** (*item* (*I*) /= ' ')

```

put (c: CHARACTER; i: INTEGER)
  -- Replace character at position i by c.
require
  good_key: valid_index (i)
ensure
  insertion_done: item (i) = c
put_substring (s: like Current; start_pos, end_pos:
  INTEGER)
  -- Copy the characters of s to positions
  -- start_pos .. end_pos.
require
  index_small_enough: end_pos <= count
  order_respected: start_pos <= end_pos
  index_large_enough: start_pos > 0
ensure
  new_count: count = old count + s.count - end_pos
  + start_pos - 1
right_adjust
  -- Remove trailing white space.
ensure
  new_count: (count /= 0) implies (item (count) /= ' ')
tail (n: INTEGER)
  -- Remove all characters except for the last n;
  -- do nothing if n >= count.
require
  non_negative_argument: n >= 0
ensure
  new_count: count = n.min (old count)
feature -- Removal
remove (i: INTEGER)
  -- Remove i-th character.
require
  index_small_enough: i <= count
  index_large_enough: i > 0
ensure
  new_count: count = old count - 1
wipe_out
  -- Remove all characters.
ensure
  empty_string: count = 0
  wiped_out: is_empty
feature -- Resizing
resize (newsiz: INTEGER)
  -- Rearrange string so that it can accommodate
  -- at least newsiz characters.
  -- Do not lose any previously entered character.
require
  new_size_non_negative: newsiz >= 0

```

```

feature -- Conversion
to_boolean: BOOLEAN
  -- Boolean value;
  -- "true" yields true, "false" yields false
  -- (case-insensitive)
to_integer: INTEGER
  -- Integer value;
  -- for example, when applied to "123", will yield 123
to_lower
  -- Convert to lower case.
to_real: REAL
  -- Real value;
  -- for example, when applied to "123.0", will yield 123.0
to_upper
  -- Convert to upper case.
feature -- Duplication
copy (other: like Current)
  -- Reinitialize by copying the characters of other.
  -- (This is also used by clone.)
  -- (From ANY.)
ensure
  new_result_count: count = other.count
  -- same_characters: For every i in 1..count,
  --   item (i) = other.item (i)
substring (n1, n2: INTEGER): like Current
  -- Copy of substring containing all characters at indices
  -- between n1 and n2
require
  meaningful_origin: 1 <= n1
  meaningful_interval: n1 <= n2
  meaningful_end: n2 <= count
ensure
  new_result_count: Result.count = n2 - n1 + 1
  -- original_characters: For every i in 1..n2-n1,
  --   Result.item (i) = item (n1+i-1)
feature -- Output
out: STRING
  -- Printable representation
  -- (From ANY.)
invariant
  irreflexive_comparison: not (Current < Current)
  empty_definition: is_empty = (count = 0)
  non_negative_count: count >= 0
end

```

A.6.22 CLASS *STD_FILES*

note

description: "[
 Commonly used input and output mechanisms. This class may be used as either ancestor or supplier by classes needing its facilities.
]"

class interface*STD_FILES***feature** -- Access

default_output: ? *FILE*
 -- Default output.

error: *FILE*
 -- Standard error file

input: *FILE*
 -- Standard input file

output: *FILE*
 -- Standard output file

standard_default: *FILE*
 -- *default_output* if not void,
 -- otherwise *output*.

feature -- Status report

last_character: *CHARACTER*
 -- Last character read by *read_character*

last_integer: *INTEGER*
 -- Last integer read by *read_integer*

last_real: *REAL*
 -- Last real read by *read_real*

last_string: *STRING*
 -- Last string read by *read_line*,
 -- *read_stream*, or *read_word*

feature -- Element change

put_boolean (*b*: *BOOLEAN*)
 -- Write *b* at end of default output.

put_character (*c*: *CHARACTER*)
 -- Write *c* at end of default output.

put_integer (*i*: *INTEGER*)
 -- Write *i* at end of default output.

put_new_line
 -- Write line feed at end of default output.

put_real (*r*: *REAL*)
 -- Write *r* at end of default output.

put_string (*s*: *STRING*)
 -- Write *s* at end of default output.

set_error_default
 -- Use standard error as default output.

set_output_default
 -- Use standard output as default output.

feature -- Input

read_character
 -- Read a new character from standard input.
 -- Make result available in *last_character*.

read_integer
 -- Read a new integer from standard input.
 -- Make result available in *last_integer*.

read_line
 -- Read a line from standard input.
 -- Make result available in *last_string*.
 -- New line will be consumed but not part of
last_string.

read_real
 -- Read a new real from standard input.
 -- Make result available in *last_real*.

read_stream (*nb_char*: *INTEGER*)
 -- Read a string of at most *nb_char* bound characters
 -- from standard input.
 -- Make result available in *last_string*.

to_next_line
 -- Move to next input line on standard input.

end

A.6.23 CLASS FILE

note

description: "[
Files viewed as persistent sequences of characters
]"

class interface

FILE

create

make (fn: STRING)

-- Create file object with *fn* as file name.

require

string_not_empty: **not** *fn*.is_empty

ensure

file_named: *name* ~ *n*

file_closed: *is_closed*

create_read_write (fn: STRING)

-- Create file object with *fn* as file name
-- and open file for both reading and writing;
-- create it if it does not exist.

require

string_not_empty: **not** *fn*.is_empty

ensure

exists: *exists*

open_read: *is_open_read*

open_write: *is_open_write*

open_append (fn: STRING)

-- Create file object with *fn* as file name
-- and open file in append-only mode.

require

string_not_empty: **not** *fn*.is_empty

ensure

exists: *exists*

open_append: *is_open_append*

open_read (fn: STRING)

-- Create file object with *fn* as file name
-- and open file in read mode.

require

string_not_empty: **not** *fn*.is_empty

ensure

exists: *exists*

open_read: *is_open_read*

open_read_write (fn: STRING)

-- Create file object with *fn* as file name
-- and open file for both reading and writing.

require

string_not_empty: **not** *fn*.is_empty

ensure

exists: *exists*

open_read: *is_open_read*

open_write: *is_open_write*

open_write (fn: STRING)

-- Create file object with *fn* as file name
-- and open file for writing;
-- create it if it does not exist.

require

string_not_empty: **not** *fn*.is_empty

ensure

exists: *exists*

open_write: *is_open_write*

feature -- Access

name: STRING

-- File name

feature -- Measurement

count: INTEGER

-- Size in bytes (0 if no associated physical file)

feature -- Status report

is_empty: BOOLEAN

-- Is structure empty?

end_of_file: BOOLEAN

-- Has an EOF been detected?

require

opened: **not** *is_closed*

exists: BOOLEAN

-- Does physical file exist?

is_closed: BOOLEAN

-- Is file closed?

is_open_read: BOOLEAN

-- Is file open for reading?

is_open_write: BOOLEAN

-- Is file open for writing?

is_plain_text: BOOLEAN

-- Is file reserved for text (character sequences)?

is_readable: BOOLEAN

-- Is file readable?

require

handle_exists: *exists*


```

is_writable: BOOLEAN
-- Is file writable?
require
  handle_exists: exists
last_character: CHARACTER
-- Last character read by read_character
last_integer: INTEGER
-- Last integer read by read_integer
last_real: REAL
-- Last real read by read_real
last_string: STRING
-- Last string read by read_line,
-- read_stream, or read_word
feature -- Status setting
close
-- Close file.
require
  medium_is_open: not is_closed
ensure
  is_closed: is_closed
open_read
-- Open file in read-only mode.
require
  is_closed: is_closed
ensure
  exists: exists
  open_read: is_open_read
open_read_append
-- Open file in read and write-at-end mode;
-- create it if it does not exist.
require
  is_closed: is_closed
ensure
  exists: exists
  open_read: is_open_read
  open_append: is_open_append
open_read_write
-- Open file in read and write mode.
require
  is_closed: is_closed
ensure
  exists: exists
  open_read: is_open_read
  open_write: is_open_write
open_write
-- Open file in write-only mode;
-- create it if it does not exist.
ensure
  exists: exists
  open_write: is_open_write
feature -- Cursor movement
to_next_line
-- Move to next input line.
require
  readable: is_readable
feature -- Element change
change_name (new_name: STRING)
-- Change file name to new_name
require
  file_exists: exists
ensure
  name_changed: name ~ new_name
feature -- Removal
delete
-- Remove link with physical file; delete physical
-- file if no more link.
require
  exists: exists
dispose
-- Ensure this medium is closed when
-- garbage-collected.
feature -- Input
read_character
-- Read a new character.
-- Make result available in last_character.
require
  readable: is_readable
--
require
  readable: is_readable
read_integer
-- Read the ASCII representation of a new integer
-- from file. Make result available in last_integer.
require
  readable: is_readable
read_line
-- Read a string until new line or end of file.
-- Make result available in laststring.
-- New line will be consumed but not part of
last_string.
require
  readable: is_readable

```

read_real

- Read the ASCII representation of a new real
- from file. Make result available in *last_real*.

require

readable: *is_readable*

read_stream (*nb_char*: *INTEGER*)

- Read a string of at most *nb_char* bound characters
- or until end of file.
- Make result available in *last_string*.

require

readable: *is_readable*

read_word

- Read a new word from standard input.
- Make result available in *last_string*.

feature -- Output*put_boolean* (*b*: *BOOLEAN*)

- Write ASCII value of *b* at current position.

require

extendible: *extendible*

put_character (*c*: *CHARACTER*)

- Write *c* at current position.

require

extendible: *extendible*

put_integer (*i*: *INTEGER*)

- Write ASCII value of *i* at current position.

require

extendible: *extendible*

put_real (*r*: *REAL*)

- Write ASCII value of *r* at current position.

require

extendible: *extendible*

put_string (*s*: *STRING*)

- Write *s* at current position.

require

extendible: *extendible*

invariant

name_not_empty: **not** *name*.*is_empty*

writable_if_extendible: *extendible* **implies** *is_writable*

end

A.6.24 CLASS *STORABLE***note**

```
description: "[
  Objects that may be stored and retrieved along with
  all their dependents
]"
```

```
usage: "[
  This class may be used as ancestor by classes needing
  its facilities.
]"
```

class interface*STORABLE***feature** -- Access

```
retrieved (file: FILE): STORABLE
  -- Retrieved object structure, from external
  -- representation previously stored in file.
  -- To access resulting object under correct type,
  -- use assignment attempt.
  -- Will raise an exception (code Retrieve_exception)
  -- if file content is not a STORABLE structure.
```

require

```
file_exists: file.exists
file_is_open_read: file.is_open_read
file_not_plain_text: not file.is_plain_text
```

feature -- Element change

```
basic_store (file: FILE)
  -- Produce on file an external representation of entire
  -- object structure reachable from current object.
  -- Retrievable within current system only.
```

require

```
file_exists: file.exists
file_is_open_write: file.is_open_write
file_not_plain_text: not file.is_plain_text
```

```
general_store (file: FILE)
```

```
  -- Produce on file an external representation of the
  -- entire object structure reachable from current
  -- object.
  -- Retrievable from other systems for same platform
  -- (machine architecture).
```

require

```
file_exists: file.exists
file_is_open_write: file.is_open_write
file_not_plain_text: not file.is_plain_text
```

```
independent_store (file: FILE)
```

```
  -- Produce on file an external representation of the
  -- entire object structure reachable from current
  -- object.
  -- Retrievable from other systems for the same or
  -- other
  -- platforms (machine architectures).
```

require

```
file_exists: file.exists
file_is_open_write: file.is_open_write
file_not_plain_text: not file.is_plain_text
```

end

A.6.25 CLASS *MEMORY*

note

description: "[
Facilities for tuning up the garbage collection
mechanism
]"

usage: "[
This class may be used as ancestor by classes needing
its facilities.
]"

class interface

MEMORY

feature -- Status report

collecting: *BOOLEAN*
-- Is garbage collection enabled?

feature -- Status setting

collection_off
-- Disable garbage collection.

collection_on
-- Enable garbage collection.

feature -- Removal

dispose
-- Action to be executed just before garbage collection
-- reclaims an object.
-- Default version does nothing; redefine in descendants
-- to perform specific dispose actions. Those actions
-- should only take care of freeing external resources
-- they should not perform remote calls on other objects
-- since these may also be dead and reclaimed.

full_collect
-- Force a full collection cycle if garbage
-- collection is enabled; do nothing otherwise.

end

A.6.26 CLASS *EXCEPTIONS***note**

description: "[
 Facilities for adapting the exception handling
 mechanism
]"

usage: "[
 This class may be used as ancestor by classes needing
 its facilities.
]"

class interface*EXCEPTIONS***feature** -- Access

developer_exception_name: *STRING*
 -- Name of last developer-raised exception

require

applicable: *is_developer_exception*

feature -- Access

Check_instruction: *INTEGER*
 -- Exception code for violated check

Class_invariant: *INTEGER*
 -- Exception code for violated class invariant

Incorrect_inspect_value: *INTEGER*
 -- Exception code for inspect value which is not one
 -- of the inspect constants, if there is no Else_part

Loop_invariant: *INTEGER*
 -- Exception code for violated loop invariant

Loop_variant: *INTEGER*
 -- Exception code for non-decreased loop variant

No_more_memory: *INTEGER*
 -- Exception code for failed memory allocation

Postcondition: *INTEGER*
 -- Exception code for violated postcondition

Precondition: *INTEGER*
 -- Exception code for violated precondition

Routine_failure: *INTEGER*
 -- Exception code for failed routine

Void_attached_to_expanded: *INTEGER*
 -- Exception code for attachment of void value
 -- to expanded entity

Void_call_target: *INTEGER*
 -- Exception code for feature call on void reference

feature -- Status report

assertion_violation: *BOOLEAN*
 -- Is last exception originally due to a violated
 -- assertion or non-decreasing variant?

exception: *INTEGER*
 -- Code of last exception that occurred

is_developer_exception: *BOOLEAN*
 -- Is the last exception originally due to
 -- a developer exception?

is_signal: *BOOLEAN*
 -- Is last exception originally due to an external
 -- event (operating system signal)?

feature -- Basic operations

die (*code*: *INTEGER*)
 -- Terminate execution with exit status *code*,
 -- without triggering an exception.

raise (*name*: *STRING*)
 -- Raise a developer exception of name *name*.

end

A.6.27 CLASS ARGUMENTS

note

description: "Access to command-line arguments"

usage: "[

This class may be used as ancestor by classes needing its facilities.

]"

class interface

ARGUMENTS

feature -- Access

argument (i: INTEGER): STRING

-- *i*-th argument of command that started system execution

-- (the command name if *i* = 0)

require

index_large_enough: i >= 0

index_small_enough: i <= argument_count

command_name: STRING

-- Name of command that started system execution

ensure

definition: *Result = argument (0)*

feature -- Measurement

argument_count: INTEGER

-- Number of arguments given to command that started

-- system execution (command name does not count)

ensure

non_negative: Result >= 0

end

A.6.28 CLASS *PLATFORM*

```

note
description: "Platform-dependent properties"
usage: "[
  This class may be used as ancestor by classes needing
  its facilities.
]"

class interface
  PLATFORM

feature -- Access

  Boolean_bits: INTEGER
    -- Number of bits in a value of type BOOLEAN
  ensure
    meaningful: Result >= 1

  Character_bits: INTEGER
    -- Number of bits in a value of type CHARACTER
  ensure
    meaningful: Result >= 1
    large_enough:  $2 \wedge \textit{Result} \geq$ 
      Maximum_character_code

  Integer_bits: INTEGER
    -- Number of bits in a value of type INTEGER
  ensure
    meaningful: Result >= 1
    large_enough:  $2 \wedge \textit{Result} \geq$  Maximum_integer
    large_enough_for_negative:  $2 \wedge \textit{Result} \geq -$ 
      Minimum_integer

  Maximum_character_code: INTEGER
    -- Largest supported code for CHARACTER values
  ensure
    meaningful: Result >= 127

  Maximum_integer: INTEGER
    -- Largest supported value of type INTEGER.
  ensure
    meaningful: Result >= 0

  Minimum_character_code: INTEGER
    -- Smallest supported code for CHARACTER values
  ensure
    meaningful: Result <= 0

  Minimum_integer: INTEGER
    -- Smallest supported value of type INTEGER
  ensure
    meaningful: Result <= 0

  Pointer_bits: INTEGER
    -- Number of bits in a value of type POINTER
  ensure
    meaningful: Result >= 1

```

```

Real_bits: INTEGER
  -- Number of bits in a value of type REAL
ensure
  meaningful: Result >= 1
end

```

A.6.29 CLASS *ONCE_MANAGER*

note

description: "[
Controller of keyed once routines
]"

usage: "[
See feature *onces* in class *ANY*.
]"

class interface

ONCE_MANAGER

feature -- Status report

fresh (*key*: *STRING*): *BOOLEAN*
-- Will the presence of *key* among a once routine's
-- once keys cause execution of the routine's body?

feature -- Element change

refresh (*key*: *STRING*)
-- Reset all once routines that use *key* as once key.

ensure

refreshed: *fresh* (*key*)

refresh_all

-- Reset all once routines.

refresh_all_except (*keys*: *ARRAY* [*STRING*])

-- Reset all once routines except those using
-- any of the items of *keys* as once keys.

refresh_some (*keys*: *ARRAY* [*STRING*])

-- Reset all once routines that use any
-- of the items of *keys* as once keys.

end

A.6.30 CLASS *ROUTINE*

note

description: "[
 Objects representing delayed calls to a routine,
 with some operands possibly still open
]"

deferred class interface

ROUTINE [*BASE_TYPE*, *OPEN_ARGS* → *TUPLE*]

feature -- Initialization

adapt (*other*: *ROUTINE* [*ANY*, *OPEN_ARGS*])
 -- Initialize from *other*.
 -- Useful in descendants.

feature -- Access

operands: *OPEN_ARGS*
 -- Open operands

target: *ANY*
 -- Target of call

open_operand_type (*i*: *INTEGER*): *INTEGER*
 -- Type of *i*-th open operand.

require

positive : *i* >= 1
 within_bounds: *i* <= *open_count*

hash_code: *INTEGER*
 -- Hash code value

precondition (*args*: **like** *operands*) *BOOLEAN*
 -- Do *args* satisfy routine's precondition
 -- in present state?

postcondition (*args*: **like** *operands*) *BOOLEAN*
 -- Does current state satisfy routine's
 -- postcondition for *args*?

feature -- Status report

callable: *BOOLEAN*
 -- Can routine be called on current object?

is_equal (*other*: **like** *Current*): *BOOLEAN*
 -- Is associated routine the same as the one
 -- associated with *other*?
 -- The object comparison operator ~ relies on this function.

valid_operands (*args*: *OPEN_ARGS*): *BOOLEAN*
 -- Are *args* valid operands for this routine?

feature -- Measurement

open_count: *INTEGER*
 -- Number of open parameters.

feature -- Element change

set_operands (*args*: *OPEN_ARGS*)
 -- Use *args* as operands for next call.

require

valid_operands: *valid_operands* (*args*)

feature -- Duplication

copy (*other*: **like** *Current*)
 -- Use same routine as *other*.

feature -- Basic operations

call (*args*: *OPEN_ARGS*)
 -- Call routine with operands *args*.

require

valid_operands: *valid_operands* (*args*)
callable: *callable*

apply is

-- Call routine with *operands* as last set.

require

valid_operands: *valid_operands* (*operands*)
callable: *callable*

deferred**end**

A.6.31 CLASS PROCEDURE

note

description: "[
 Objects representing delayed calls to a procedure,
 with some operands possibly still open
]"

comment: "[
 Features are the same as those of *ROUTINE*,
 with *apply* made effective, and no further
 redefinition of *is_equal* and *copy*.
]"

class interface

PROCEDURE [*BASE_TYPE*, *OPEN_ARGS* →
TUPLE]

feature -- Access

operands: *OPEN_ARGS*
 -- Open operands

target: *ANY*
 -- Target of call

open_operand_type (*i*: *INTEGER*): *INTEGER*
 -- Type of *i*-th open operand.

require

positive : *i* >= 1
 within_bounds: *i* <= *open_count*

hash_code: *INTEGER*
 -- Hash code value

feature -- Status report

callable: *BOOLEAN*
 -- Can procedure be called on current object?

is_equal (*other*: **like** *Current*): *BOOLEAN*
 -- Is associated procedure the same as the one
 -- associated with *other*?
 -- The object comparison operator ~ relies on this function.

valid_operands (*args*: *OPEN_ARGS*): *BOOLEAN*
 -- Are *args* valid operands for this procedure?

precondition (*args*: **like** *operands*) *BOOLEAN*
 -- Do *args* satisfy procedure's precondition
 -- in present state?

postcondition (*args*: **like** *operands*) *BOOLEAN*
 -- Does current state satisfy procedure's
 -- postcondition for *args*?

feature -- Measurement

open_count: *INTEGER*
 -- Number of open parameters.

feature -- Element change

set_operands (*args*: *OPEN_ARGS*)
 -- Use *args* as operands for next call.

require

valid_operands: *valid_operands* (*args*)

feature -- Duplication

copy (*other*: **like** *Current*)
 -- Use same procedure as *other*.

feature -- Basic operations

call (*args*: *OPEN_ARGS*)
 -- Call procedure with operands *args*.

require

valid_operands: *valid_operands* (*args*)
callable: *callable*

apply is
 -- Call procedure with *operands* as last set.

require

valid_operands: *valid_operands* (*operands*)
callable: *callable*

end

A.6.32 CLASS *FUNCTION***note**

description: "[
 Objects representing delayed calls to a function,
 with some operands possibly still open
]"

comment: "[
 Features are the same as those of *ROUTINE*,
 with *apply* made effective, and the addition
 of *last_result* and *item*.
]"

class interface

FUNCTION [*BASE_TYPE*,
OPEN_ARGS → *TUPLE*, *RESULT_TYPE*]

feature -- Access

last_result: *RESULT_TYPE*
 -- Result of last call, if any.

require

valid_operands: *valid_operands* (*args*)
callable: *callable*

operands: *OPEN_ARGS*
 -- Open operands

target: *ANY*
 -- Target of call

open_operand_type (*i*: *INTEGER*): *INTEGER*
 -- Type of *i*-th open operand.

require

positive : *i* >= 1
 within_bounds: *i* <= *open_count*

hash_code: *INTEGER*
 -- Hash code value

precondition (*args*: **like** *operands*) *BOOLEAN*
 -- Do *args* satisfy function's precondition
 -- in present state?

postcondition (*args*: **like** *operands*) *BOOLEAN*
 -- Does current state satisfy function's
 -- *postcondition* for *args*?

feature -- Status report

callable: *BOOLEAN*
 -- Can function be called on current object?

is_equal (*other*: **like** *Current*): *BOOLEAN*
 -- Is associated function the same as the one
 -- associated with *other*?
 -- The object comparison operator ~ relies on this function.

valid_operands (*args*: *OPEN_ARGS*): *BOOLEAN*
 -- Are *args* valid operands for this function?

feature -- Measurement

open_count: *INTEGER*
 -- Number of open parameters.

feature -- Element change

set_operands (*args*: *OPEN_ARGS*)
 -- Use *args* as operands for next call.

require

valid_operands: *valid_operands* (*args*)

feature -- Duplication

copy (*other*: **like** *Current*)
 -- Use same function as *other*.

feature -- Basic operations

call (*args*: *OPEN_ARGS*)
 -- Call function with operands *args*.

require

valid_operands: *valid_operands* (*args*)
callable: *callable*

apply is

-- Call function with *operands* as last set.

require

valid_operands: *valid_operands* (*operands*)
callable: *callable*

item (*args*: **like** *operands*)

-- Result of calling function with *args* as operands

require

valid_operands: *valid_operands* (*operands*)
callable: *callable*

ensure

set_by_call: *Result* = *last_result*

end

A.6.33 CLASS PREDICATE

note

description: "[
 Objects representing delayed calls to boolean-valued
 function, with some operands possibly still open
]"

inheritance: "[
 This class inherits (see section [A.5.17](#)) from
 FUNCTION [BASE_TYPE, OPEN_ARGS,
 BOOLEAN]
]"

comment: "[
 Features are the same as those of *FUNCTION*,
 with *RESULT_TYPE* replaced by *BOOLEAN*,
 and no further redefinition of *is_equal* and *copy*.
]"

class interface

PREDICATE [BASE_TYPE, OPEN_ARGS → TUPLE]

feature -- Access

last_result: RESULT_TYPE
 -- Result of last call, if any.

require

valid_operands: valid_operands (args)
callable: callable

operands: OPEN_ARGS
 -- Open operands

target: ANY
 -- Target of call

open_operand_type (i: INTEGER): INTEGER
 -- Type of *i*-th open operand.

require

positive : *i* >= 1
 within_bounds: *i* <= open_count

hash_code: INTEGER
 -- Hash code value

precondition (args: **like** operands) BOOLEAN
 -- Do args satisfy function's precondition
 -- in present state?

postcondition (args: **like** operands) BOOLEAN
 -- Does current state satisfy function's
 -- postcondition for args?

feature -- Status report

callable: BOOLEAN
 -- Can function be called on current object?

is_equal (other: **like** Current): BOOLEAN

-- Is associated function the same as the one
 -- associated with other?
 -- The object comparison operator ~ relies on this function.

valid_operands (args: OPEN_ARGS): BOOLEAN
 -- Are args valid operands for this function?

feature -- Measurement

open_count: INTEGER
 -- Number of open parameters.

feature -- Element change

set_operands (args: OPEN_ARGS)
 -- Use args as operands for next call.

require

valid_operands: valid_operands (args)

feature -- Duplication

copy (other: **like** Current)
 -- Use same function as other.

feature -- Basic operations

call (args: OPEN_ARGS)
 -- Call function with operands args.

require

valid_operands: valid_operands (args)
callable: callable

apply is

-- Call function with operands as last set.

require

valid_operands: valid_operands (operands)
callable: callable

item (args: **like** operands)

-- Result of calling function with args as operands

require

valid_operands: valid_operands (operands)
callable: callable

ensure

set_by_call: Result = last_result

end

PART IV: THE LACE CONTROL LANGUAGE

This fourth part of the book contains a single chapter devoted to the description of Lace (*Language for the Assembly of Classes in Eiffel*), a simple Eiffel-like control language used to construct actual executable systems out of Eiffel classes by specifying the files where Eiffel classes reside, the compilation options to be used, the external (non-Eiffel) software elements to be included, and any other control information that the compiler and other tools may need to assemble a system from its components.

B

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of “Eiffel: The Language”. Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

Specifying systems in Lace *(in progress)*

B.1 OVERVIEW

As you start producing clusters of classes, you will expect the supporting environment to provide language processing tools — compilers, interpreters, documenters, browsers — to process these classes and assemble them into systems.

These tools will need a specification of where to find the classes and what to do with them. Such a specification is called an **Assembly of Classes in Eiffel**, or Ace for short. This appendix presents a notation, the **Language for Assembling Classes in Eiffel**, or Lace, for writing Aces. Although Lace is separate from Eiffel, Eiffel environments must support it.

Two words of encouragement if you feel that after reaching page [1017](#) of the description of Eiffel you should not have to learn yet another language. First, Lace is very much Eiffel-like, so you’ll find yourself treading familiar ground in this chapter. But more importantly, for anything other than advanced uses of Eiffel you don’t need to study the details of Lace — the way you would study a design or programming language — since you may expect an Eiffel environment to provide an interactive tools that lets you fill out your project’s specific needs, provides defaults for everything else, and generates the Ace for you as a result. ISE Eiffel, for example, provides a graphical **Project Wizard** that does all this.



For a good understanding of what’s going on behind the scenes it is useful to have a basic understanding of Lace. You can obtain it by reading the basic Ace example of the next section. What comes after that is detailed reference and may be skipped on first reading.

If you are familiar with Lace basics and simply need a reminder on some details, you may find it profitable to use the [complete example](#) of a later section. You will find the [complete Lace grammar](#) at the end of this chapter.

B.2 A SIMPLE EXAMPLE

Typical elements of an Ace include information about directories and files containing the text of the system's clusters and classes, compilation options (assertion monitoring, debugging etc.) for the classes involved, name of the root class (used to start off execution), location of non-Eiffel elements such as external libraries, target file for the compilation's output.

To help you get quickly a idea of the basic concepts of Lace, here is a simple but typical Ace.

Although simple, this example includes the Lace facilities that suffice for many practical Eiffel systems.

```

system browser root
    EB (browsing)
default
    assertions (ensure); trace (no)
    collect (yes); debug (no)
cluster
    "$INSTALLATION/library/support"
    "$INSTALLATION/library/parsing"

    browsing: "tilda/current/browser"
        default
            assertion (all)
        option
            debug (yes): LAYOUT, FUNCTIONS
        end
    end

```

This describes a system called browser. The root of this system is a class called EB. The text of EB is to be found in cluster *browsing* (described a few lines below in the Ace); this mention of the root class's cluster, in parentheses, is optional if the entire system has only one class of the name given, here EB.

Default compilation options for the classes of this system are: for assertions, check postconditions (**ensure** clauses), which also implies checking preconditions; do not trace execution; enable garbage collection (*collect*); do not execute **debug** instructions.

These options are system-wide defaults; individual clusters may override them through their own **default** clauses, as does cluster *browsing*. Individual classes may also override the system and cluster defaults through the **option** clause.

The Clusters part, beginning with the **cluster** Lace keyword (reminiscent of the **feature** keyword introducing a Features part in Eiffel), defines the set of clusters; clusters, as you **remember**, are groups of classes, and the system's classes are collected from the classes of its clusters. ← *Chapter 3 introduced the structure of systems and the notion of cluster.*

The specification of the first two clusters only gives directory names, each written as a **Manifest_string**. By default, the cluster consists of all classes to be found in the files having names ending with **.e** in this directory. Each one of these files may contain one or more classes. The **.e** name convention is the default; we shall see below how to include files with other names, or to exclude some **.e** files.

A **Manifest_string** ef

The first two clusters are elements of the Basic Libraries (support and parsing). Their names use Unix-like conventions for environment variables (such as **\$INSTALLATION**) to facilitate using the same Ace on different machines. Clearly, such conventions are operating-system-dependent.

The last cluster also has a directory name (this is always required), preceded here by a **Cluster_name**, *browsing*, and a colon. You will need to include such a **Cluster_name** whenever other elements of the Ace refer to the cluster: here, for example, the Root clause refers to cluster *browsing* through its **Cluster_name**, to indicate that this is where the root class EB is.

For this cluster, the default assertion monitoring option, overriding the default specified at the system level, is **all** (monitor everything). Furthermore, the *debug* option is enabled for two classes of the cluster, **LAYOUT** and **FUNCTIONS**.

This example is typical of Aces used to assemble and compile systems without any advanced options.

B.3 ON THE ROLE OF LACE

Before showing the remaining details of Lace, it is important to ponder briefly over the connection of this description to the rest of this book.

Lace support, it was mentioned above, is not a required element of an Eiffel implementation. Why then talk about Lace at all as part of a specification of the Eiffel language? There are two reasons, one pedagogical and one practical.

The pedagogical reason is that since some Lace-like mechanisms, at least elementary ones, will be necessary anyway to execute your software, you would not get a full picture of Eiffel software development without some understanding of possible assembly mechanisms.

the preceding two sections are probably sufficient to get a general idea of the purpose of Lace, but the rest of this appendix will give more details for those readers who are seriously interested. As mentioned already, these details are not essential on first reading, hence the `SHORTCUT` sign which signals the rest of this appendix as non-crucial material.

The practical reason for paying attention to Lace involves **portability**, and should be of particular concern to authors of Eiffel implementations.

True, because of the variety of possible implementation platforms (hardware, operating systems, user interfaces) and of possible implementation techniques such as interpretation, compilation to machine code, compilation to an intermediate assembly-like code such as C etc., one may not guarantee total portability or enforce a fully general Lace standard. For one thing, an implementation could altogether bypass text-based descriptions such as those of Lace, in favor of interactive input of compilation and assembly options (with a modern graphical or “point and click” user interface); then it would have no need for a description *language* in the textual sense of this word, even though it will still provide the Lace semantics — specification of compilation options, class text location etc. — in some other way.

Even if system descriptions use a textual form, an individual implementation may have non-portable characteristics, stemming for example from the peculiarities of the file and directory system of the underlying platform, or from specific optimization options provided by the implementor.

Along with such non-portable aspects, however, certain facilities will be needed in every implementation. For example, it is necessary to let developers specify whether or not to monitor the assertions of any given class at run time. Then everyone will benefit if all implementations using text-based system descriptions rely on a common set of notations and conventions. This does not guarantee full portability, but avoids unjustified sources of non-portability.

The design of Lace is a result of these considerations. It suggests a default notation for the standard components of system descriptions, while leaving individual implementors the freedom to add platform-dependent or implementation-specific facilities.

B.4 A COMPLETE EXAMPLE

For ease of reference (especially meant for those readers who already know the basics of Lace but are coming back to this presentation for a quick and informal reminder on the form of some clauses), here is a complete Ace using most of the available possibilities. The various components are explained in subsequent sections.

Because it illustrates all the major Lace facilities, this Ace is more complex than most usual ones, which tend to use the basic facilities illustrated by the simple example on page

```

system browser root
    EB (browsing)
default
    assertions (ensure)
    trace (no)
    collect (yes)
    debug (no)
cluster
    "$INSTALLATION/library/support"
    basics: "$INSTALLATION/library/structures"
    parsing: "$INSTALLATION/library/parsing"
    browsing: "not/current/browser"
    use
        ".lace"
    include
        "commands"
    exclude
        "g.t.e"
    default
        debug ("level2"); debug ("io_check")
    option
        assertion (all): CONSTANTS, FULL_TEXT
        trace: FUNCTIONS, QUIT, RENAMED
        debug (yes): LAYOUT, FUNCTIONS
        debug ("format"): FULL_TEXT, OUTPUT
        debug ("numerical_accuracy"): OUTPUT
    visible
        CONSTANTS as BROWSING_CONSTANTS
        LAYOUT
        EB
        creation
            initialize
        export
            execute, set_target, initialize
        end
end

```

```

external
    Object: "object_name.o", "../basics.o",
           "-ltermcap", "otherlib.a"
    C: "previous.h", "/usr/$MACHINE/src/scree-.c"
    Make: "../Clib/makefile"
generate
    executable: "$INSTALLATION/bin"
    C (yes): "$INSTALLATION/src/browser/package/eb.c"
    Object (no): "$INSTALLATION/bin/browser"
end

```

B.5 BASIC CONVENTIONS

Let us now proceed to the details of Lace.

You should not have been too surprised by the syntax, which is Eiffel-like. The syntax descriptions below use the conventions applied to Eiffel throughout the rest of this book. ← Chapter 2 introduced the conventions for syntax description.

Comments, as in Eiffel, begin with two consecutive dashes -- and extend to the end of the line.

The grammar of Lace also uses some of the same basic components as Eiffel:

- Identifier, such as `A_CLASS_NAME`
- Manifest string, such as `"A STRING$"`
- Integer constant, such as `-4562`

← "IDENTIFIERS",
32.12, page 891;
"MANIFEST
STRINGS", 29.8, page
794; "INTEGERS",
32.16, page 899.

As in Eiffel, letter case is not significant for identifiers. The recommended standard is to use upper case for class names and lower case for everything else. Letter case is also not significant for strings except when they refer to outside elements such as file names, directory names or linker options; such strings will be passed verbatim to outside tools (such as the operating system or linker), which may or may not treat letter case as significant.

Lace has the following keywords, which you may not use as identifiers:

adapt	all	as	check	cluster	creation	default	end	ensure
exclude	export	external	generate	ignore	include	invariant	keep	loop
no	option	require	rename	root	system	use	visible	yes

An important convention applied throughout the Lace syntax is that an Identifier is syntactically legal wherever a Manifest string is, and conversely. For this purpose, the grammar productions given below do not refer directly to these two constructs, but use the construct Name, defined as

Name \triangleq Identifier | Manifest_string

As a consequence, if your system contains a class called CLUSTER, which is not a valid Lace identifier since it conflicts with one of the keywords in the above list, you may still refer to it in the Ace by using the Manifest_string "CLUSTER". Similarly, although you may give a simple file name such as *my_file* as an identifier, one which does not conform to Lace identifier conventions, such as "tilda/directory/*my_file*", will have to be expressed as a string.

For clarity, all the examples of this presentation use strings for file and directory names.

A consistency condition applies to names used in an Ace: the Cluster_name must be different for each cluster. It is valid, however, to use the same identifier in two or more of the roles of Cluster_name, System_name, Class_name.

ACE STRUCTURE

The structure of an Ace is given by the following grammar.

All clauses were present in the long example above; the earlier, shorter example had all clauses except Externals and Generation.

The Defaults clause gives general options which apply to all classes in the system, except where overridden by cluster defaults or options specified for individual classes. It may also indicate options that apply to the system as a whole; for example, the option

collect (yes)

requests garbage collection to be turned on; this only makes sense for the whole system. The precise form of options is explained below.

→ "*SPECIFYING OPTIONS*", B.9, page 1028.

The Clusters part lists individual clusters and the associated options.

The Externals clause gives information about any non-Eiffel software element needed to assemble the system.

The Generation clause indicates where to store the output of system assembly and compilation (executable module, object code, code in another target language). By default the output will be produced in the directory where the compilation command is executed.

The order of these clauses should be easy to remember: first you give the system a name (System) and express where it starts its execution (Root); then you specify the options that apply across the board, except where specifically overridden (Defaults); you list the Clusters that make up the system's universe; you indicate what else is needed, beyond Eiffel clusters, to assemble the system (Externals); finally, you indicate where the outcome of the assembly and compilation process must be generated (Generation).

The next sections study the various clauses of an Ace.

B.6 BASICS OF CLUSTER CLAUSES

In an Ace containing a **Clusters** part, the keyword **cluster** will be followed by zero or more **Cluster_clause**, each specifying the location in the file system of one of the clusters of the universe, and the properties applying to the classes of that cluster.

Let us examine the possibilities by writing a **Cluster_clause** through successive additions showing most of the available possibilities.

In its simplest form, a **Cluster_clause** is simply a **Directory_name**, expressed as a **Manifest_string**, as in:

On some operating systems, directories may be called differently (for example “folders”) or replaced by some other mechanism.

```
"$INSTALLATION/library/browsing"
```

If you must refer to the cluster in other clauses of the Ace, you will need to give it a **Cluster_name**. (This will be the name for Lace, and is distinct from the cluster's name for the operating system, which appears as the **Directory_name**.) The **Cluster_name** will precede the **Directory_name**, separated by a colon. If you want to call the above cluster *browsing*, you will declare it as

```
browsing: "$INSTALLATION/library/browsing"
```

An optional **Cluster_properties** part may then appear, specifying further properties of the cluster. It may contain the following paragraphs, all optional, in the order given: **Use**, **Include**, **Exclude**, **Name_adaptation**, **Defaults**, **Options** and **Visible**. If present, the **Cluster_properties** part is terminated by an **end** (and, as with an Eiffel routine, a suggested comment repeating the cluster name).

Here is the syntax of the **Clusters** and **Cluster_properties** parts:

```

Clusters  $\triangleq$  cluster {Cluster_clause ";" ...}
Cluster_clause  $\triangleq$  [Cluster_tag]
                Directory_name
                [Cluster_properties]
Cluster_tag  $\triangleq$  Cluster_name ":"
Directory_name  $\triangleq$  Name
                Cluster_properties
                [Use]
                [Include]
                [Exclude]
                [Name_adaptation]
                [Defaults]
                [Options]
                [Visible]
                end

```

The following sections explore the various **Cluster_properties** paragraphs. If, in the meantime, you fear that you might forget the order of paragraphs in a **Cluster_properties** part, remember the following simple principle: the order is the natural one from the point of view of a language processing **tool** that must process the cluster. For example, a compiler which uses a **Cluster_properties** specification to compile the classes a cluster, and has already obtained any default specifications associated with the cluster (through the **Use** paragraph), will take the following actions:

As mentioned earlier, developers should produce an Ace by completing a pre-filled template, rather than from scratch. The template will have the paragraphs in the right order.

Find any files to take into account besides the default (**Include**).

Discard any unneeded files (**Exclude**).

To prepare for compiling the class texts, find out if any class name appearing there actually refers to a class having another name (**Name_adaptation**).

Find out the cluster-level compilation options (**Defaults**). and start compilation of the cluster's various classes.

When compiling a given class, find out if a specific option applies to it (**Options**).

Having compiled classes, decide which ones of their properties, if any, must be made available to other systems (**Visible**).

B.7 STORING PROPERTIES WITH A CLUSTER

The `Cluster_properties` part may begin with a `Use` paragraph, as in

```
browsing: "~/current/browser"
  use
    "Ace.mswin"
  end
```

to indicate that the cluster's directory contains a "Use file" (here of name `.lace`) containing the specification of some of the cluster's properties. The content of a Use file must itself be a `Cluster_properties` conforming to the Lace syntax. This makes it possible to specify cluster properties (for example compilation options) in a file that remains stored with the cluster itself.

In the above example, the `Cluster_properties` part for cluster `browsing` in the Ace has no further paragraphs beside `Use`, so all the cluster properties for `browsing` will be taken from the Use file. In the examples that follow, however, the Ace will contain other paragraphs for `browsing`, such as `Include`, `Exclude` or `Options`. In such a case the properties specified in the Ace are added to those of the Use file, and they take precedence in case of conflict.

It is a general Lace principle that whenever two comparable properties may apply (here a property specified in a Use file, and a property specified in the Ace after the Use paragraph) the one appearing last is added to the first or, in case of conflict, overrides it.

Here is the syntax of the optional `Use` paragraph of a `Cluster_properties` part:

```
Use  $\triangleq$  use File
File  $\triangleq$  Name
```

The `Cluster_properties` part contained in a Use file may itself contain a `Use` paragraph.

B.8 EXCLUDING AND INCLUDING SOURCE FILES

The next two optional `Cluster_properties` paragraphs, `Include` and `Exclude`, serve to request the explicit inclusion or exclusion of specific source files. Two important applications are overriding the default naming convention for files containing class texts, and selecting non-standard versions of a library class.

By default, when you list a cluster as part of a system, this includes all the class texts contained in files having names of a certain standard form in the cluster's directory; normally this standard form is *xxx.e* for any string *xxx*, although certain platforms may have different conventions (for example if periods are not legal characters in file names). The rest of this presentation assumes the *xxx.e* convention.

.IY

A *xxx.e* file may contain one or more classes, written consecutively. It is often a good idea to have just one class per file, with the *xxx* part of the file name being the lower-case version of the **Class_name**; for example file *cursor.e* would contain the source text for class CURSOR. In some cases, however, you may wish to group the texts of a few small and closely related classes in a single file.

The *xxx.e* convention or its equivalent is only the default. You may wish to remove from consideration a file with a name of this form (because you do not want to include the corresponding classes in your system, or simply because the file contains non-Eiffel text); conversely, you may wish to add to the cluster some classes residing in files having non-conforming names. The **exclude** and **include** clauses achieve this.

Here is a typical use, which excludes file *g.t.e* and includes two files with non-standard names:

```
browsing: "~/current/browser"
use
    "Ace.mswin"
include
    "commands"
    "states"
exclude
    "g.t.e"
end
```

You may also apply the **Exclude** facility when you wish a class from a certain cluster to override a class from another cluster. If you exclude a file containing a class of name C, and another cluster contains a class with the same name, this class will override the original C. This is useful in particular if you wish to replace a library class by your own version. Assume for example you want to use your own version of **ANY**, the universal class serving as ancestor to all developer-defined classes. You may achieve this by storing the new version in one of your clusters and excluding the default one (assumed to be in file *any.e* in cluster *default*):

← See chapter 35 about class **ANY**.

```
default: "/usr/local/Eiffel/library/kernel"
  exclude
    "any.e"
  end
```

Here is the syntax of the Include and Exclude optional paragraphs of a **Cluster_properties** part:

```
Include ≙ include File_list
Exclude ≙ exclude File_list
File_list ≙ {File ";" ... }
```

B.9 SPECIFYING OPTIONS

Option values govern actions of the tools that will process the Ace; for example they may affect compilation, interpretation or linking.

An option specification may appear in any of the following three Ace components, all optional:

The Ace-level **Defaults** clause.

The Defaults paragraph of a **Cluster_properties** part.

The Options paragraph of a **Cluster_properties** part.

In the last two cases, the **Cluster_properties** may be in the Ace itself or in the Use file for one of its clusters.

If two or more conflicting values are given for an option, the last overrides any preceding ones. This means that values in the Options paragraph override cluster-level **Defaults** values, which override Ace-level Defaults values, and that a value in any of these components overrides any preceding value in the same component.

Here is a specimen of an Ace-level **Defaults** clause already shown above:

```
default
  assertions (ensure); trace (no)
  collect (yes); debug (no);
```

This example enables options as indicated. It is also acceptable as a cluster-level **Defaults**, except for the presence of **collect** (enabling garbage collection), which may only be given at the Ace-level since garbage collection applies to an entire system.

To get an example of cluster-level **Defaults** and **Options**, let us extend our *browsing* **Cluster_clause** example:

```
browsing: "~/current/browser"
  use
    "Ace.mswin"
  include
    "commands"
    "states"
  exclude
    "g.t.e"
  default
    debug ("level2");
    debug ("io_check")
  option
    assertion (all): CONSTANTS, FULL_TEXT;
    trace: FUNCTIONS, QUIT, RENAMED;
    debug (yes): LAYOUT, FUNCTIONS;
    debug ("format"): FULL_TEXT, OUTPUT;
    debug ("numerical_accuracy"): OUTPUT
  end
```

The **Defaults** paragraph overrides any Ace-level default for the **debug** option by enabling execution of **Debug_instructions** in routines of classes of the cluster, for the **Debug_key** *level2* and the **Debug_key** *io_check*. ← [“THE DEBUG INSTRUCTION”, 17.8, page 497.](#)

The **Options** paragraph in turn overrides all preceding **Defaults**. The syntactic structure of is the same as for a **Defaults** paragraph, except that here every **Option_tag** (and optional **Option_value** in parentheses) may be followed by a **Target_list**, beginning with a colon, which lists one or more **Name**; these must be the names of classes in the cluster. In that case the option given overrides the default only for the classes given.

If there is no **Target_list**, the option applies to all classes in the cluster.

Here is the syntax of Options and Defaults paragraphs:

```

Defaults ≙ default {Option_clause ";" ...}
Options ≙ option {Option_clause ";" ...}
Option_clause ≙ Option_tag [Option_mark] [Target_
list]
Target_list ≙ ":" {Class_name "," ...}"" sup +
Option_tag ≙ Class_tag System_tag
System_tag ≙ collect Free_tag
Class_tag ≙ assertion | debug | optimize | trace |
Free_tag
Free_tag ≙ Name
Option_mark ≙ "(" Option_value ")"
Option_value ≙ Standard_value | Class_value
Standard_value ≙ yes | no | all | Free_value
Class_value ≙ require | ensure | invariant |
loop | check |
Free_value
Free_value ≙ File_name |
Directory_name |
Name

```

A **Target_list** may only appear in an Options paragraph, not in a **Defaults** paragraph. A **System_tag** may only appear in an Ace-level **Defaults** clause.

The syntax permits only one **Option_value**, not a list of values, after an **Option_tag**. You may obtain the effect of multiple values by repeating the same **Option_tag** with different values, as was done in the example with the lines

```

debug ("format"): FULL_TEXT, OUTPUT
debug ("numerical_accuracy"): OUTPUT

```

which imply enabling the debug option for class *OUTPUT* both for the **Debug_key** *format* and for the **Debug_key** *numerical_accuracy*. In case of conflict, as usual, the last value given overrides any preceding ones.

This syntax shows that for an `Option_tag` as well as an `Option_value` you may use not just predefined forms (such as `assertion` for an `Option_tag` and `no` for an `Option_value`) but also `Free` forms, each of which is defined just as a `Name` (`Identifier` or `Manifest_string`). This means that along with general-purpose options which are presumably of interest to all implementations of Eiffel (level of assertion monitoring, garbage collection etc.), individual implementors may add their own specific options.

The predefined possibilities for `Option_tag` (`collect`, `assertion` etc.) are not Lace keywords, and so may be used as identifiers in an Ace. The predefined possibilities for `Standard_value`, however, are keywords; they appear in bold italics (`yes`, `require` etc.). Remember that you can always use a `Manifest_string` (such as `"YES"` or `"DEFAULTS"`) to write a Lace name, for example the name of a class in the system, which conflicts with a keyword.

When the predefined forms are supported, they should satisfy the constraints and produce the effects summarized in the following table.

Option	Governs	Possible values	Default	Scope
<code>assertion</code>	Level of assertion monitoring and execution of <code>Check</code> instructions	<code>no</code> , <code>require</code> , <code>ensure</code> , <code>invariant</code> , <code>loop</code> , <code>check</code> , <code>all</code> . Monitoring at each level in this list also applies to the subsequent levels (<code>ensure</code> implies precondition checking etc.). Value <code>invariant</code> means class invariant; <code>loop</code> means monitoring of loop invariants and of loop variant decrease; <code>check</code> adds execution of <code>check</code> instructions; <code>all</code> means same as <code>check</code> .	<code>require</code>	
<code>collect</code>	Garbage collection	<code>no</code> , <code>yes</code> .	<code>yes</code>	Entire system
<code>debug</code>	Execution of <code>Debug</code> instructions	<code>no</code> , <code>yes</code> , <code>all</code> or a <code>Name</code> representing a <code>Debug_key</code> . <code>yes</code> means same as <code>all</code> .	<code>no</code>	

optimize	Optimize generated code.	no , yes , all , or a Name representing specific optimization level offered by compiler. In Defaults or Options clause for a given cluster, yes governs class-level optimization and all means same as yes . In Ace-level Defaults clause, yes governs system-wide optimization, and all means same as yes plus class-level optimization.	no	
trace	Generate run-time tracing information.	no , yes or all . yes means the same as all		

B.10 SPECIFYING EXTERNAL ELEMENTS

To assemble a system you may need “external” elements, written in another language or available in object form from earlier compilations. The **Externals** clause serves to list these elements.

Here is an example **Externals** clause:

```
external
  Object:
    "object_name.o"; "../basics.o"
    "-ltermcap"; "otherlib.a"
  C: "previous.h"; "/usr/$MACHINE/src/scree-.c"
  Make: "../Clib/makefile"
```

Such a clause contains one or more **Language_contribution**, each being relative to a certain **Language**. Every **Language** is given by an **Identifier**, such as:

Object: object code, produced by a compiler for some language, to be linked with the result of system compilation or included for interpretation.

Remember that letter case is not significant, so that “FORTRAN” and “make” would also be permitted. **Make** is a Unix tool, with equivalents on many other operating systems, which works from a dependency list, or **Makefile**, to recompile or reconstruct software. **Make** and **makefiles** are normally not needed for Eiffel classes, but may be needed for external non-Eiffel software.

Ada: Ada language elements.

Pascal: Pascal language elements.

Fortran: Fortran language elements.

C: C language elements.

Make: Descriptions of dependencies needed to recompile non-Eiffel software elements.

The exact list of supported Language possibilities depends on the implementation.

In each `Language_contribution`, the `Language` is followed by a semicolon and a list of File names containing the corresponding elements.

The syntax of the `Externals` clause is the following.

```

Externals  $\triangleq$  external Language_specifics
Language_specifics  $\triangleq$  {Language_contribution ";" ...}
Language_contribution  $\triangleq$  Language ":" File_list
Language  $\triangleq$  Eiffel | Ada | Pascal |
Fortran | C | Object | Make |
Name

```

The predefined language names (`Eiffel`, `Ada` etc.) are not Lace keywords, and so may be used as identifiers in an Ace.

B.11 ONCE CONTROL

[To be filled in. Remember to update the “complete example” to include this possibility.]

B.12 GENERATION

The Generation clause indicates what output, if any, should be generated by the assembly process, and where that output should be stored.

A specimen of the clause is:

```

generate
  Executable: "$INSTALLATION/bin"
  C (yes): "$INSTALLATION/src/browser/package"
  Object (no): "$INSTALLATION/bin/M_68040/eb"

```

This Generation clause requests generation of both an executable module and a C package containing the translation of the original Eiffel. Clearly, although any Eiffel environment which is not solely meant for analysis or design will support *executable* generation, the availability of any other target language is implementation-dependent.

ISE's Eiffel compiler generates executable code as well as C packages (including a copy of the run-time system, a Make file and all other elements needed to compile and run the result). The C package generation mechanism provides support for cross-development.

The generate Target, coming after the colon, is either a Directory, as in the first example, or a File, as in the second. If it is a directory, the output will be stored in a file of that directory; the name of that file will normally be the *System_name*, here *browser*. The tools may also use both the *System_name* and the name of the root's chosen creation procedure to make up the name of the executable output file.

The Language name (*Executable*, *C* or Object in the example) may be followed by a *Generate_option_value*, **yes** or **no**, in parentheses. The absence of this component, as in the first two cases of the example, is equivalent to (**yes**). The last line requests that no *Object* package be generated. The Ace's author may re-enable *Object* generation simply by replacing **no** by **yes**.

Here is the syntax of the Generation clause:

```

Generation  $\triangleq$  generate Generation_clauses
Generation  $\triangleq$  {Language_generation ";" ...}
Language_generation  $\triangleq$  Language [Generate_option] ":" Target
Generate_option  $\triangleq$  "(" Generate_option_value ")"
Generate_option_value  $\triangleq$  yes | no
Target  $\triangleq$  Directory | File

```

B.13 VISIBLE FEATURES

As you generate output from a system, you may want to make some of the system's classes available to external software elements that will create instances of these classes (through creation procedures) and apply features to those instances (through exported features).

.IP

Using the Visible paragraph of a *Cluster_properties* part, you may indicate which classes of the cluster must be externally visible; this will apply by

default to all the creation procedures and exported features of these classes, but you may also request external visibility for some of them only. Furthermore, you may make some of them externally available under names which are different from their original names in the class text, for example if they are to be called from a language whose identifier conventions differ from those of Eiffel.

Some external software may also need to refer to the class name itself; this is the case with *eif_proc* and similar functions from the Cecil library, which obtain a routine pointer. If the Eiffel name of the class is not appropriate for this purpose (in particular when it would cause ambiguity), you may define a different external class name. ← [“THE CECIL LIBRARY”, 31.16, page 865.](#)

Here is a Visible added to our example, browsing cluster extended with a Visible paragraph, requesting external visibility for three classes of the cluster, *CONSTANTS*, *LAYOUT* and *EB*:

```

browsing: "~/current/browser"
... use, include, exclude, adapt, default, option as before...
visible
    CONSTANTS as BROWSING_CONSTANTS;
    LAYOUT
        rename
            choice_menu as "choice.menu",
            set_reverse as "set.reverse"
        end
    EB
        create
            initialize
        export
            execute, set_target, initialize
        rename
            set_target as "set.target"
        end
    end

```

For *CONSTANTS*, you have defined a different external class name, *BROWSING_CONSTANTS*, for use by external software such as Cecil functions. ← [“THE CECIL LIBRARY”, 31.16, page 865.](#)

For *CONSTANTS* and *LAYOUT*, external software can create objects using all the creation procedures of these classes (if any), and call all exported features on these objects. Two features of *LAYOUT* are available to external software (for creation or call) under names different from their Eiffel names, making them callable from a language which prohibits underscores *_* in identifiers.

For EB, feature *set_target* is also externally renamed. In addition, you have only requested external availability for specific features of EB: among creation procedures, you only need initialize to be externally available for object creation; and among exported features, you only need *execute*, *set_target* (under its external name *set.target*) and *initialize* to be externally available for calls.

.IA

External software may never use a feature for creating objects unless the class text declares it as a creation procedure, and may never use a feature for calls unless the class text declares it as exported. The **Export_restriction** subclause (beginning with **export**) and the **Creation_restriction** subclause (beginning with **creation**) are not permitted to extend external availability beyond what is implied by the Eiffel class text. (For one thing, a secret feature is not required to preserve the invariant, so calling it from external software elements could put an object into an inconsistent state, which is the first step towards Armageddon.)



An “exported” feature is one that is generally available to all clients (exported without restriction). A “secret” feature, which is a special case of non-exported feature, is one which is available to no client except *NONE*.

← “*EXPORT CONTROLS AND INFORMATION HIDING*”, 7.8, page 200.

.IC

It is not incorrect for an implementation to make all exported features of all classes externally available. With such an implementation, you will usually not need any Visible paragraph. You may still, however, use an Ace (perhaps written for another implementation) that has a Visible paragraph: the semantics of such a paragraph is to specify that certain features should be externally visible; it does not preclude an implementation from providing more externally visible features — the implementation just does more than it has to.

Even an implementation which by default makes all compiled features externally visible may in fact need to support the Visible paragraph. The reason is that a compiler may include a global system optimizer, which will detect routines that are not reachable from the creation procedure of the system’s root class, and eliminate such routines from the generated code. The optimizer might also decide to inline all calls to certain routines, and then remove the object code for these routines. In such cases you will need to use a Visible paragraph to guarantee that the routines remain available for use by external software.

The syntax of the optional **Visible** paragraph is the following:

Visible \triangleq **visible** {Class_visibility ";" ... }

B.14 COMPLETE LACE GRAMMAR

For ease of reference, you will find below the complete grammar of Lace, repeating the individual descriptions given earlier in this appendix.

B.15 LACE VALIDITY RULES

The following is a list of Lace validity constraints, presented as a single rule with multiple clauses. As you know, Eiffel validity constraints are presented as “if and only if” rules, letting you know not only what you *must* do to strive for validity, but also how much is enough that you do to be *assured* of validity. The Lace constraints do not follow this style because some conditions depend on the underlying operating system and its handling of files, folders and other non-language elements that condition the workings of Eiffel tool. So the rules simply state what you must do; I did try to make the rules as complete as possible by including all known platform-independent conditions, but some conditions may have been missed.

Ace validity

An Ace must satisfy the following conditions.

- 1 • All files listed exist.
- 2 • All files listed are accessible to Eiffel tools for reading.
- 3 • All directories listed

PART V: COMPLEMENTS

This fifth part of the book contains complementary material:

- Style rules (appendix [34](#)).
- A reflection on language evolution, and its application to Eiffel (appendix [C](#)).
- An attempt to credit Eiffel properties to their inventors (appendix [D](#)).
- A summary of Eiffel's background and history (appendix [E](#)).
- The list of changes from Eiffel 3 to Eiffel 5 (appendix [F](#)).
- The list of changes from earlier versions (appendix [G](#)).
- A detailed Eiffel tutorial (appendix [H](#)).
- An Eiffel bibliography (appendix [I](#)).

Although this not reference material, it provides important complements to the detailed description of the preceding parts and the formal reference of the following one.

On language design and evolution

After an evening at the theater, we may have enjoyed the show or hated it, but meeting the playwright for a few more explanations at no extra charge — our last chance of understanding what he *really* meant — is not most people’s idea of how to finish off the evening nicely.

Undaunted by the dangers, however, I have included in this appendix a few comments on the process of language design, which will perhaps help put the rest of this book in a broader perspective. The only other aims of this informal and unpretentious discussion are to encourage further thinking, and to direct the reader’s attention to the seldom discussed topic of language evolution — what happens after the initial design.

The classic article on language design is C.A.R. Hoare’s “Hints on Programming Language Design”, reprinted in [Hoare’s] “Essays in Computing Science”, ed. C.B Jones, Prentice-Hall International, 1989, pp. 193-214.

C.1 SIMPLICITY, COMPLEXITY

One view of design holds that good languages should be small. For many years the best way to discredit any proposed design was to hint at similarity with PL/I. Just uttering that name from the back of the room was guaranteed to bring laughter to the audience and ridicule to the presenter. But many successful languages are large and complex; C++ is the most obvious example, but Java is just as typical; a look at the description of Java initialization semantics at <http://www.javaworld.com/javaworld/jw-03-1998/jw-03-initialization.html> should be enough to dispel any suspicion of simplicity.

Oversize has many damaging consequences: making it harder to learn the language; causing surprises even to experienced users, since they often will master only a subset, and may involuntarily use properties they don’t know; increasing the likelihood that compilers will be buggy, bloated, and late.

But languages should not be too simple, and the language designer should not resist useful additions on principle. One can conjecture that Pascal could have had a much more significant industrial role if a few extensions (such as variable-length array access and an elementary module facility) had been included in the standard in the late nineteen-seventies or early eighties. They were not, and Pascal was largely displaced by C, certainly a regrettable development for software engineering.

So the truth has to be somewhere between the monsters of complexity and the zen-like masterpieces of ascetism — between the bonzai and the baobab.

To complicate the discussion, there is no single definition of size. This book occupies 800 pages, which would seem to suggest that Eiffel is complex. But then most of these pages are devoted to comments and explanations, and it is possible to talk just about pure Lisp (or for that matter just about love, another seemingly simple concept) over many more pages. Then if you consider that the syntax diagrams occupy only four pages, Eiffel is very simple. From yet another viewpoint, the language properties which enable a beginner to start writing useful software, may be defined in the 20 pages of chapter 1; that is pretty short too. A “reference only” extract of the book, retaining only the formal rules (syntax, validity, semantics) interspersed throughout the text, takes up about 40 pages.

We could paraphrase a famous quote and state that a language should be as small as possible but no smaller. That doesn't help much. More interesting is the answer Jean Ichbiah gave to the journalist (for the bulletin of INRIA) who, at the time of Ada's original publication, asked him what he had to say to those who criticized the language as too big and complex: “Small languages”, he retorted, “solve small problems”.

This comment is relevant because Ada, although undoubtedly a “big language”, differs from others in that category by clearly showing (even to its critics) that it was *designed* and has little gratuitous featurism. As with other serious languages, the whole design is driven by a few powerful ideas, and every feature has a rational justification. You may disagree with some of these ideas, contest some of the justifications, and dislike some of the features, but it would be unfair to deny the consistency of the edifice. Consistency is indeed the key: size, however defined, is a measure, but consistency is the goal.

C.2 CONSISTENCY

Consistency means having a goal: never departing from a small number of powerful ideas, taking them to their full realization, and not bothering with anything that does not fit with the overall picture. Transposed to human affairs this may lead to fanaticism, but for language design no other way exists: unless you apply this principle you will never obtain an elegant, teachable and convincing result.

Note the importance for the selected ideas to possess both of the properties mentioned: each idea should be *powerful*, and there should be a *small number* of them. Eiffel may be defined by something like twenty key concepts. Here, as an illustration, are a few of them:

- Software architectures should be based on elements communicating through clearly defined *contracts*, expressed through formal preconditions, postconditions and invariants.
- *Classes* (abstract data types) should serve as both modules and types, and the modular and typing systems should entirely be based on classes. (Two immediate consequences are that no routine may exist except as part of a class defining its target type, and that Eiffel systems do not have a main program.)

- Classes should be *parameterizable* by types to support the construction of reusable software components.
- *Inheritance* is both a module extension facility and a subtyping mechanism. Attempts to restrict the mechanism to only one of these aspects, in the name of some misdirected attempt at purity, only serve to trouble the programmer with irrelevant questions. Attempt to portray *multiple* inheritance as evil only stem from clearly inadequate uses, or badly conceived language mechanisms.
- The only way to perform an actual computation is to *call* a (dynamically bound) feature on an object.
- Whenever possible, software systems should *avoid explicit discrimination* between a fixed list of cases, and instead rely on automatic selection at run time through dynamic binding.
- Client uses of classes should only rely on the official *interface*.
- A strong distinction should be maintained between *commands* (procedures) and *queries* (functions and attributes).
- A *contract violation* (exception) should lead to either organized failure or an attempt to use another strategy.
- It should be possible for a static tool to determine the type consistency of every operation by examining the software text, before execution (*static typing*).
- It should be possible to build sophisticated *run-time object structures*, modeling the often complex relations that exist in the external systems being modeled, and to let the supporting implementations take care of *garbage collection* to reclaim unused space automatically.

Eiffel is nothing else than these ideas and their companions taken to their full consequences.

Why is consistency so important? One obvious reason is that it determines your ability to teach the language: someone who understands the twenty or so basic ideas will have no trouble mastering the details.

Another justification of the consistency principle is that with more than a few basic ideas the language design becomes simply unmanageable. Language constructs have a way of interacting with each other which can drive the most careful designers crazy. This is why the idea of orthogonality, popularized by Algol 68, does not live up to its promises: apparently unrelated aspects will produce strange combinations, which the language specification must cover explicitly.

An extreme example in Eiffel is the combination of the *obsolete* and *join* mechanisms, two seemingly unrelated facilities. A class may declare a feature as obsolete to prepare for its eventual removal without destroying existing software; this is a fundamental tool for library design and evolution. In the inheritance mechanism, a class may merge (“join”) features inherited from different parents. No two mechanisms seem at first sight more “orthogonal” with each other. Yet they raise a specific question:

the Join rule must give all the properties of the feature that results from joining a few inherited features, in terms of the properties of the inherited versions; but then one of these features may be obsolete. Not the most fascinating use of language facilities; but there is no reason to disallow it. (This would require an explicit constraint anyway, and simplicity would not be the winner.) Now does this make the joined version obsolete? The language specification must give an answer. (The answer is no.)

Such cases should suffice to indicate how crucial it is to eliminate anything that is not essential. Many extensions, which might seem reasonable at first, would raise endless questions because of their possible interactions with others.

Another interesting example of interference is the absence of garbage collection in most C++ implementation. Although often justified *ex post facto* in the name of the C philosophy of putting the programmer in control of every detail, this limitation is in reality a consequence of the language's design: the presence of C-style casts makes it possible to disguise a pointer into something else, thus fooling a garbage collector and leading to serious potential errors. Many programmers do not realize how a seemingly remote property of the type system exerts such a direct influence on the very practical issue of memory management.

C.3 UNIQUENESS

Taken to its full consequences, the principle of Consistency implies the principle of Uniqueness, which states that the language design should provide one good way to express every operation of interest; it should avoid providing two.

This idea explains, for example, why Eiffel, almost alone among general-purpose languages, supports only one form of loop. Why offer five or six variants (test at the beginning, the end or the middle, direct or reverse condition, “for” loop offering automatic transition to the next element etc.) while a single, general one will be easy to learn and remember, and everything else may be programmed from it?

The loop example deserves further attention. A well-written Eiffel application will have few loops: a loop is an iteration mechanism on a data structure (such as a file or list); it should be written as a general-purpose routine in a reusable class, and then adapted to specific contexts through the techniques illustrated in the discussion of iterators. (Such pre-programmed iteration mechanisms are indeed available from libraries.) Then having to write $i := i + 1$ manually for the equivalent of a For loop is not a problem.

This observation, which would not necessarily transpose to another language, illustrates an important aspect of the Eiffel method, which makes almost all “X considered harmful” observations, for arbitrary X, obsolete.

The mechanism for marking constructs as harmful is paradoxical: as soon as you recognize some pattern X as useful, this immediately makes it harmful, by suggesting that you should not from then on reproduce X-like patterns in your software texts, but instead hide X in a reusable software component and then reuse that component directly.

Loops are harmful, then, not because they pose a danger by themselves (as may be argued of goto instructions), but because their very usefulness as a common pattern of data structure traversal suggests packaging them in reusable components describing higher-level, more abstract forms of these patterns. The only danger here would be long-term — not taking advantage of potential reuse.

The principle of Uniqueness is a particularly useful guide for language evolution, after initial design. It is natural for users of a language to request new facilities that simplify their job. Most of the time, it was possible to do this job before, which suggests that the principle requires rejecting these extensions. But that's not necessarily a correct interpretation, since the principle requires providing one *good* way of addressing each need. The question then becomes whether the previous way is good enough.

Creation expressions provide a good example. Until recently, Eiffel had a creation instruction (to create and initialize an object) but no creation expressions. The initial version of the present chapter in the first edition of this book explained the rationale in detail, stating, however, that creation expressions might have a role in the future. That future has come. Along with a creation instruction

```
create x .make (...) [A]
```

which creates an object of the appropriate type, attaches it to *x*, and initializes it with the given procedure and arguments, you may also write

```
x := {TYPE} .make (...) [B]
```

where *TYPE* is the type of *x*. (In both cases some variations and simplifications are available.) Is this a violation of the principle of Uniqueness? As presented, yes. But in practice no good programmer will ever use form [B] in the case given, because there is a better way: form [A], which avoids the need to specify the type. Why specify *TYPE* since (the language being strongly typed) it follows from the declaration of *x*? There is no good reason. Creation expressions, however, are useful in another case: creating an object whose only use is to be passed as an argument to a routine. Then you can write

```
some_routine (... , {TYPE} .make (...), ...)
```

where the restriction to creation instructions would make things far more cumbersome:

```
new_object: TYPE      -- Declare local Variable just for this purpose
...
create new_object .make (...)
some_routine (... , new_object, ...)
```

Experienced users found that such schemes occurred frequently and caused useless effort and distraction. It's not a matter of keystrokes, as a longer form is preferable when it adds relevant information; it's a matter of not wasting one's time in repetitive schemes that bring nothing new and obscure the truly relevant parts of the software.

So the two mechanisms, creation instructions and creation expressions, are both useful because they cover complementary needs.

A similar example is and “Inspect” instructions. Because of Eiffel’s emphasis on avoiding explicit discrimination and relying on dynamic binding instead, all in the name of modular, extensible, reusable architectures, the language did not initially (until 1989) include multi-branch mechanisms. As experience grew, it became clear that such mechanisms were still needed in some cases, where they did not conflict with object-oriented principles. Hence the introduction of Inspect instruction (a kind of **case ... of** discriminating on integers or characters). It is significant that the original solution erred on the side of caution: only when extensive experience clarified the conditions under which explicit discrimination was still legitimate did we go for the corresponding extensions. Better be restrictive at first, and loosen the strings later when you fully understand what’s truly needed and what would be mere featurism.

C.4 TOLERANCE AND DISCIPLINE

Using the word “restrictive” reminds us of the somewhat disciplinarian attitude that is not infrequent in the software community. One commonly hears such phrases as “preventing the programmers from doing their dirty tricks”. It is as if language designers were invested with a moral mission, and languages were a rampart against the threat of the developers’ natural uncleanness.

I disagree with this view. (This will seem surprising to those who have heard Eiffel being categorized, I believe quite wrongly, as a language of the restrictive school.) Programming language designers are not in the chastity belt business. Their role, to repeat a comment which I first heard many years ago from C.H.A. Koster, is not to prevent developers from writing bad software (a hopeless endeavor anyway), but to enable them to write good software; and perhaps to make the task pleasurable as well.

This must be applied together with the principle of Uniqueness. If you exclude a certain facility, be it the goto or function pointers, it is not to save humanity from some abomination (although you may also be doing that) but because you are providing elsewhere a better way to achieve the goals which the excluded constructs purported to address. Loops and conditionals are better than gotos, and dynamic binding under the control of static typing is better than function pointers or explicit discrimination.

In other words, if a design is defined as much by what it leaves out as by what it includes, one cannot justify the exclusions without knowing the inclusions.

These ideas pervade Eiffel. The language’s ambition is to support an elegant and powerful method for analysis, design, implementation and reuse, and to help competent developers produce high-quality software. The method is precisely defined, and the language does not attempt to promote any other way of developing software; but it also does not attempt to prevent its users from applying their creativity.

The details of the inheritance mechanism provide a clear example of these principles. The relation between inheritance and information hiding is a controversial topic; Eiffel takes the view that descendants should be entirely free to define the export status of inherited features, without being constrained by their ancestors’ choice. Nothing really forces

everyone to agree: a project leader may take a more restrictive approach and, for example, prohibit the hiding of a feature exported by a parent. It is not difficult to write a tool that will check adherence to this rule. Had the language specification taken the restrictive stand, it would have been impossible for a project leader to enforce the inverse policy.

In summary: language designers should not exclude “bad” constructs out of a desire to punish or restrict the users of the language; that is not their job. The exclusions are justified only by the inclusions: the designer should focus on the constructs that he deems essential, and his responsibility is then to remove everything else, lest he produce a monster of complexity.

C.5 METHODOLOGY

In a bad language design, the programmer is presented with a wealth of facilities, and left to figure out when to use each, when not, and which to choose when more than one appears applicable.

In a good design, each language facility goes with a precise theory — presumably explained in the accompanying book or books — of the purpose it serves: when it is desirable, when it is not.

C.6 MEA CULPA, MEA MAXIMA CULPA

The surest sign of a problematic design is the presence, in a language manual, of comments stating that some constructs should never be used. A typical example in the C++ and Java literature is the (justified) advice to avoid direct assignments to fields of objects, as in $x.a := b$, which indeed violate all the principles of information hiding and object technology.

The natural question — especially for such a recent design as Java, which does not have the excuse of being constrained by the requirement of full compatibility with C — is how one can justify producing a programming language and immediately starting to warn users against certain facilities. If the designer truly thinks (asks the naïve observer) that a certain construct is harmful, could he perhaps not have refrained from including it in the first place? Is the designer not the one who decides what goes in and what stays out?

Loving your language means never having to say you’re sorry.

C.7 THE LANGUAGE AND THE LIBRARIES

In a method supporting reusability, it is often possible and desirable to provide a new feature through a library facility rather than through a language change.

Like some other languages, Eiffel uses libraries for mechanisms such as input and output, rather than defining language constructs. The inheritance mechanism also provides a class *ANY*, inherited by all classes and offering them a number of crucial general-purpose features: *copy*, *clone*, *deep_clone* (producing recursive copies of arbitrarily large and complex object structures), equality, *out* (which produces a printable image of any value or object).

Other powerful library mechanisms include the *STORABLE* class, providing a straightforward way to store an object structure — again, arbitrarily large and complex — into a file, or to transmit it across a network, in a machine-independent format if desired.

A cynic might question the benefit of extending the libraries to keep the language simple. Indeed, tough problems of consistency and simplicity do arise for libraries. There is an important difference, however: one of level. The library as well as any user application are defined with respect to the basis provided by the language. Because everything else relies on it, this basis must be kept simple at all costs. Complexity should be avoided in libraries too, of course, but the consequences are less grave.

Mathematical theories provide the appropriate comparison. Adding a language construct is like adding an axiom, certainly not a decision to be taken lightly. Adding a library class or routine is simply like adding another theorem, inferred from the current axioms.

The interaction of libraries and language in Eiffel is sometimes intricate. The basic exception mechanism is very simple; class *EXCEPTIONS* provides further tuning, for example to handle various kinds of exception differently, or to ignore certain signals. Similarly, *MEMORY* provides finer control over the garbage collector. *INTERNAL* gives access to the internal structure of objects, useful to write system-level tools or interfaces to databases. Arrays are not a language construct but come from a library class *ARRAY*, since an array can be described as an abstractly specified object, in the same way as a list or a stack; this greatly simplifies the language and makes programs more consistent and readable. The notion of *TUPLE* is handled in a similar way. In both cases, there is a language connection through special syntax for manifest arrays or tuples.

Similarly, all basic types, from *INTEGER* to *BOOLEAN* and *STRING* are formally treated as classes (unlike the solution of C++ and Java, which separates the basic types from the rest of the type system). To the programmer, these are normal classes, which can be browsed through the normal tools. The compiler, however, cheats since it knows about these classes and can generate better code for them. This is an attempt to combine the best of both worlds: the consistency, simplicity and elegance resulting from a uniform type system; and the efficiency resulting from special knowledge.

C.8 ON SYNTAX

One of the most amusing characteristics of the software development community, from a language designer's viewpoint, is the discrepancy between professed beliefs and real opinions on the subject of programming language syntax. The official consensus is that syntax, especially "concrete" syntax (governing the textual appearance of software texts) does not matter. All that counts is structure and semantics.

Believe this and be prepared for a few surprises. You replace a parenthesis by a square bracket in the syntax of some construct, and the next day a million people march on Parliament to demand hanging of the traitors.

Of the pretense (syntax is irrelevant) and the actual reaction (syntax matters), the one to be believed is the latter. Not that haggling over parentheses is very productive, of

course, but unsatisfactory syntax usually reflects deeper problems, often semantic ones: form betrays contents.

Once a certain notation makes its way into the language, it will be used thousands of times by thousands of people: by readers to discover and understand software texts; by writers to express their ideas. If its esthetically wrong, it cannot be successful.

There is no recipe for esthetic success, but here again consistency is key. To take just one example, Eiffel follows Ada in making sure that any construct that requires an instruction (such as the body of a Loop, the body of a Routine or a branch of a Conditional) actually takes a sequence of instructions, or Compound. This is one of the simple and universal conventions which make the language easy to remember.

For syntax, some pragmatism does not hurt. A modern version of the struggle between big-endians and little-endians provides a good example. The programming language world is unevenly divided between partisans of the semicolon (or equivalent) as terminator and the Algol camp of semicolon-as-delimiter. Although the accepted wisdom nowadays is heavily in favor of the first approach, I belong to the second school. But in practice what matters is not anyone's taste but convenience for software developers: adding or forgetting a semicolon should not result in any unpleasant consequences.

In the syntax of Eiffel, the semicolon is theoretically a delimiter (between instructions, declarations, `Note_values` clauses, `Parent` parts); but the syntax was so designed as to make the semicolon syntactically redundant, useful only to improve readability; so in most contexts it is optional.

This tolerance is made possible by two syntactical properties: an empty construct is always legal; and the use of proper construct terminators (often **end**) ensures that no new component of a text may be mistaken for the continuation of the previous construct. For example in

```
.x := y  
a := b
```

there is no syntactic ambiguity, even without a semicolon, since no construct may involve two adjacent identifiers.

It is interesting to note here that the study often invoked to justify the C-Java-Ada style of semicolon as terminator (Gannon and Horning, *IEEE TSE*, June 1975) actually used subjects that were trained in PL/I and a test “separator” language that apparently treated successive semicolons as an error, a completely unrealistic assumption. This seems to invalidate the piece of conventional wisdom that asserts separators are better than terminators. The experience of Eiffel since semicolons were made optional massively suggests that semicolons are in most cases a mere nuisance.

Another example of the importance of syntax is the dominant practice, in the C-C++-Java-Perl etc. world, of the equality symbol = as assignment operator, going against centuries of mathematical tradition. Experienced programmers, so the argument goes, will never make the error. In fact they make it often. A recent review of the BSD operating system source, performed over one week-end, identified three cases of `if(x = y)` — a typo for `if(x == y)` which, unfortunately, is legal in C and C++ although it leads to

unexpected results. (In Java, at least, the first form is invalid so the error will have no catastrophic consequence.) Syntax matters.

C.9 THE INVENTOR AND THE ASSEMBLER

One of the most original comments in Hoare's *Hints* is the suggestion that the two main tasks of language design are best handled by different people: one proposes constructs, the other refrains from invention but assembles other people's suggestions into a coherent engineering construction.

The design of Eiffel has tried to disprove this rule. Eiffel embodies a significant number of inventions. Although many have been contributed by other people, a number of the concepts were devised and integrated in a single process. They include such ideas as once routines for shared objects and decentralized initialization, the multiple inheritance mechanism, object-oriented contracts and their relation with inheritance, renaming, and many others. I hope the result shows that the roles of construct inventor and system assembler are in fact compatible.

C.10 FROM THE INITIAL DESIGN TO THE ASYMPTOTE

Although the programming literature contains a few references on language design, less attention has been devoted to the subject of evolution after initial design. Yet successful languages live and change; none of the major languages in use today still adheres to the letter of its original definition. How do the design principles governing the childhood of a language carry over to adolescence and adulthood?

Software developers are inordinately opinionated people, especially on the subject of languages. Inevitably, they will come up with requests for change and extensions. Add to this tremendous and constant source of ideas the contribution of co-workers, users, course participants, colleagues in panels at conferences, and you get a constant influx of new ideas.

In the current state of technology a new element, exciting and sometimes frightening, complements these traditional sources of input: the net. Electronic mail and Usenet forums mean that thousands of people can learn in a few hours about the latest announcements, ideas, proposals, opinions and suggestions — and react to them. For Eiffel this has been a tremendous benefit. The number of people who have sent public or private comments is incomparably greater than what it would have been just a few years earlier. Even Ada, probably the language most widely and thoroughly debated before its final design, was born before network access became available on a grand scale, and did not benefit from the unique combination of breadth, depth and timeliness made possible by today's technology.

It is striking to see how many of these ideas are in fact excellent; but this does not mean that they should all be included!

First they may raise subtle or major incompatibilities with other language features; but even if this is not the case they will make the language more complex. The designers must weigh the evidence: is the purported benefit really worth the increase in

complexity? In nine out of ten cases the answer is no. Again this usually is no reflection on the quality of the idea. But the designers' primary responsibility is to keep in mind the elegance of the overall picture.

What can one do in such a context? The best tactics is to say “no”, explain that you are on your way to Vladivostok, and emerge some time later to see if there is still anyone around. This is the basic policy: do not change anything unless you cannot find any more arguments for the status quo.

But saying “no” most of the time is not an excuse for not listening. Almost any single criticism or suggestion contains something useful for the language designers. This includes comments by novices as well as expert users. Most of the time, however, you must go beyond what the comment says. Usually, what you get is presented as a solution; you must see through it and discover the *problem* that it obscures. The users and critics understand many things that the designers do not; the users, in particular, are the ones who have to live with the language day in and day out. But design is the job of the designers; you cannot expect users to do it for you. (Sometimes, of course, they will: someone comes up with just the right suggestion. This happened several times in the history of Eiffel. Then you can be really grateful.)

So there are deep and shallow comments but almost no useless ones. Sometimes the solution simply resides in better documentation. Often it lies in a tool, not in any language change. Even more often, as discussed above, the problem may be handled by library facilities: after all, this is the aim of an object-oriented language — not to solve all problems, but to provide the basic mechanisms for solving highly diverse problems.

Once in a while, however, none of this will work. You realize that some facility is missing, or inadequately addressed. When this happens — and only as a last resort — the tough conservative temporarily softens his stance. There are two cases, truly different: an extension, or a change.

C.11 EXTENSIONS

Extensions are the language designer's secret vice — the dieter's chocolate mousse on his birthday. After much remonstrance and lobbying you finally realize what many users of the language had known for a long time: that some useful type of computation is harder to express than it should be. You know it is extension season.

There is one and only one kind of acceptable language extension: the one that dawns on you with the sudden self-evidence of morning mist. It must provide a complete solution to a real problem, but usually that is not enough: almost all good extensions solve *several* potential problems at once, through a simple addition. It must be straightforward, elegant, explainable to any competent user of the language in a minute or two. (If it takes three, forget it.) It must fit perfectly within the spirit and letter of the rest of the language. It must not have any dark sides or raise any unanswerable questions. And because software engineering is engineering, and unimplemented ideas are worth little more than the whiteboard marker which serves to sketch them, you must see the implementation technique. The implementors' group in the corner of the room is

grumbling, of course — what good would a nongrumbling implementor be? — but you and they see that they can do it.

When this happens, then there is only one thing to do: go home and forget about it all until the next morning. For in most cases it will be a false alarm. If it still looks good after a whole night, then the current month may not altogether have been lost.

C.12 CHANGES

What happens if you realize that some existing language feature, which may be used by thousands of applications out in the field, could have been designed better?

The most common answer is that one should forget about it. This is also the path of least resistance: listening to the Devil of Eternal Compatibility with the Horrors of the Past, whose constant advice is to preserve at all costs the tranquillity of current users. The long-term price, however, is languages that forever keep remnants from another age. For a glimpse of the consequences, it suffices to look at recent versions of Fortran, still retaining (although they are meant for the most powerful parallel computers of tomorrow) some constructs reflecting the idiosyncrasies of the IBM 701's 1951 architecture, or at more recent "object-oriented" extensions of C, faithfully reproducing all the flaws of their parent, compounded by extra levels of complexity.

The other policy is harder to sustain, but it is also safer for the long term: if something can indeed be done better, and the difference matters, then change the construct. Such cases should of course be rare and far between — otherwise one can doubt the very soundness of the original design. They should meet two conditions:

- 1 • There must be wide agreement that the new solution is significantly better than the original one. It must not entail any negative consequence other than its incompatibility.
- 2 • The implementors must provide a conversion mechanism for existing software.

If these conditions are met, then I believe one should cut one's losses and go ahead with the change. To act otherwise is to act arrogantly (pretending that something is perfect when it is not), or to sacrifice long-term quality for short-term tranquillity.

All the issues discussed above arose in the transition between successive versions of Eiffel. It is only for the language users to judge whether the changes and extensions were justified, and whether they followed the principles discussed here. More striking than the changes has been the stability of Eiffel: the language's key properties, especially its semantics, are essentially identical to what was described in the very first publication. But the maintainers of Eiffel have not refrained from making changes, including incompatible ones. It is surprising to see both the intellectual cowardice of many people in language committees, and the positive reaction of actual users. If a change is beneficial, clearly explained, carefully prepared, and well organized (avoiding pointing a gun to their head: change *now* or die!), they will go for it.

C.13 THE POLITICS OF LANGUAGE EVOLUTION

The mention of committees brings in the final observation of this overview, addressing not the technology of language evolution but its politics. A number of models are possible:

- The Town Hall model.
Everyone votes, and the majority wins.
- The Venetian model.
The Doges haggle it out between themselves.
- The Tammany Hall model.
Everyone votes, and the bosses haggle it out between themselves.
- The dog pack model.
He who shouts the loudest wins.
- The Usenet model.
He who shouts the longest wins.
- The dictatorship model.
The dictator wins (until toppled).
- The engineering project model.
The chief engineer wins, but only if he can convince the other engineers most of the time.
- The CEO model
Like the engineering project model, but the board must approve major decisions.

Without reference to the management of society, where different criteria apply, I have through my experience come to the conclusion that the appropriate model for language evolution is one of the last two. Democracy is admirable for the government of humans, but a language is before all an engineering project, and someone should be in charge. As in a company, many checks and balances should be provided, and the chief engineer should very seldom be permitted to pass his views just because he is the chief engineer. A technical leader who has to govern by fiat — as opposed to convincing the troops on the sheer strength of technical arguments — will not remain a leader for very long.

Once in a while, he *does* get his druthers on the grounds of authority, simply because several good choices are available and someone needs to decide; this is usually for concrete syntax details. Such cases should remain rare. After all, if the chief engineer deserves the position at all, his ideas, or more commonly his ability to sort out the good ideas from the bad, regardless of who originated them, should be better than everyone else's, so he should expect to win on the merits.

E

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of “Eiffel: The Language”. Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

A brief history of Eiffel

Eiffel was designed on 23 September 1985. It was initially intended as an internal tool for the newly created ISE (Interactive Software Engineering, based in Santa Barbara, California). The first internal implementation was ready in mid-1986.

The main influences on the design of Eiffel have been:

- The object-oriented concepts introduced by Simula 67, which I had been able to practice starting at the end of 1973.
- Work on formal specification, in particular Abrial’s original version of the Z specification language (which I described in a 1978 book).
- Work on abstract data types by Liskov, Zilles, Guttag and myself.
- The Algol 60-Algol W-Pascal-Ada line of programming languages.
- Work on program proving and axiomatic semantics (Floyd, Hoare, Dijkstra).
- Modern concepts of software engineering, in particular the work on software quality.

A presentation at the first OOPSLA conference (Object-Oriented Programming, Systems, Languages and Applications, Portland, September 1986) revealed that many of the concepts and their implementation were ahead of the rest of the industry as well as of academic research, and led to the transformation of the compiler into a commercial product, which started to be sold to companies and universities worldwide in December of 1986. Version 2 was introduced in 1988. ISE’s original technology reached its peak with version 2.3, released in 1990.

The book *Object-Oriented Software Construction*, whose first edition was published in 1988, enjoyed a large success and introduced Eiffel to a broader community. (The second, greatly expanded edition appeared in 1997.)

In 1990, ISE released the language definition to the public domain, spawning several compiler and library projects. At that time a general cleanup of the language was undertaken, leading to a number of simplifications and a few extensions (recalled in appendix [G](#)). These changes did not, however, affect the essential concepts and techniques of the language and method; Eiffel has been remarkably stable, and remains close today to the original 1985 design. The language reference, *Eiffel: The Language*, the first edition of the present book,

was published in 1991 (although earlier versions had been available since 1988 as ISE manuals) and revised the following year.

The first versions of ISE Eiffel, for obvious reasons of necessity, had been written in C. From 1990 to 1993 the technology was reengineered in Eiffel, using version 2.3 for the initial bootstrap. This led to ISE Eiffel 3, a complete graphical development environment first released in 1993, and to its successors ISE Eiffel 4 (1997) and Eiffel 5 (2001). Language extensions contributed during that period include agents (providing Eiffel with much of the power of functional languages, and introspection), the **Precursor** mechanism, a simplification of some aspects of repeated inheritance, more elaborate facilities for object creation, the conversion constructs, and other techniques described in detail in this book.

Another notable event was the publication in 1995 of Waldén's and Nerson's *Seamless Object-Oriented Software Construction* which introduced the Business Object Notation, prolonging Eiffel on the analysis and design side in a form that is attractive to managers, analysts and system architects.

Besides ISE, other suppliers of Eiffel tools include Object Tools (Germany) with the Visual Eiffel environment, a commercial compiler, the successor to Eiffel/S; Small Eiffel from the University of Nancy (France), a free software implementation; and another commercial offering from Halstenbach GmbH (Germany). Now defunct implementations have included Eon Eiffel, Tower Eiffel. Libraries are available from numerous sources, covering areas such as 3D graphics, lexical analysis and parsing (Gobo), DirectX graphics and several other graphical libraries, variable-precision arithmetic and many others.

Today Eiffel is used to develop some of the largest, most ambitious successful software projects in the world. Areas of application include banking, financial systems, accounting, telecommunications, health care, CAD-CAM, simulation, real time, scientific computing, scientific visualization. Some of the most visible projects, such as CALFP Bank's Rainbow system, initially a derivative trading system but having grown to oversee most of the bank's operation, have been extensively documented in the press and are also featured at www.eiffel.com.

Eiffel is also popular as a teaching tool in universities and even high schools. A large number of universities are in fact using Eiffel as the first programming language taught to students. Others use it at various levels in the curriculum, aided by attractive packages from the Eiffel product providers.

The name *Eiffel* is an homage to Gustave Eiffel, the man who built the eponymous Tower in Paris as well as many other durable constructions such as the metallic armature of the Statue of Liberty in New York and a Budapest railway station. The Eiffel Tower, started in 1887 for the 1889 World Fair, was completed on time and within budget; it has survived political hostility and attempts at destruction; found many new uses (such as radio and television); proved to be robust and efficient. Built out of a small number of robust, elegant design patterns, combined and varied repeatedly to yield a powerful result, it is the best symbol of what Eiffel can achieve for the software world.

F

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

Language changes from the previous edition

F.1 OVERVIEW

Stability has been the principal characteristic of Eiffel's history since the language was designed on 27 September 1985. The concepts behind the language, the structure of software texts, and the principal constructs have remained the same. There have of course been significant changes:

- ISE Eiffel 2.1 (1988) introduced constrained genericity and the Assignment Attempt mechanism.
- Versions 2.1 to 2.3 introduced expanded types, double-precision reals, expanded classes and types, the join mechanism for deferred features, assignment attempt, the **Note** clause (then **Indexing**), infix and prefix operators (now treated through **alias** clauses), the **Obsolete** clause, **Unique** values (removed in the present iteration), the **Multi_branch** instruction.
- The transition from Eiffel 2 to Eiffel 3 (1990-1993) was the opportunity for a general cleanup of the language, unification and simplification of the concepts; in particular it made basic types full-fledged classes, to yield a completely consistent type system, and got rid of special features such as *Forget*, so that feature call always applies to objects rather than references. The first edition of this book officially introduced Eiffel 3; by providing the complete reference for a full-function language, it permitted the growth of the Eiffel industry and served as the basis for all current commercial and non-commercial compilers.
- Eiffel 4 (in particular ISE's Eiffel 4.2 in 1998 and 4.3 to 4.5 in 1999) introduced the **Precursor** construct, recursive generic constraints, tuples, agents, creation expressions and a new creation syntax.

- The present edition describes Eiffel 5, which brings a few significant improvements, although it remains close to previous versions. In the Eiffel tradition, the changes are not so much extensions (we are constantly wary of the danger of “creeping featurism”) as efforts to make the language cleaner, simpler, more consistent, easier to learn, easier to use. This revision also *removes* a number of mechanisms (*BIT* types, *Strip* expressions), for which we identified better alternatives.

This appendix describes the language changes from the preceding edition to the present one, which are also the changes from Eiffel 3 to Eiffel 5.

Since the majority of Eiffel 5 users with pre-Eiffel-5 experience started with Eiffel 3, the pre-Eiffel-3 changes are of mostly historical interest. For that reason they appear in a separate appendix.

The presentation of Eiffel 5 changes will successively consider: removed mechanisms; compatibility issues; new constructs; semantic changes to existing constructs; lexical and syntactic changes; changes to validity constraints and conformance rules.

F.2 REMOVED MECHANISMS

It has been a general principle of Eiffel evolution that in spite of its high expressive power the language should remain of manageable size, allowing Eiffel programmers to master *all* of Eiffel: there must be no dark holes in the language. In particular, if we find a better way of doing something, there is no reason to retain the previous constructs, as long as we make the transition easy for existing programs (see the compatibility notes in the next section). Along with its introduction of powerful new mechanisms, Eiffel 5 removes a few that are no longer needed.

The notion of infix and prefix features are now handled by a simpler and more general mechanism, using the existing keyword *alias*. The keywords *infix* and *prefix* are, as a consequence, no longer necessary. There is no loss of functionality — rather, a more general mechanism.

The notion of *BIT* type has been removed. It enabled manipulation of bit sequences. The richer set of features in class *INTEGER* — *bit_and*, *bit_not* and so on, as well as the creation procedure *make* that sets the bit size to an arbitrary value — provides a more versatile replacement.

The notion of *Strip* expression has been removed. It was mainly useful in assertions and is advantageously replaced by a combination of tuple and agent mechanisms.

Type *DOUBLE*, for “double-precision” reals, has been removed. The evolution of computer hardware and the needs of numerical computation lead to making every *REAL* 64-bit long. The new sized type *REAL_32* is available to declare shorter floating-point numbers.

The *global inheritance structure* has been simplified: *ANY* no longer has ancestors *GENERAL* and *PLATFORM*. *GENERAL* is gone, so *ANY*'s features are declared in *ANY* itself. *PLATFORM* is still there, but as a supplier rather than ancestor of *ANY*, through a new query *platform* of type *PLATFORM* in *ANY*, providing access to platform-specific properties. Appendix A.

F.3 BACKWARD COMPATIBILITY

The transition from Eiffel 2 to Eiffel 3 required changing some ways of expressing fundamental operations, such as comparison to *Void*. Accordingly, a translator was made available by ISE at the time.

The changes from Eiffel 3 to Eiffel 5 may only cause minor incompatibilities for existing Eiffel 3 software:

- The following new reserved words may not be used as identifiers: **assign**, **attached**, **attribute**, **create**, *Precursor*, **only**, **note**, *TUPLE*.

The keywords **creation**, **indexing**, **infix**, **prefix** and **select** have been removed but compilers may continue to support them for a while, so you should refrain from using them as identifiers.

- If you had a feature called *default_create*, you should find another name, unless you wish to use it as a redefinition of the corresponding feature from *ANY*.
- If you had classes called *FUNCTION*, *PROCEDURE*, *ROUTINE* or *TYPE*, they will conflict with the corresponding new classes from the Kernel Library, so you should use a different name.
- In a **Note** clause (previously **Indexing**) the initial colon-terminated **Note_name** term, previously optional, is now required; you will have to add it if missing.
- Creation is now written **create** *x* rather than **!!** *x* and **create** {*TYPE*} *x* rather than **! TYPE !** *x*. This is the most visible syntax change, but does not raise any immediate concern since compilers should continue to support the previous syntax for several years. (This is the case with ISE Eiffel.) A translator does not appear necessary, although some scripts may be made available to update creation instructions to the new form.

Incompatibilities may also result from the removal of *BIT* types and **Strip** expressions. The new bit manipulation features of class *INTEGER* provide a superior replacement for *BIT* types; **Strip** expressions were rarely used and their effect can be obtained in a simpler way through the agent mechanism. Here too compilers such as ISE Eiffel will continue to support the older mechanisms for several years.

Any compatibility problem resulting from the removal of *GENERAL* and *PLATFORM* should be easy to correct.

F.4 NEW CONSTRUCTS

The *agent* mechanism (using tuples) is a major addition.

Chapter 27.

Tuples (anonymous classes) are new.

Chapter 13.

The *generic creation* mechanism, making it possible to create objects of a `Formal_generic_name` type, is new.

Chapters 12 and 20.

Creation expressions are new. (Pre-Eiffel-5, only creation instructions were available.)

20.14, page 558.

Assigner procedures, allowing a procedure call `x.put(v, i)` to be written in assignment-like syntax as `x.item(i) := v` if `put` has been declared as an associated procedure for a query `item`, is new.

====

A related mechanism, *bracket syntax* for queries and commands, allowing the previous instruction also to be written `x[i] := v`, is new.

====

A new *conversion mechanism* generalizes the ad hoc conformance rules that allowed conformance of `INTEGER` to `REAL` and of `INTEGER` and `REAL` to `DOUBLE`, as well as the “balancing rule” which permitted mixed-mode arithmetic, as in `your_integer + your_real`. Instead, there is now a general-purpose conversion and expression balancing mechanism, used by the basic types in the Kernel Library but applicable to any other classes. The notion of “*compatibility*”, covering both conformance and convertibility, is a result of this addition; for assignment and argument passing, the rule is that the source type must be compatible with the target type, not just conforming as before.

Chapter 15.

The `Precursor` construct is new, replacing techniques (still applicable in complex cases) relying on repeated inheritance.

10.24, page 299.

The `only` postcondition clause, useful to avoid unwanted side effects especially in assertions and concurrent computation, is new.

Chapters 8 and 20.

The use of a `Note` clause (previously `Indexing`) to annotate a feature, a control structure or the end of a class is new. Previously, `Indexing` clauses were applicable at the beginning of a class only.

The ability to declare an attribute explicitly, with the keyword `attribute`, is new. This allows attaching preconditions, postconditions and note clauses to attributes as well as routines. The previous syntax, just `x: A`, remains applicable as a common abbreviation.

Verbatim strings are new.

.29.8, page 794.

The sized variants of basic types, such as `INTEGER_8` and `REAL_64` are new.

====

====

The `~` operator for object equality, associated with `is_equal`, is new.

38.5 SEMANTIC EXTENSIONS AND CHANGES

The generic mechanism now explicitly supports “recursive generic constraints”, in which a constraint for a generic parameter may involve another (or the same) generic parameter, as in `class C [G, H → ARRAY [G]]`.

The semantics of *creation* has been made simpler, for creation instructions that do not explicitly list a creation procedure, by assuming that this uses the *default_create* procedure, introduced in *ANY* and redefinable in any class. *Chapter 20.*

A class may now be declared as deferred even if it has no deferred feature. This makes it non-instantiable like any other deferred class. A consequence is that it is no longer permitted to have an empty `Creation_procedure_list` in a `Creation_clause`; specifying `class A create feature ...` with nothing after `create` was a way to prohibit instantiating the class. It now suffices to make *A* deferred, even if all its features are effective. *Class Header rule, page 126; creation clause syntax, page 547.*

The anchor of an Anchored type `like anchor` may now itself be anchored, as long as there is no cycle in the anchoring structure. In addition it is now possible to use an expanded or formal generic anchor. With the exception of expanded anchors this officializes possibilities that ISE Eiffel has supported for a long time. *11.10, page 339.*

The *Feature Identifier principle* is new in its full generality. The difference between operator and identifier features was and is intended for feature calls only; what is new is that every feature now has an associated identifier, with the infix, prefix or bracket alias providing only a simplification for calls. This convention doesn't just serve consistency, but also allows, for example, to define agents on features of any kind. *Page 153.*

The *once routine* mechanism has gained new flexibility through the introduction of “once keys” allowing “once per thread”, “once per object”, and manual control through the new class `ONCE_MANAGER`. *“ONCE ROUTINES”, 23.14, page 641.*

Multi-branch instructions support two new forms, one discriminating on strings (in addition to the integers and characters previously supported), the other on the type of an object. *“MULTI-BRANCH CHOICE”, 17.4, page 482.*

The arithmetic types have been developed and made more precise; this includes new types such as `INTEGER_8` noted in the previous section, but also the specification that `INTEGER` means 32-bit integer and `REAL` means 64-bit real, and also explains the removal of `DOUBLE`.

Equality semantics now specifies that two objects cannot be equal unless their types are identical; previously, it was possible for an object to be equal to one of conforming type. The main reason for this change was to follow mathematical tradition by ensuring that equality is fully symmetric. Correspondingly, *copy semantics* requires an argument of type is identical — not just conforming — to the type of the target.

[“OBJECT EQUALITY”, 21.6, page 580,](#)
and [“COPYING AN OBJECT”, 21.2, page 565.](#)

Non-conforming inheritance was present in the case of inheritance from an expanded class, but has been generalized to permit a **Parent** clause of the form **inherit {NONE} C**, hereby providing a simpler solution to the issues of repeated inheritance and removing the need for **Select**.

[“NON-CONFORMING INHERITANCE”, 6.8, page 180;](#)
[“THE CASE OF REDECLARED FEATURES”, 16.5, page 442](#)

The possibility to declare a class — not just a routine — as *frozen* is new.

[“CLASS HEADER”, 4.9, page 124.](#)

Although *external features* have always been present, they originally supported only a **Language_name**, such as **"C"**, and an optional **alias** specification (**External_name**). The inclusion of mini-sublanguages allowing detailed C specifications comes from ISE Eiffel 3, which provided direct support for C macros, include files and DLLs. Changes from that version include: removing of 16-bit DLL support (technically obsolete); replacing the keyword **dll32** and the class name **DLL_16** by **dll** and **DLL**; accepting routine names as well as routine indexes in **dll** specifications; specifying that in the absence of an **alias** subclause the name to be passed to the external language is the lower name of the external Eiffel feature ; replacing the vertical bar |, used to introduce include files, by the keyword **include**. ISE Eiffel 4 introduced C++-specific mechanisms, allowing an Eiffel class to use the member functions, static functions, data members, constructors and destructors of a C++ class. That version also introduced the Legacy++ class wrapper and the Java interface. Eiffel 5 adds support for **inline** C functions and C **struct** specifications. The Cecil library mechanisms have also been considerably refined and extended based on extensive experience with the library.

[Chapter 31.](#)

F.5 KERNEL LIBRARY CHANGES

A number of changes have been brought to the Kernel Library (ELKS); [Appendix A](#). only the most important ones will be listed here.

The names of features for comparison, object duplication and copying have been made more consistent, as shown by the following tables. Asterisks indicate new names — for existing features or, in the case of *twin*, new ones; names in roman and in parentheses indicate previous names.

OBJECT EQUALITY	FIX FIX FIX FIX!!!! Between arguments	Between target and argument
Redefinable	<i>equal</i> alias " }={ " <—	<i>is_equal</i>
Frozen	<i>*identical</i> <— (<i>standard_equal</i>)	<i>*is_identical</i> <— (<i>standard_is_equal</i>)

OBJECT DUPLICATION	Of argument	Of target
Redefinable	<i>clone</i>	<i>twin</i>
Frozen	<i>*identical_clone</i> (<i>standard_clone</i>)	<i>*identical_twin</i>

OBJECT COPY	Of argument onto target
Redefinable	<i>copy</i>
Frozen	<i>identical_copy</i> <— (<i>standard_copy</i>)

The purpose of this change is to make the names uniform and easy to remember:

- Add *is_* for queries applying to the target: *equal* (*x*, *y*) compares its arguments, *x.is_equal* (*y*) compares the argument to the target.
- Use *identical* for frozen (non-redefinable) operations, which guarantee the original semantics of field-by-field equality or copying: *equal* and *copy* are redefinable, *identical* and *identical_copy* are not. Note that *clone* and its target-oriented variant *twin* are not directly redefinable, but they follow the redefinitions of *copy*.

*The previous conventions were not bad, but the new ones seem a little better, especially with the introduction of *twin*.*

---- **FIX FIX FIX "~"** is a new synonym of *equal*, making it a little easier to express object equality as *a }={ b*. (The symbol suggests an equal sign opening up both left and right to embrace the objects denoted by the operands.)

In addition, as noted in the previous section, copy and equality features now use type identity rather than type conformance between their arguments. This has led to a stronger precondition for *copy*, using *same_type* rather than *conforms_to*.

---- **FIX FIX FIX** Thanks to the introduction of `Class_type_reference`, it has been possible to remove classes `INTEGER_REF`, `CHARACTER_REF` and so on; the equivalent is now provided by

F.6 LEXICAL AND SYNTACTIC CHANGES

A small change to the method of language description, rather than the language itself: in the conventions for describing the syntax, a “zero or more” repetition is now marked by an asterisk, as in `{Type ";" ... }*`, for symmetry with the convention for “one or more”, which uses a plus sign. Previously, the asterisk was omitted.

[“Repetition productions”, page 90.](#)

There are eight new reserved words as already noted: **agent**, **attribute**, **create** (making a comeback from Eiffel 1 and 2), **note**, **only**, *Precursor*, **reference**, *TUPLE*. Among these, **create** is a replacement for **creation** and **note** for **indexing**.

The words **creation**, **note** and **select** are no longer keywords (hence no longer reserved), but compilers will probably treat them as reserved for a while, the first as a synonym for **create**, the second to support previous repeated inheritance rules.

The following words are no longer reserved: *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, *DOUBLE*, *POINTER*. You should still not use them as class names, since they would conflict with classes that an Eiffel compiler will expect to find in the Kernel Library, and optimize. But you may now call a feature *integer* (although that’s probably not a good idea).

A `Note_entry` is of the form

something: *a, b, c*

[“ANNOTATING A CLASS”, 4.8, page 122.](#)

where *something*: is the `Note_name` and one or more `Note_item` follow the colon. Previously the `Note_name` part (including the colon) was optional. In practice developers included it almost all of the time. It is now required. This makes the grammar more regular, and facilitates parsing, especially as the semicolon is optional between a `Note_entry` and the next.

A syntax rule required underscores, if used in manifest integer and real numbers, to separate digits by groups of three. It has been replaced by a mere style recommendation.

[“INTEGERS”, 32.16, page 899, and “REAL NUMBERS”, 32.17, page 902.](#)

The syntax for creation instructions previously used exclamation mark characters **!**. For clarity, this has now been replaced by a keyword-based notation relying on the keyword **create**, permitted for creation expressions as well (see new constructs below). For consistency and to avoid any confusion, the keyword **create** is also used to introduce a `Creators` part listing the creation procedures of a class (previously the keyword there was **creation**).

The recommended separator between successive generic parameters, *Sections 12.2 and 12.3*, either formal as in a class declaration `class C [G; H] ...` or actual as in a generic derivation `C [TYPE1; TYPE2]`, is now the semicolon. The comma (the previous choice) is still supported.

The **Precursor** construct, which may include an explicit type as in

```
Precursor {TYPE}           -- Or the version with arguments:
Precursor {TYPE} (arguments)
```

was first introduced in *Object-Oriented Software Construction, 2nd edition* (Prentice Hall, 1997), where this form of the construct is written with the type specification first: `{TYPE} Precursor (...)`. An early printing even had double ... braces, as in `{{TYPE}} Precursor (...)`, showing once again that simple solutions sometimes come last. ISE Eiffel currently supports all three variants, but with the publication of this book the discarded ones should quickly disappear from practical use.

The syntax for **New_export_item**, in the **New_exports** clause that allows *Examples in ,page 204; syntax on page 209.* a class to change the export status of some inherited features, now supports an optional **Header_comment** to indicate the status of the corresponding features, such as `-- Implementation`. This is consistent with the corresponding convention for labeling feature clauses.

E.7 CHANGES IN VALIDITY CONSTRAINTS AND CONFORMANCE RULES

Some changes, most of them simplifications, have been brought to validity constraints (including conformance rules, treated in the same style as validity constraints in chapter 14). The changes are summarized in the following table.

Some of these changes involve a constraint that has been **removed**, for one of three reasons:

- The constraint was found to be too restrictive, and its removal not to have any negative effect on software quality.
- The constraint was really a style rule, and users felt it should not be enforced by compilers.
- Other language changes made the constraint unnecessary.

A few constraints have been **added** to reflect the rules associated with the new constructs of Eiffel 5.

In addition, the table includes entries for some constraints having undergone changes affecting only their presentation:

- The order of clauses may have been changed for clearer exposition.
- Every constraint has a name; for consistency, some names have been changed (or added, in a few cases of originally nameless constraints).

- Every constraint has a **Cxyz** code (previously **Vxyz**); in a few cases this has been changed for better mnemonic value and consistency. (The table, as noted, only lists a constraint if the **xyz** part has changed.)

Page numbers in *small italics* in the second column refer to the first edition of this book and determine the order of entries in the table.

Constraint name	Old code, <i>page</i>	New code	Page	Explanation
Root Class rule	VSRC 36	VSRT	<u>112</u>	Clause 3 added to preclude root class of a system from being deferred, necessary condition omitted in first edition. Removes limitation to one creation procedure. Previous clause 2 is now clause <u>2</u> of new constraint VSRP (next entry).
Root Procedure rule (previously covered by Root Class rule, see previous entry)	VSRC 36	VS RP	<u>113</u>	New rule covering what was clause 2 of VSRC (previous entry). Previous phrasing, applying to <i>all</i> creation procedures of root class, was too restrictive. Clause <u>2</u> of new rule governs root procedure only. Clause <u>1</u> states that root procedure must be creation procedure of root class. Clause 3 is a new condition, prohibiting preconditions.
Cluster Class Name rule (previously: no name)	VSCN 51	<i>Removed</i>		---- COMPLETE ----
Class Header rule	VCCH 51	VCCH	<u>126</u>	Loosened to permit the declaration of a class as deferred even if it has no deferred feature.
(No name)	VCRN 53	<i>Removed</i>		Required ending comment of class, if present, to repeat class name. Ending comment has been removed, even as a style rule.
Feature Declaration rule	VFFD 69	VFFD	<u>162</u>	Replacement of clauses 5 and 6 by reference to Alias Validity (see next entry).
Alias validity	VFFD 69 (<i>Clauses 5 and 6</i>)	VFAV	<u>163</u>	Revision of part of VFFD accounting for new of alias clauses replacing prefix and infix and introducing bracket features.
Parent rule	VHPR 81	VHPR	<u>178</u>	The rule now refers to the <i>Unfolded Inheritance Clause</i> of a class to account for implicit inheritance from ANY . Clause <u>2</u> is new, to take into account the new notion of frozen class. Clause <u>4</u> is new, to ensure VHUC (see next entry). Clause <u>5</u> should have been there all along but is new.
Universal Conformance rule	(<i>NEW</i>) 81	VHUC	<u>173</u>	Theorem, follows from other validity rules. Was essentially satisfied before, but not stated.
Rename Clause rule	VHRC 81	VHRC	<u>185</u>	Two new clauses: <u>3</u> requires Feature Name rule (VMFN , page <u>474</u>) to apply (previously only expressed as margin comment); <u>4</u> covers renaming into feature with operator or bracket alias.

Constraint name	Old code, page	New code	Page	Explanation
Class <i>ANY</i> rule	<i>VHAY</i> 88	<i>VHCA</i>	<u>173</u>	Code change for clarity.
Expanded Client rule	<i>VLEC</i> 94	<i>Removed</i>		New semantics of expanded variables makes it possible to accommodate expanded client cycles.
(No name)	<i>VLCP</i> 101	<i>Removed</i>		Required identifiers listed in a Clients part to be names of classes in the universe. See rationale for the removal in the paragraphs starting with “ <i>There is no validity constraint on Clients parts</i> ”, page 209 .
Entity Declaration rule	<i>VREG</i> 110	<i>VRED</i>	<u>221</u>	Code change for clarity.
Local Variable rule	<i>VRLE</i> 115	<i>VRLV</i>	<u>226</u>	Code change for clarity (previous terminology was “Local Entity”).
Feature Body rule (replacing Routine rule)	<i>VRRR</i> 113	<i>VFFB</i>	<u>144</u>	New rule is generalization of old one: covers all features, not just routines. It follows from the introduction of the attribute keyword, making some clauses (in particular Precondition and Postcondition) applicable to all features.
Old Expression rule	<i>VAOL</i> 124	<i>VAOX</i>	<u>239</u>	Code change for clarity.
Old Expression rule	(<i>NEW</i>)	<i>VAON</i>	<u>243</u>	Validity rule for new only construct.
Precursor rule	(<i>NEW</i>)	<i>VDPR</i>	<u>304</u>	New rule , covering new construct.
Definition of deferred and effective class	161		<u>127</u>	(Not validity constraint, but definition used by other constraints.) Moved to earlier chapter; updated to permit class to be deferred even without deferred features. See entry on VCCH above.
Deferred class property	(161)		<u>310</u>	(Not separate constraint, but consequence of others.) Clarifies that a class can be deferred even without deferred features. See previous and next entries.
Effective class property	(161)		<u>311</u>	(Not separate constraint, but consequence of others.) Clarifies that a class can be deferred even without deferred features. See previous two entries.
Redeclaration rule	<i>VDRD</i> 163	<i>VDRD</i>	<u>313</u>	Last clause removed; prohibited redefining an external feature into an Internal one. This was an implementation constraint, no longer justified.
Join rule	<i>VDJR</i> 165	<i>VDJR</i>	<u>319</u>	Rephrased to take into account two cases missed by original: joining of one effective feature with one or more deferred ones; redefinition of all. Not language change but clarification of rule that was always there.
Join semantics rule (not validity constraint but semantic rule)	166		<u>320</u>	Beginning of rule updated to include cases mentioned in previous entry. ===== FIX ===== Clause 6 added to cover case of effecting one or more deferred features.

Constraint name	Old code, page	New code	Page	Explanation
Name Clash rule (previously: no name)	<i>VNCN</i> 189	<i>VMNC</i>	<u>475</u>	Name change for consistency. Slight rephrasing, but no change of substance. This is a redundant rule, following from <i>VMFN/VMFN</i> (Feature Name, unchanged).
Select Subclause rule	<i>VMSS</i> 192	<i>Removed</i>		Governed a clause, <i>Select</i> , that no longer exists thanks to simplification of repeated inheritance mechanism.
Unconstrained Genericity rule	<i>VTUG</i> 201	<i>Removed</i>		Now merged with <i>VTGD</i> of which it was a special case (repeated in its clause <u>1</u>).
Generic Constraint rule	(<i>NEW</i>)	<i>VTGC</i>	<u>357</u>	New rule taking into account generic creation and multiple generic constraints.
Genericity Derivation rule (previously: Constrained Genericity rule)	<i>VTGC</i> 203	<i>VTGD</i>	<u>359</u>	Clause <u>2</u> amended to permit recursive constraints, as in class C [<i>G, H</i> → <i>ARRAY</i> [<i>G</i>]].
Expanded Type rule	<i>VTEC</i> 209	<i>VCCH</i>	<u>126</u>	Rule no longer needed as type rule thanks to removal of expanded T types (all expanded types are now based on an expanded class) and removal of requirement of <i>default_create</i> for expanded types.
Anchored Type rule	<i>VTAT</i> 214	<i>VTAT</i>	<u>345</u>	Considerably loosened conditions: anchor chains now possible (<i>a</i> declared like <i>b</i> with <i>b</i> declared like <i>c</i>) if there's no cycle; anchoring now permitted on expanded and formal generic. No more anchoring on arguments. Properties of anchored type now completely defined by those of its unfolded form.
General conformance	<i>VNCC</i> 219	<i>VNCC</i>	<u>388</u>	Clause <u>3</u> integrates attached type requirements; new clause <u>6</u> handles anchored types and allows removal of <i>VNCG</i> (see below).
Direct conformance: class types	<i>VNCN</i> 221	<i>VNCN</i>	<u>390</u>	Simplified thanks to the notion of generic substitution; also subsumes <i>VNCG</i> (next entry).
Direct conformance: generic substitution	<i>VNCG</i> 222	<i>Removed</i>		Covered by new formulation of <i>VNCN</i> (see previous entry).
Direct conformance: formal generic	<i>VNCF</i> 224	<i>VNCF</i>	<u>393</u>	Simplified thanks to a more general notion of constraint. Also, addresses multiple constraints.
Direct conformance: anchored types	<i>VNCH</i> 225	<i>Removed</i>		Anchored types are now treated more simply like “macros”. See clause of
Direct conformance: expanded types	<i>VNCE</i> 229	<i>VNCE</i>	<u>396</u>	---- FIX --- Previous clauses 2 and 3 removed as they are now covered by convertibility rather than conformance (in a more general form including new explicitly sized arithmetic types such as <i>INTEGER_16</i>).
Direct conformance: Bit types	<i>VNCB</i> 229	<i>Removed</i>		No longer applicable since Bit types were removed.

Constraint name	Old code, page	New code	Page	Explanation
Direct conformance: tuple types	(NEW)	<u>VNCT</u>	<u>397</u>	New rule, covering conformance for new kind of type.
Conversion Procedure rule	(NEW)	<u>VYCP</u>	<u>411</u>	Convertibility is new.
Conversion Query rule	(NEW)	<u>VYCQ</u>	<u>413</u>	Convertibility is new.
Expression convertibility	(NEW)	<u>VYEC</u>	<u>424</u>	Convertibility is new.
Precondition-free	(NEW)	<u>VYPF</u>	<u>426</u>	New concept closely connected with convertibility.
Multi-Branch rule	<i>VOMB</i> 239	<u>VOMB</u>	<u>488</u>	Removed all constraints relating to Unique values, no longer present in the language.
Unique declaration rule	266	<i>Removed</i>		Removed all constraints relating to Unique values, no longer present in the language.
Unique Declaration rule (previously: no name)	<i>VQUI</i> 266	<i>Removed</i>		Removed all constraints relating to Unique values, no longer present in the language.
Entity rule	<i>VEEN</i> 276	<u>VEEN</u>	<u>513</u>	Clearer clause numbering; new clause <u>7</u> (imitated from clause <u>6</u>) to cover new notion of inline agent.
Variable rule	(NEW)	<u>VEVA</u>	<u>514</u>	New rule made necessary by inline agents.
Creation Precondition rule	(NEW)	<u>VGCP</u>	<u>547</u>	New rule restricting what's permissible in the precondition of a creation procedure.
Creation Clause rule	<i>VGCP</i> 285	<u>VGCC</u>	<u>548</u>	Code change for clarity. Previous clause 4 removed: made unnecessary by <i>default_create</i> convention; <u>VCCH</u> takes care of the rest. New clause <u>4</u> added to preclude using once routines. New clause <u>5</u> to rule out unsound precondition clauses. Do not confuse with new <i>VGCP</i> (previous entry) or old <i>VGCC</i> (next entry).
Creation Instruction rule	<i>VGCC</i> 286	<u>VGCI</u>	<u>553</u>	Code change for clarity. Drastic simplification. Note that some of the old clauses reappear as “corollaries” of <u>VGCI</u> in the new <i>VGCP</i> , page <u>555</u> . New clause <u>4</u> takes into account generic creation. Do not confuse with new <i>VGCC</i> (previous entry).
(No name)	<i>VGCI</i> 288	<i>Removed</i>		System validity part removed. Do not confuse with clause now called <u>VGCI</u> (previous entry).
Creation Instruction Properties	(Parts of <i>VGCC</i>) 288	<u>VGCP</u>	<u>555</u>	New rule, corollary of <u>VGCI</u> (next-to-previous entry) and hence redundant, but providing extra error messages for compilers.
Creation Expression rule	(NEW)	<u>VGCE</u>	<u>562</u>	Creation expressions are new.
Creation Expression properties	(NEW)	<u>VG CX</u>	<u>562</u>	Same relation to <u>VGCE</u> as <u>VGCP</u> to <u>VGCI</u> (see previous entries).

Constraint name	Old code, page	New code	Page	Explanation
Assigner Call rule	(NEW)	<u>VBAC</u>	<u>610</u>	Assigner calls are new.
Assignment Attempt rule	<u>VJRV</u> 332	Removed		No more assignment attempt (replaced by <u>Object_test</u>)
Non-Object Call rule	(NEW)	<u>VUNO</u>	<u>631</u>	Non-object calls are new.
Call Use rule (previously: no name)	<u>VKCN</u> 368	<u>VUCN</u>	<u>623</u>	Code change for consistency.
Export rule	<u>VUEX</u> 368	<u>VUEX</u>	<u>632</u>	Simplification (the former case 2 wasn't necessary) and addition of <u>Non_object_call</u> case.
Argument rule	<u>VUAR</u> 369	<u>VUAR</u>	<u>634</u>	Rule simplified thanks to the addition of <u>VUDA</u> (see below) for the more complex case. Clause 3 (redundant) removed. Clause 4 moved to constraint on <u>Address</u> expression.
Class-Level Call rule	(NEW)	<u>VUCC</u>	<u>636</u>	Separating class validity from more complex rules.
Object Test rule	(NEW)	<u>VUOT</u>	<u>659</u>	New rule, covering new construct.
Descendant Argument rule	(<u>VUAR</u> , p. 367)	<u>VUDA</u>	<u>667</u>	Rule split away from <u>VUAR</u> to separate more advanced cases from simple ones.
Single-Level Call rule (previously: no name)	<u>VUCS</u> 367	<u>VUSC</u>	<u>668</u>	Code change; name added.
General Call rule (previously: Call rule)	<u>VUGV</u> 367	<u>VUGC</u>	<u>681</u>	Name change for consistency.
(No name)	<u>VWEQ</u>	Removed		No more conformance constraint on equality.
Call Agent rule	(NEW)	<u>VPCA</u>	<u>754</u>	Agents are new.
Inline Agent rule	(NEW)	<u>VPIA</u>	<u>755</u>	Inline agents are new.
Inline Agent requirements	(NEW)	<u>VPIR</u>	<u>756</u>	Inline agents are new.
Bracket Expression rule	(NEW)	<u>VWBE</u>	<u>781</u>	Bracket expressions are new..
Manifest Type rule	(NEW)	<u>VWM</u> <u>Q</u>	<u>791</u>	Manifest types for expressions are new..
(No name)	<u>VWMS</u> 390	Removed		Now handled through syntax and definition of <u>Line_wrapping_part</u> .
Manifest Array rule	<u>VWMA</u> 393	Removed		No longer necessary thanks to manifest tuples. Backward compatibility enforced through rule that manifest tuples conform to manifest arrays.
Identifier rule (previously: no name)	<u>VIRW</u> 418	<u>VIID</u>	<u>891</u>	Code and name change.

Changes from early versions

G.1 OVERVIEW

The previous appendix summarized the history of Eiffel versions and described the changes from Eiffel 3, as described in the first edition of this book, to Eiffel 5.

The present discussion recalls briefly what had changed from the very first incarnations of Eiffel, especially ISE Eiffel 2 — used in the first (1988) edition of the book *Object-Oriented Software Construction*, which was many people’s original introduction to Eiffel — to Eiffel 3. It will provide current Eiffel users with a glimpse of the language’s early evolution.

For more historical background see Appendix [E](#), [A](#) [brief history of Eiffel](#).

G.2 SCOPE OF THE CHANGES

Whereas changes from Eiffel 3 to Eiffel 5 essentially don’t break any existing code, the changes from Eiffel 2 to Eiffel 3 did not guarantee backward compatibility, since it was felt appropriate to tune some of the original constructs. The translation, however, was simple and systematic, enabling ISE to provide a translator that automatically converted most of a system and left only a few items for manual programmer action, such as renaming any identifiers conflicting with new keywords.

The differences were of three kinds:

- Changes to the concrete syntax, improving the consistency of the language and the clarity of software texts.
- Adjustment or clarification of the semantics of a few constructs, taking care of cases which proved confusing, such as the combination of repeated inheritance and redeclaration.
- A few new constructs to increase the expressive power of the language.

G.3 OLDER POST-OOSC-1 EXTENSIONS

Prior to Eiffel 3, the following mechanisms were added in versions 2.1 (mid-1988), 2.2 (mid-1989) and 2.3 (mid-1990) of ISE Eiffel, after the original publication of the book *Object-Oriented Software Construction* (hereafter *OOSC-1*) in March of 1988:

- Constrained genericity, enabling a generic class to place certain requirements, expressed through inheritance, on possible actual generic parameters. (*OOSC-1* in fact mentioned this, but only in an exercise.)
- The **Indexing** clause (now **Notes**) for recording important information about a class, to be used by archival, browsing and query tools.
- The **Assignment_attempt**, with its **?=** symbol, for type-safe assignments going against the inheritance hierarchy, widely imitated in other languages.
- Infix and prefix operators, for more flexible call syntax.
- Expanded types, supporting composite objects and avoiding unnecessary dynamic allocation.
- The **Obsolete** clause (in classes and routines) for smooth library evolution.
- **Unique** values to define integer codes without having to choose values.
- The **Multi_branch** instruction for discriminating between a set of cases without using dynamic binding. (This was limited to character and integer values; the extension to intervals came with Eiffel 3, and to strings and type descriptors with Eiffel 5.)
- The boolean operator for implication (**implies**), which was previously expressed through the operator **or else**.
- Support for double-precision reals (type **DOUBLE**, later removed).
- Basic expanded classes from the Kernel Library, defining **BOOLEAN**, **CHARACTER**, **INTEGER**, **REAL** and (then) **DOUBLE**.
- The join mechanism for merging one or more inherited deferred routines with compatible signatures and specifications. (In 2.3 this required a now obsolete keyword, **define**, and an effecting of the resulting features.)
- More flexibility in the interface with other languages, in particular through the introduction of the **\$** symbol (**@** in 2.3).

The rest of this appendix covers changes from Eiffel 2.3 to Eiffel 3, first introduced in 1993.

G.4 SEMICOLONS

Eiffel originally used semicolons as separators. With Eiffel 3, semicolons were made optional in most cases. For a while, the style rules still recommended including them, until it was realized — partly from comments of students in programming classes — that instead of helping readability they obscured software texts, providing no benefit except in the rare case of multiple instructions on a single line. The style rules were revised to reflect this realization that most instruction-separating semicolons are just noise. *“OPTIONAL SEMICOLONS”, 34.10, page 919.*

G.5 FEATURE ADAPTATION

The syntax of the Feature_adaptation subclause, in the Parent clause of an Inheritance part, indicating changes in inherited features, was made more regular by the introduction of a required **end** terminator, consistent with the conventions used elsewhere in the language (routine declarations, control structures). Previously, there was no **end**; this meant that a mistakenly added extra semicolon, for example between a Rename and a Redefine subclauses, could make the construct ambiguous, resulting in minor but annoying syntactical errors. This is now harmless, and semicolons have, as noted, been made mostly irrelevant anyway. *Page 171.*

G.6 SPECIFYING EXPORT STATUS

Eiffel 3 removed the **export** clause which was used, at the beginning of a class, to specify the export regime of every feature of the class. Instead, there may be more than one Feature_clause; each defines the export regime of the features it introduces. If a Feature_clause just begins with the **feature** keyword with no further qualification, all the features it introduces are publicly available. *Chapter 7, Clients and exports, gives all the details of how to set the export status of features.*

To obtain the effect of a secret feature, begin the Feature_clause with

```
feature {NONE}
```

To obtain the effect of a feature available selectively to specified classes, begin the Feature_clause with

```
feature {A, B, C}
```

This also removed the need for the **repeat** subclause (which was part of an **export** clause and served to repeat a parent’s export specification). By default, inherited features keep the export status they had in the parent, unless they are redefined. The status of a redefined feature is determined by the qualification of the **Feature_clause** in which the redefinition occurs. To change the status of an inherited feature that is not redefined, use an **export** subclause in the **Feature_adaptation** clause at the point of inheritance, as in

```
class D inherit
  C
    export
      {NONE} all
      {A, B} remove, count
      {ANY} put
    end
  ...
```

Here all features inherited from *C* are secret, except for *remove* and *count*, available to *A* and *B*, and *put*, available to all clients

G.7 ADAPTING PRECONDITIONS AND POSTCONDITIONS

Another important language improvement affects the rule on adaptation of preconditions and postconditions for redefined routines is now a language mechanism, rather than a purely methodological guideline. In pre-version 3, a **Precondition** or **Postcondition** always appeared in full, even for a redefined routine for which the assertions had not changed. If they did change, you were only supposed to replace an original precondition with a weaker one, or an original postcondition with a stronger one; but the language did not support these rules directly.

“REDECLARATION AND ASSERTIONS”,
10.17, page 283.

It now does. In a redefined routine, an absent Precondition means “keep the original’s precondition”, and similarly for an absent postcondition. You may change these assertions using the forms

```
require else new_precondition_clause
ensure then new_postcondition_clause
```

which yields as new preconditions and postconditions the **or** and **and**, respectively, of the original versions with the added ones, automatically enforcing the rule on precondition weakening and postcondition strengthening.

G.8 REMOVING AMBIGUITIES IN REPEATED INHERITANCE

Separate paths of repeated inheritance may cause a feature to be redefined in different ways. The 2.3 language specification left it to the implementation to resolve the dynamic binding conflicts that may arise in such a case.

To solve this issue, Eiffel 3 introduced a **Select** clause, in the **Inheritance** part for a class. An example, assuming *B* and *D* both inherit a feature *f* from a common ancestor *A* and both redefine it, was:

```
class D inherit
  B
    rename f as bf select bf end
    -- This select the B version for
    -- dynamic binding from A.
  C
    rename f as cf end
  ...
```

A potential ambiguity arises only with calls of the form *a1.f* for *a* declared of type *A* but dynamically attached to an instance of *D*. The **select** resolves this ambiguity by prescribing the use of *bf*, the *B* version, in this case.

Eiffel 5 replaces this mechanism by the mechanism of non-conforming inheritance, slightly less flexible but simpler.

[“THE REPEATED INHERITANCE CONSISTENCY CONSTRAINT”, 16.13, page 463.](#)

G.9 RENAMING, REDEFINING, UNDEFINING AND JOINING

In pre-version 3, it was possible to duplicate an inherited feature by renaming it and keeping the old one under a different name; dynamic binding would then apply to entities of the parent type will trigger the redefined version. This mechanism was difficult to explain and was replaced by the **Select** clause just described (then by non-conforming inheritance). It was in fact unnecessary since repeated inheritance also achieves feature duplication in a more uniform way.

Complementing **Redefine**, a **new clause**, **Undefine**, was introduced to allow de-effecting a feature inherited in effective form, making it deferred. A related constraint was added to prohibit redefining an effective feature into a deferred one, since one may now use undefinition instead.

[“UNDEFINING A FEATURE”, 10.19, page 290.](#)

In an extension and simplification of the language semantics, inheriting two or more deferred features under the same name will yield a single deferred feature. This is known as the join mechanism and is useful to merge abstractions. An essentially equivalent mechanism existed in pre-version 3 but required the inheriting class to effect the features and to mark them using the keyword **define** (not a reserved word any more). These restrictions do not apply any more.

By combining the previous two possibilities, you may merge a set of effective features inherited from parents, one of these features imposing its implementation on the others.

G.10 SYNONYMS

Eiffel 3 introduced the possibility of introducing two or more features with a single declaration, as in

[“SYNONYMS AND MULTIPLE DECLARATION”, 5.18, page 150](#)

```
f1, f2 (...) is ...
```

This is equivalent to duplicate declarations; the features declared together are not otherwise connected. Redefining or renaming one in a proper descendant has no effect on the others.

G.11 FROZEN FEATURES

To preserve not just the specification of a feature (through its assertions) but also its exact implementation in descendants, you may, since Eiffel 3, declare it as **frozen**. This prevents any redefinition in descendants. Combined with the synonym mechanism, as in

[“REDECLARATION RULES”, 10.28, page 312](#)

```
frozen f1, f2 (...) is ...
```

which prevents *f1* from being redefined, but does not so restrict *f2*, this makes it possible to provide both a fixed version and a redefinable one. This scheme can be used for a number of features of the universal class *ANY*, such as *copy*, *close*, *is_equal*, which have both a standard version and one adaptable to any class.

G.12 ANCHORING TO A FORMAL ARGUMENT

In an anchored declaration of the form **like** *anchor*, Eiffel 3 made it possible to use *anchor* not just *Current* or an attribute of the enclosing class, but also, in a routine text, a formal argument of that routine, as in

[“ANCHORED TYPES”, 11.10, page 339](#)

```
equal (some: ANY; other: like some): BOOLEAN is ...  
clone (other: ANY): like other is ...
```

In a call to *equal*, the type of the second actual argument must conform to that of the first. In $y := \textit{clone}(x)$, the type of x must conform to that of y .

G.13 CREATION SYNTAX

Eiffel 1 and 2 permitted a single creation mechanism per class, called under the form $x.\textit{Create}$. Eiffel 3 introduced the notion of multiple creation procedures, and a syntax of the form

<code>!! x.make (arg1, ...)</code>	-- With creation procedure <i>make</i>
<code>!! x</code>	-- Without a creation procedure

or, if D is a descendant of the type declared for x :

<code>! TYPE ! x.make (arg1, ...)</code>
<code>! TYPE ! x</code>

The idea was right but the syntax, with its reliance on a special symbol `!`, departed from the usual principles of clarity of Eiffel. It was replaced in Eiffel 5 by a keyword-based form, using the keyword **create**.

Chapter 20 discusses creation.

G.14 UNIFORM SEMANTICS FOR DOT NOTATION

The $x.\textit{Create}$ notation of Eiffel 1 and 2 was not the only case in which the dot in $x.f$ had special semantics. For all “normal” f , the notation x dot f described the application of feature f to the object attached to f , and required x to be non-void, triggering an exception otherwise.

The convention was different, however, for a small set of special language-defined features: *Create*, *Clone*, *Forget*, *Void* and *Equal*. For these, the operation really applied to the reference value of x , and was legal even if x was void (not attached to any object).

These cases were removed in Eiffel 3 to ensure full consistency: dot notation always has the semantics of an operation applicable to an object, and requires x to be non-void. A void x will cause an exception.

Clone, *Forget*, *Void* and *Equal* are no longer reserved words of the language; instead, the operations use features of the universal class *ANY*, of which all Eiffel classes are descendants. These features’ names (*clone*, *Void* and *equal*) are normal identifiers, and proper descendants of *ANY* may rename the features. The cloning instruction $y.\textit{Clone}(x)$ is now written as the assignment $y := \textit{clone}(x)$. The instruction $x.\textit{Forget}$ is written as the

assignment $x := \text{Void}$. Feature *Void* of class *ANY* returns a reference of type *NONE*, the class that has no instances. The test for a void reference, previously written $x.\text{Void}$, is now $x = \text{Void}$. The object equality test, instead of $x.\text{Equal}(y)$, is now $\text{equal}(x, y)$. Since the routines involved are normal features of *ANY*, descendants may redefine them while, as noted, always retaining their frozen synonyms.

G.15 MANIFEST ARRAYS

In the same way that a *STRING* object may be given in manifest form (such as "*some string value*"), rather than by successive calls to fill its character positions, Eiffel 3 introduced manifest arrays, such as

```
<<val1, val2, ...>>
```

which defines an array by its elements. Complemented in Eiffel 5 by tuples, this provides a simple way to achieve the effect of routines with a variable number of arguments.

G.16 DEFAULT RESCUE

It is often convenient to define a default exception response for those routines which do not have a specific *Rescue* clauses. In pre-version 3, this was done by having a *Rescue* clause at the class level. The rescue clause was not passed on to descendants because of potential conflicts in the case of multiple inheritance.

As simpler and more flexible *convention* was introduced by Eiffel 3. ["THE DEFAULT RESCUE", 26.5, page 694.](#) The universal class *ANY* has a procedure *default_rescue*, which does nothing. Any class may redefine this procedure to perform specific exception handling actions. Any routine with no *Rescue* clause is considered to have a *Rescue* clause that just calls *default_rescue*. This means that any exception occurring in such a routine will lead to the default exception handling mechanism defined at the level of its class.

G.17 EXPANDED CLASSES

As a notational facility, Eiffel 3 made it possible to declare a class as expanded class *E*, implying that any type based on *E* will be expanded. Previously, you could use the type **expanded *T*** based on an existing type *T*, but you couldn't specify that a class gives expanded type by default. ["CLASS HEADER", 4.9, page 124.](#)

G.18 SEMANTICS OF EXPANDED TYPES

In what was probably the only non-trivial modification of an existing semantic property, the effect of an assignment

```
ref := exp
```

where the type of *ref* is a reference type and the type of *exp* is expanded, is specified as creating a new object identical to the value of *exp* (a clone) and attaching *ref* to it.

Table titled “The semantics of conformance reattachment”, page 598.

Previously, no cloning occurred; *ref* would just become attached to the value of *exp*, a sub-object or some other object. This introduced a possibility for objects to contain references to sub-objects of other objects. This possibility, of dubious benefit, appears to have been used rarely if ever; it did, however, considerably complicate the run-time model and the implementation, in particular the garbage collector.

G.19 FREE INFIX AND PREFIX OPERATORS

Infix and prefix operators, restricted in Eiffel 2 to predefined symbols — arithmetic such as `+`, relational such as `<`, boolean such as **and** — now enjoy full syntactic status: you may give an infix or prefix alias to any function with the appropriate signature (no argument for prefix, one argument for infix), and define your own “free operators”, whose symbols must start in Eiffel 3 with one of the four characters `@ # | &`. Eiffel 5 further generalized this to almost arbitrary names.

“OPERATOR FEATURES”, 5.15, page 154.

For compatibility with tradition, boolean operators still use alphabetic keywords (such as **and** and **or else**). They are the only ones, however; integer operators use non-alphabetic symbols: `// and \\` replaced the **div** and **mod** of Eiffel 2.

G.20 OBSOLETE CLAUSE

For consistency, the Obsolete clause of an obsolete routine now appears after the **is** keyword rather than before.

“OBSOLETE FEATURES”, 5.21, page 165.

A class may also have an obsolete clause, indicating that usage of the class as a whole is discouraged — because you have written a better version that is not fully compatible, or just prefer a different class name. The Obsolete clause in this case comes just before the Class_header (that is to say, before **class**, **deferred class** or **expanded class**, but after the Notes clause if any).

G.21 RESERVED WORDS

The following ten names could be used as identifiers in pre-3 Eiffel. They became reserved words with Eiffel 3:

alias, **all**, **creation**, **elseif**, **frozen**, **NONE**, **POINTER**, **select**, **separate**, **strip**

Appendix L lists reserved words. See also, in the appendix before the present one, “LEXICAL AND SYNTACTIC CHANGES”, F.6, page 1070.

The following eleven, decreasing the overall count, ceased to be reserved:

Clone, Create, define, div, elsif, Equal, Forget, mod, name, Nochange, repeat

(Other than not being needed any more, **name** may have been the worst choice of keyword in the history of programming languages, as every Eiffel beginner was bound to use it as identifier in a simple application, then wonder why the compiler was complaining.) The change from **elsif** to **elseif** reflected the general rule that no Eiffel reserved word should use an abbreviation, although in the absence of a proper English word for the associated concept **elseif** remains, to this day, the only reserved word in the language that does not consist of a single English word.

Of the new Eiffel 3 reserved words listed above, several have lost their reserved status in Eiffel 5: **creation** (now merged with **create**, coming back from Eiffel 1 and 2 with a new font style), **select** and **strip**. Don't use them yet as identifiers, however, since compilers such as ISE Eiffel may still for a while support the Eiffel 3 constructs in the sake of compatibility.

The role of **creation** in Eiffel 3 was to introduce the construct **Creators** that lists the creation procedures of a class. It's simpler and clearer to use the same keyword **create** as in creation instructions.

Constructs Creators, page 547, and Creation_ instruction, page 551.

G.22 OTHER LEXICAL CHANGES

To improve readability of manifest number constants (integers, reals), Eiffel 3 introduced the possibility of using underscores to delimit groups of three digits in both the integral and (for a real constant) decimal parts. The commas do not affect the value. For example, 62_525_300.751_6 denotes the same value as 62525300.7516.

“INTEGER CONSTANTS”. 29.5, page 792.; “REAL CONSTANTS”. 29.6, page 792

The representation of special characters uses the percent sign **%** rather than the backslash ****.

“CHARACTER CONSTANTS”. 29.7, page 793.

H

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

An Eiffel tutorial

(in progress)

For a shorter introduction to Eiffel, see ["An Eiffel tutorial", 1, page 3](#)

•

I

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

Eiffel bibliography (not done)

I.1 OVERVIEW

The documents listed below describe various aspects of Eiffel: method, language, diverse user applications, implementation, supporting tools. The order is not chronological. A code beginning with TR- indicates a technical report available from Interactive Software Engineering. Works without an author indication are by the author of the present book.

Not included are the proceedings of the **International Eiffel User Conferences** (ten sessions as of this writing, in Paris, Sydney, San Diego, New Orleans, Ottawa, Santa Barbara, Dortmund). These collections of articles about actual user experiences with Eiffel were distributed to conference participants but have not been republished as yet.

I.2 BOOKS

I.3 ISE MANUALS

.zX "10" "OO" "ISBN: 0-13-629049-3"

Object-Oriented Software Construction, a book published by Prentice-Hall. 534+xviii pages. Explains the Eiffel approach to the design and implementation of high-quality software. !(=====17) .zX 17 RM Version 3, 1991 (=====17) \{.N1 This book.

.N2

.\}

Eiffel: The Language, a book published by Prentice-Hall. Provides a complete description of the language. (=====20) .zX 3 GI Updated version appears in "An Eiffel Collection" as "Invitation to Eiffel". !(=====20) .aX 3 GI *Eiffel: An Introduction*. Presents a brief overview of the language and ISE's environment. =====17 (The material is close to chapter ===== of

.aX "7" "LI" "BOOK"

Eiffel: The Libraries. Describes the Eiffel Libraries of reusable software components. Revised version will be published by Prentice-Hall.

.aX "5" "UM" "BOOK"

Eiffel: The Environment. Shows how to use Eiffel in practice through the tools of ISE's environment (compiling, debugging, browsing). Revised version will be published by Prentice-Hall. !(=====17) \{

.aX "4" "IM"

Eiffel Installation Instructions. Describes the procedure for installing ISE's tools and environment.

.\}

.aX "20" "EC" "BOOK"

An Eiffel Collection. A collection of articles, many of them previously published in journals or conferences, about various Eiffel-related topics. Contains some of the articles in the present list, as indicated below.

.zX "27" "TL" "ISBN: 0-13-498510-9"

Introduction to the Theory of Programming Languages, a book published by Prentice-Hall. 448+xvi pages. Although not devoted to Eiffel, this book on the fundamentals of programming language theory (abstract syntax, denotational and axiomatic semantics, complementarity of methods) may help understand many of the ideas behind Eiffel software development, in particular assertions and typing.

.zX "14" "CO" "Version 4, 1991. (Original version, 1987.) Appears in "An Eiffel Collection"."

Design by Contract. Reviews the Eiffel approach to software reliability, emphasizing assertions, disciplined exceptions and controlled inheritance. Chapter 1 of *Advances in Object-Oriented Software Engineering*, eds. Dino Mandrioli and Bertrand Meyer, Prentice-Hall, 1992.

.eC

From Structured Programming to Object-Oriented Design: The Road to Eiffel. Appeared in *Structured Programming*, Volume 10, Number 1, pages 19-39, January 1989. A free-form discussion of the thinking that led to the design of Eiffel.

.eC

Conversation with Editorial Board Member B.M. Appeared in *Journal of Object-Oriented Programming*, Volume 2, Number 2, pages 41-42, May-June 1989. An interview where the author explains some of the background that led to Eiffel, and his views of the evolution of object-oriented technology.

.eC

The New Culture of Software Development: Reflections on the Practice of Object-Oriented Design. Appeared in *TOOLS 1* (Technology of Object-Oriented Languages and Systems, Paris, November 1989), SOL, Paris, pages 13-23, November 1989. Revised version in *Journal of Object-Oriented Programming*, Volume 3, Number 4, pages 76-81, November-December 1990; also as chapter 2 of *Advances...* (see number =====

above). Discusses object-oriented programming as a new \(\lqcomponent\(\rq culture, a radical departure from the traditional \(\lqproject\(\rq culture. Addresses the managerial consequences of an organization's move to object-oriented technology and software reuse.

.eC

Sequential and Concurrent Object-Oriented Programming. Appeared in *TOOLS 2* (Technology of Object-Oriented Languages and Systems, Paris, 23-26 June 1990), Angkor/SOL, Paris, pages 17-28, June 1990. Justifies and describes a concurrency mechanism for Eiffel, meant to cover parallel, coroutine, real-time, distributed and process control applications.

.eC

Tools for the New Culture: Lessons from the Design of the Eiffel Libraries. Appeared in *Communications of the ACM*, Volume 33, No. 9, pages 69-88, September 1990. Discusses the design and implementation of the Eiffel libraries, and general principles for developing good libraries of reusable software components.

.eC

A Development in Eiffel: Design and Implementation of a Network Simulator, by Cyrille Gindre and Fre*'de*'rique Sada. Appeared in *Journal of Object-Oriented Programming*, Volume 2, Number 2, pages 27-33, May-June 1989. A report on the experience of developing an industrial product with Eiffel at Thomson-CSF. Includes discussion of design issues and measures of productivity and reusability.

My Life with Eiffel, by Koichiro Yoshida; column in *Software Design* magazine, Tokyo, appearing in every issue (monthly) since November 1989. In Japanese.

.zX "6" "RE" "Version 1.2, September 1986." "EC"

Reusability: The Case for Object-Oriented Design; appeared in *IEEE Software*, March 1987. Analyzes the object-oriented approach to software reusability, emphasizing the Eiffel approach through examples.

.zX "8" "GI" "Version 2, 1987."

Genericity versus Inheritance, Proceedings of ACM OOPSLA Conference, Portland, Sept. 1986, SIGPLAN Notices, 21, 10, pp. 391-405; revised version appeared in *Journal of Pascal, Ada and Modula-2*, 1987. Compares the object-oriented notion of inheritance with the genericity mechanism of Ada. Explains how the two concepts were reconciled by the design of Eiffel.

Eiffel: Applying the Principles of Object-Oriented Design. Appeared in *Computer Language*, pages 81-87, May 1988. A short introduction to Eiffel and ISE's environment.

.eC

Bidding Farewell to Globals. Appeared in *Journal of Object-Oriented Programming* (Eiffel column), Volume 1, Number 4, pages 73-76, August-September 1988. An explanation of why global variables, which hamper

software quality, do not exist in Eiffel, and a presentation of Eiffel techniques for sharing information between modules.

.eC

Harnessing Multiple Inheritance. Appeared in *Journal of Object-Oriented Programming* (Eiffel column), Volume 1, Number 5, pages 48-51, November-December 1988.

.eC

You can write, but can you type?. Appeared in *Journal of Object-Oriented Programming* (Eiffel column), Volume 1, Number 6, pages 58-67, March-April 1989. An introductory discussion of what typing means in the object-oriented context.

.zX "18" "ST" "July 1989 (original: Jan. 1989)." "EC"

Static Typing for Eiffel. A detailed technical discussion of some of the more intricate aspects of static typing for object-oriented programming, explaining the design choices made in Eiffel.

Writing Correct Software. Appeared in *Dr. Dobb's Journal*, pages 48-63, February 1990. An explanation of how assertion and exception techniques can aid class correctness.

.eC

Pure Object-Oriented Programming with Eiffel. Appeared in *Programmer's Update*, pages 59-69, February 1990. An interview where the author explains some of the key Eiffel ideas.

.zX "25" "AN" "Version 1, June 1990."

Object-Oriented Analysis: Case Studies, by Jean-Marc Nerson, Tutorial Notes for TOOLS 2 (Technology of Object-Oriented Languages and Systems, Paris, 23-26 June 1990). Describes an object-oriented system analysis method. The notation is Eiffel-based.

.eC

Objective Reality, by Alan Winston. Appeared in *Unix World*, pages 72-75, April 1990. Taken from an article on applications of object-oriented programming, this extract gives the view of a company developing telecommunication applications in Eiffel.

.eC

The Eiffel Environment. Appeared in *Unix Review*, Volume 6, Number 8, pages 44-55, August 1988. Describes the tools supporting software development in ISE's implementation, as they existed in 1988.

.zX "33" "AT" "July 1991"

ArchiText User's Manual. Introduces the general-purpose ArchiText language-customizable editor, developed in Eiffel; a specialized version of this editor exists for Eiffel itself.

.zX "19" "ET" "July 1989 (original: March 1989)."

Eiffel Types. A unified view of the type system (version 2.2).

.zX "2" "BR" "Version 2.2, January 1987."

Eiffel: A Language and Environment for Software Engineering. Appeared

in *Journal of Systems and Software*, 1988. Offers a detailed introduction to the language and ISE's environment as they existed in 1988.

.N1

\(lqAn invitation to Eiffel\)(rq is an updated version.

.N2

Eiffel: Programming for Reusability and Extendibility. Appeared in *ACM SIGPLAN Notices*, Volume 22, Number 2, pages 85-94, February 1987. The first published introduction to Eiffel.

Eiffel: Object-Oriented Design for Software Engineering by Bertrand Meyer, Jean-Marc Nerson and Masanobu Matsuo. Appeared in Proceedings of ESEC 87 (First European Software Engineering Conference), Strasbourg, 8-11 September 1987, Springer-Verlag, LNCS, Berlin-New York, 1987. An overview of the principles of Eiffel, describing the then current state of ISE's implementation.

.zX "28" "AD" "Version 1, December 1990"

Extending Eiffel Toward O-O Analysis and Design by Jean-Marc Nerson. Describes an approach to software systems analysis and design, with the associated BON graphical formalism (Better Object Notation); covers case studies.

.zX "16" "22" "August 1989"

Release 2.2 Overview. Surveys the enhancements and extensions introduced in release 2.2 of Eiffel (August 1989).

.zX "23" "23" "October 1990"

Release 2.3 Overview. Surveys the enhancements and extensions introduced in release 2.3 of Eiffel (October 1990).

Object-Oriented Software Construction, Bertrand Meyer, Prentice Hall. First edition, 1988; second, thoroughly revised and extended edition, 1997. This is not a book about Eiffel per se but about object technology in general, using the Eiffel approach and relying on the Eiffel notation.

Eiffel: The Language, Bertrand Meyer, Prentice Hall, 1999. This serves as both a detailed language description and as the language reference.

Object-Oriented Applications, Prentice Hall, 1994, edited by Bertrand Meyer and Jean-Marc Nerson, is a collection of chapters written by various project leaders from industrial companies (CAD-CAM, telecommunications, AI...) and describing Eiffel projects in detail: system goals, techniques used, issues encountered, architectural decisions, practical status.

Eiffel: An Introduction, Robert Switzer, Prentice Hall, 1993. A short and clear presentation of Eiffel, suitable for anyone having had prior experience in another language. Written by one of the authors of the Eiffel/S system.

Reusable Software: The Base Object-Oriented Component Libraries, Bertrand Meyer, Prentice Hall 1994. A discussion of library design principles as supported by Eiffel, and their application to the EiffelBase libraries.

Seamless Object-Oriented Software Architecture — Analysis and Design of Reliable Systems, Kim Waldén and Jean-Marc Nerson, 1995. A lucid description of issues and principles of object-oriented analysis and design, using ideas close to those of Eiffel. Introduces the BON method (Business Object Notation).

Object-Oriented Software Engineering with Eiffel, Jean-Marc Jézéquel, Addison-Wesley, 1996. Emphasizes the application of the Eiffel method and modern software engineering principles to the development of large, mission-critical systems.

Software Development Using Eiffel — There Can Be Life Other than C++, Richard Wiener, Prentice Hall, 1995. A presentation particularly aimed at readers already familiar with another O-O language such as C++. Richard Wiener has also written two textbooks: an introduction to computer science and programming (*An Object-Oriented Introduction to Computer Science Using Eiffel*, Prentice Hall, 1996) and its sequel, on data structures and algorithms (*Data Structures Using Eiffel*, Prentice Hall, 1997).

Object Structures: Building Object-Oriented Software Components, Jacob Gore. Addison-Wesley, 1996. Covers data structures using Eiffel with an emphasis on abstraction, reusability and the proper use of inheritance.

Two textbooks with the same title, *Object-Oriented Programming in Eiffel*, serve as introductions to programming: by Robert Rist and Robert Terwilliger (Prentice Hall, 1995), with emphasis on software design principles, and by Pete Thomas and Ray Weedon (Addison-Wesley, 1995), with emphasis on data abstraction and Design by Contract.

Object Technology for Scientific Computing — Object-Oriented Numerical Software in Eiffel and C, Paul Dubois, Prentice Hall, 1996. Describes the application of the Eiffel method and language to numerical computation, and the design of the EiffelMath library.

I.4 INFORMATION SOURCES

ISE's home page at <http://eiffel.com> is an extensive repository of information about Eiffel, with numerous introductory presentations on the technology and its application and on-line technology papers on concurrency, multithreading, external interfaces, Eiffel projects etc. It also includes details about the ISE Eiffel environment and its extensive set of tools and libraries.

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

PART VI: REFERENCE

This part of the book is the reference for the Eiffel language.

It contains no new material but only extracts from part **II**.

J Language reference

J.1 INTRODUCTION

This appendix provides the full, uncommented reference for the Eiffel language. It only retains, in section [J.2](#), the formal elements of the language definition appearing in the rest of this book:

- Definitions of technical terms and Eiffel concepts.
- Syntax specifications.
- Validity constraints (with their codes, such as [VVBG](#)).
- Semantic specifications.

The material of section [J.2](#) is entirely extracted from the other chapters of this book, but discards all comments and discussions. The same material also appears, with basic explanations, in the ECMA standard for Eiffel, also appearing in this book as part [VII](#) (starting on page [1161](#)). So the text below contains no new elements, but serves as a concise and complete language reference.

Underlined terms have a precise meaning, introduced in one of the definitions.

J.2 LANGUAGE SPECIFICATION

FROM CHAPTER 2: SYNTAX, VALIDITY AND SEMANTICS

Definition: Syntax, BNF-E 85

Syntax is the set of rules describing the structure of software texts.

The notation used to define Eiffel’s syntax is called **BNF-E**.

Definition: Component, construct, specimen 86

Any class text, or syntactically meaningful part of it, such as an instruction, an expression or an identifier, is called a **component**.

The structure of any kind of components is described by a **construct**. A component of a kind described by a certain construct is called a **specimen** of that construct.

Construct Specimen convention 86

The phrase “an **X**”, where **X** is the name of a construct, serves as a shorthand for “a specimen of **X**”.

Construct Name convention 86

Every construct has a name starting with an upper-case letter and continuing with lower-case letters, possibly with underscores (to separate parts of the name if it uses several English words).

Definition: Terminal, non-terminal, token 87

Specimens of a **terminal construct** have no further syntactical structure. Examples include:

- Reserved words such as **if** and **Result**.
- Manifest constants such as the integer [234](#); symbols such as **;** (semicolon) and **+** (plus sign).
- Identifiers (used to denote classes, features, entities) such as [LINKED_LIST](#) and [put](#) .

The specimens of terminal constructs are called **tokens**.

In contrast, the specimens of a **non-terminal** construct are defined in terms of other constructs.

Definition: Production 89

A **production** is a formal description of the structure of all specimens of a non-terminal construct. It has the form

$\text{Construct} \triangleq \text{right-side}$

where **right-side** describes how to obtain specimens of the **Construct**.

Kinds of production 89

A production is of one of the following three kinds, distinguished by the form of the **right-side**:

- **Aggregate**, describing a construct whose specimens are made of a fixed sequence of parts, some of which may be optional.
- **Choice**, describing a construct having a set of given variants.
- **Repetition**, describing a construct whose specimens are made of a variable number of parts, all specimens of a given construct.

Definition: Aggregate production 90

An **aggregate** right side is of the form $C_1 C_2 \dots C_n$ ($n > 0$), where every one of the C_i is a construct and any contiguous subsequence may appear in square brackets as $[C_i \dots C_j]$ for $1 \leq i \leq j \leq n$.

Every specimen of the corresponding construct consists of a specimen of C_1 , followed by a specimen of C_2 , ..., followed by a specimen of C_n , with the provision that for any subsequence in brackets the corresponding specimens may be absent.

Definition: Choice production 90

A **choice** right side is of the form $C_1 | C_2 | \dots | C_n$ ($n > 1$), where every one of the C_i is a construct.

Every specimen of the corresponding construct consists of exactly one specimen of one of the C_i .

Definition: Repetition production, separator 91

A **repetition** right side is of one of the two forms

$\{C \ \$ \ \dots\}^*$
 $\{C \ \$ \ \dots\}^+$

where **C** and **\$** (the **separator**) are constructs.

Every specimen of the corresponding construct consists of zero or more (one or more in the second form) specimens of **C**, each separated from the next, if any, by a specimen of **\$**.

The following abbreviations may be used if the separator is empty:

C^*
 C^+

Basic syntax description rule 93

Every non-terminal construct is defined by exactly one production.

Definition: Non-production syntax rule 94

A **non-production syntax rule**, marked “(*non-production*)”, is a syntax property expressed outside of the BNF-E formalism.

Textual conventions 94

The syntax (BNF-E) productions and other rules of the Standard apply the following conventions:

- 1 • Symbols of BNF-E itself, such as the vertical bars | signaling a choice production, appear in black (non-bold, non-italic).
- 2 • Any construct name appears in **dark green** (non-bold, non-italic), with a first letter in upper case, as **Class**.
- 3 • Any component (Eiffel text element) appears in **blue**.
- 4 • The double quote, one of Eiffel’s special symbols, appears in productions as `''`: a double quote character (blue like other Eiffel text) enclosed in two single quote characters (black since they belong to BNF-E, not Eiffel).
- 5 • All other special symbols appear in double quotes, for example a comma as `","`, an assignment symbol as `":="`, a single quote as `'''` (double quotes black, single quote blue).
- 6 • **Keywords** and other reserved words, such as **class** and **Result**, appear in **bold** (blue like other Eiffel text). They do not require quotes since the conventions avoid ambiguity with construct names: **Class** is the name of a construct, **class** a keyword.
- 7 • Examples of Eiffel comment text appear in non-bold, non-italic (and in blue), as `-- A comment`.
- 8 • Other elements of Eiffel text, such as entities and feature names (including in comments) appear in non-bold *italic* (blue).

The color-related parts of these conventions do not affect the language definition, which remains unambiguous under black-and-white printing (thanks to the letter-case and font parts of the conventions). Color printing is recommended for readability.

`-- Update the value of value.`

Definition: Validity constraint 96

A **validity constraint** on a construct is a requirement that every syntactically well-formed specimen of the construct must satisfy to be acceptable as part of a software text.

Definition: Valid 97

A construct specimen, built according to the syntax structure defined by the construct’s production, is said

to be **valid**, and will be accepted by the language processing tools of any Eiffel environment, if and only if it satisfies the validity constraints, if any, applying to the construct.

Validity: General Validity rule VBGV 98

Every validity constraint relative to a construct is considered to include an implicit supplementary condition stating that every component of the construct satisfies every validity constraint applicable to the component.

Definition: Semantics 99

The **semantics** of a construct specimen that is syntactically legal and valid is the construct's effect on the execution of a system that includes the specimen.

Definition: Execution terminology 101

- **Run time** is the period during which a system is executed.
- The **machine** is the combination of hardware (one or more computers) and operating system through which you can execute systems.
- The machine type, that is to say a certain combination of computer type and operating system, is called a **platform**.
- **Language processing tools** serve to build, manipulate, explore and execute the text of an Eiffel system on a machine.

Semantics: Case Insensitivity principle 102

In writing the letters of an Identifier serving as name for a class, feature or entity, or a reserved word, using the upper-case or lower-case versions has no effect on the semantics.

Definition: Upper name, lower name 102

The **upper name** of an Identifier or Operator *i* is *i* written with all letters in upper case; its **lower name**, *i* with all letters in lower case.

Syntax (non-production): Semicolon Optionality rule 103

In writing specimens of **any** construct defined by a Repetition production specifying the semicolon ";" as separator, it is permitted, without any effect on the syntax structure, validity and semantics of the software, to omit any of the semicolons, or to add a semicolon after the last element.

FROM CHAPTER 3: THE ARCHITECTURE OF EIFFEL SOFTWARE

Definition: Cluster, subcluster, contains directly, contains 108

A **cluster** is a collection of classes, (recursively) other clusters called its **subclusters**, or both. The cluster is said to **contain directly** these classes and subclusters. A cluster **contains** a class *C* if it contains directly either *C* or a cluster that (recursively) contains *C*.

Definition: Terminal cluster, internal cluster 109

A cluster is **terminal** if it contains directly at least one class.

A cluster is **internal** if it contains at least one subcluster.

Definition: Universe 110

A **universe** is a set of classes.

Syntax: Class names 110

Class_name \triangleq Identifier

Validity: Class Name rule VSCN 111

It is valid for a universe to include a class if and only if no other class of the universe has the same upper name.

Semantics: Class name semantics 111

A Class_name *C* appearing in the text of a class *D* denotes the class called *C* in the enclosing universe.

Definition: System, root type name, root procedure name 111

A **system** is defined by the combination of:

- 1 • A universe.
- 2 • A type name, called the **root type name**.
- 3 • A feature name, called the **root procedure name**.

Definition: Type dependency 112

A type *T* **depends** on a type *R* if any of the following holds:

- 1 • *R* is a parent of the base class *C* of *T*.
- 2 • *T* is a client of *R*.
- 3 • (Recursively) there is a type *S* such that *T* depends on *S* and *S* depends on *R*.

Validity: Root Type rule VSRT 112

It is valid to designate a type *TN* as root type of a system of universe *U* if and only if it satisfies the following conditions:

- 1 • *TN* is the name of a stand-alone type *T*.
- 2 • *T* only involves classes in *U*.
- 3 • *T*'s base class is not deferred.
- 4 • The base class of any type on which *T* depends is in *U*.

Validity: Root Procedure rule VSRP 113

It is valid to specify a name *pn* as root procedure name for a system *S* if and only if it satisfies the following conditions:

- 1 • *pn* is the name of a creation procedure *p* of *S*'s root type.
- 2 • *p* has no formal argument.
- 3 • *p* is precondition-free.

Definition: Root type, root procedure, root class 114

In a system *S* of root type name *TN* and root procedure name *pn*, the **root type** is the type of name *TN*, the **root class** is the base class of that root type, and the **root procedure** is the procedure of name *pn* in that class.

Semantics: System execution 114

To **execute** a system on a machine means to cause the machine to apply a creation instruction to the system's root type.

FROM CHAPTER 4: CLASSES**Definition: Current class** 117

The **current class** of a construct specimen is the class in which it appears.

Syntax: Class declarations 119

```
Class_declaration  $\triangleq$  [Notes]
Class_header
[Formal_generics]
[Obsolete]
[Inheritance]
[Creators]
[Converters]
[Features]
[Invariant]
[Notes]
end
```

Syntax: Notes 123

```
Notes  $\triangleq$  note Note_list
Note_list  $\triangleq$  {Note_entry ";" ...}*
Note_entry  $\triangleq$  Note_name Note_values
Note_name  $\triangleq$  Identifier ":"
Note_values  $\triangleq$  {Note_item ";" ...}+
Note_item  $\triangleq$  Identifier | Manifest_constant
```

Semantics: Notes semantics 124

A Notes part has no effect on system execution.

Syntax: Class headers 124

```
Class_header  $\triangleq$  [Header_mark] class Class_name
Header_mark  $\triangleq$  deferred | expanded | frozen
```

Validity: Class Header rule VCCB 126

A Class_header appearing in the text of a class *C* is valid if and only if has either no deferred feature or a Header_mark of the **deferred** form.

Definition: Expanded, frozen, deferred, effective class 127

A class is:

- **Expanded** if its Class_header is of the **expanded** form.
- **Frozen** if its Class_header is of the **frozen** or **expanded** form.
- **Deferred** if its Class_header is of the **deferred** form.
- **Effective** if it is not deferred.

Syntax: Obsolete marks 129

```
Obsolete  $\triangleq$  obsolete Message
Message  $\triangleq$  Manifest_string
```

Semantics: Obsolete semantics 129

Specifying an **Obsolete** mark for a class or feature has no run-time effect.

When encountering such a mark, language processing tools may issue a report, citing the obsolescence Message and advising software authors to replace the class or feature by a newer version.

FROM CHAPTER 5: FEATURES**Definition: Inherited, immediate; origin; redeclaration; introduce** 133

Any feature *f* of a class *C* is of one of the following two kinds:

- 1 • If *C* obtains *f* from one of its parents, *f* is an **inherited** feature of *C*. In this case any declaration of *f* in *C* (adapting the original properties of *f* for *C*) is a **redeclaration**.
- 2 • If a declaration appearing in *C* applies to a feature that is not inherited, the feature is said to be **immediate** in *C*. Then *C* is the **origin** (short for "class of origin") of *f*, and is said to **introduce** *f*.

Syntax: Feature parts 137

```
Features  $\triangleq$  Feature_clause+
Feature_clause  $\triangleq$  feature [Clients] [Header_comment]
Feature_declaration_list
Feature_declaration_list  $\triangleq$  {Feature_declaration ";" ...}*
Header_comment  $\triangleq$  Comment
```

Feature categories: overview 138

Every feature of a class is either an *attribute* or a *routine*.

An attribute is either *constant* or *variable*.

A routine is either a *procedure* or a *function*.

Syntax: Feature declarations 141

```
Feature_declaration  $\triangleq$  New_feature_list Declaration_body
Declaration_body  $\triangleq$  [Formal_arguments] [Query_mark]
[Feature_value]
Query_mark  $\triangleq$  Type_mark [Assigner_mark]
Type_mark  $\triangleq$  ":" Type
```

Feature_value \triangleq [Explicit_value]

[Obsolete]

[Header_comment]

[Attribute_or_routine]

Explicit_value \triangleq "=" Manifest_constant

Syntax: New feature lists 141

New_feature_list \triangleq {New_feature ";" ...}+

New_feature \triangleq [frozen] Extended_feature_name

Syntax: Feature bodies 143

Attribute_or_routine \triangleq [Precondition]

[Local_declarations]

Feature_body

[Postcondition]

[Rescue]

end

Feature_body \triangleq Deferred | Effective_routine | Attribute

Validity: Feature Body rule VFFB 144

A Feature_value is valid if and only if it satisfies one of the following conditions:

- 1 • It has an Explicit_value and no Attribute_or_routine.
- 2 • It has an Attribute_or_routine with a Feature_body of the Attribute kind.
- 3 • It has no Explicit_value and has an Attribute_or_routine with a Feature_body of the Effective_routine kind, itself of the Internal kind (beginning with **do** or **once**).
- 4 • It has no Explicit_value and has an Attribute_or_routine with neither a Local_declarations nor a Rescue part, and a Feature_body that is either Deferred or an Effective_routine of the External kind.

Definition: Variable attribute 145

A Feature_declaration is a **variable attribute** declaration if and only if it satisfies the following conditions:

- 1 • There is no Formal_arguments part.
- 2 • There is a Query_mark part.
- 3 • There is no Explicit_value part.
- 4 • If there is a Feature_value part, it has an Attribute_or_routine with a Feature_body of the Attribute kind.

Definition: Constant attribute 146

A Feature_declaration is a **constant attribute** declaration if and only if it satisfies the following conditions:

- 1 • It has no Formal_arguments part.
- 2 • It has a Query_mark part.
- 3 • There is a Feature_value part including an Explicit_value.

Definition: Routine, function, procedure 147

A Feature_declaration is a **routine** declaration if and only if it satisfies the following condition:

- There is a Feature_value including an Attribute_or_routine, whose Feature_body is of the Deferred or Effective_routine kind.

If a Query_mark is present, the routine is a **function**; otherwise it is a **procedure**.

Definition: Command, query 148

A **command** is a procedure. A **query** is an attribute or function.

Definition: Signature, argument signature of a feature 149

The **signature** of a feature *f* is a pair *argument_types*, *result_type* where *argument_types* and *result_type* are the following sequences of types:

- For *argument_types*: if *f* is a routine, the possibly empty sequence of its formal argument types, in the order of the arguments; if *f* is an attribute, an empty sequence.
- For *result_type*: if *f* is a query, a one-element sequence, whose element is the type of *f*; if *f* is a procedure, an empty sequence.

The *argument_types* part is the feature's **argument signature**.

Feature principle 150

Every feature has an associated identifier.

Any valid call (qualified or unqualified) to the feature can be expressed through this identifier.

Syntax: Feature names 151

Extended_feature_name \triangleq Feature_name [Alias]

Feature_name \triangleq Identifier

Alias \triangleq alias "" Alias_name "" [convert]

Alias_name \triangleq Operator | Bracket

Bracket \triangleq "[]"

Syntax (non-production): Alias Syntax rule 151

The Alias_name of an Alias must immediately follow and precede the enclosing double quote symbols, with no intervening characters (in particular no breaks).

When appearing in such an Alias_name, the two-word operators **and then** and **or else** must be written with exactly one space (but no other characters) between the two words.

Definition: Operator feature, bracket feature, identifier-only 152

A feature is an **operator feature** if its Extended_feature_name *fn* includes an Operator alias, a **bracket feature** if *fn* includes a Bracket alias. It is **identifier-only** if neither of these cases applies.

Definition: Identifier of a feature name 153

The **Identifier** that starts a **Extended_feature_name** is called the **identifier of** that **Extended_feature_name** and, by extension, of the associated feature.

Given a class C and an identifier f , C contains at most one feature of identifier f .

Definition: Same feature name, same operator, same alias 153

Two feature names are considered to be “**the same feature name**” if and only if their identifiers have identical lower names.

Two operators are “**the same operator**” if they have identical lower names.

An **Alias** in an **Extended_feature_name** is “**the same alias**” as another if and only if they satisfy the following conditions:

- They are either the same **Operator** or both **Bracket**.
- If either has a **convert** mark, so does the other.

Syntax: Operators 154

Operator \triangleq **Unary** | **Binary**

Unary \triangleq **not** | **+** | **-** | **Free_unary**

Binary \triangleq **+** | **-** | ***** | **/** | **//** | **** | **^** | **.** | **.** | **<** | **>** | **<=** | **>=** |

and | **or** | **xor** | **and then** | **or else** | **implies** |

Free_binary

Syntax: Assigner marks 155

Assigner_mark \triangleq **assign** **Feature_name**

Validity: Assigner Command rule *VFAC* 156

An **Assigner_mark** appearing in the declaration of a **query** q with n arguments ($n \geq 0$) and listing a **Feature_name** fn , called the **assigner command** for q , is valid if and only if it satisfies the following conditions:

- 1 • fn is the identifier of a command c of the class.
- 2 • c has $n + 1$ arguments.
- 3 • The type of c 's first argument and the result type of q have the same deanchored form.
- 4 • For every i in $1..n$, the type of the $i+1$ -st argument of c and the type of the i -th argument of q have the same deanchored form.

Definition: Synonym 159

A **synonym** of a feature of a class C is a feature with a different **Extended_feature_name** such that both names appear in the same **New_feature_list** of a **Feature_declaration** of C .

Definition: Unfolded form of a possibly multiple declaration 159

The **unfolded form** of a **Feature_declaration** listing one or more feature names, as in:

$$f_1, f_2, \dots, f_n \text{ declaration_body} \quad (n \geq 1)$$

where each f_i is a **New_feature**, is the corresponding sequence of declarations naming only one feature each, and with identical declaration bodies, as in:

$$f_1 \text{ declaration_body}$$

$$f_2 \text{ declaration_body}$$

$$\dots$$

$$f_n \text{ declaration_body}$$
Validity: Feature Declaration rule *VFFD* 162

A **Feature_declaration** appearing in a class C is valid if and only if it satisfies all of the following conditions for every declaration of a feature f in its unfolded form:

- 1 • The **Declaration_body** describes a feature which, according to the rules given earlier, is one of: variable attribute, constant attribute, procedure, function.
- 2 • f does not have the same feature name as any other feature introduced in C (in particular, any other feature of the unfolded form).
- 3 • If f has the same feature name as the final name of any inherited feature, the **Declaration_body** satisfies the Redeclaration rule.
- 4 • If the **Declaration_body** describes a deferred feature, then the **Extended_feature_name** of f is not preceded by **frozen**.
- 5 • If the **Declaration_body** describes a once function, the result type is stand-alone.
- 6 • Any anchored type for an argument is detachable.
- 7 • The **Alias** clause, if present, is alias-valid for f .

Validity: Alias Validity rule *VFAV* 163

An **Alias** clause is alias-valid for a feature f of a class C if and only if it satisfies the following conditions:

- 1 • If it lists an **Operator** op : f is a query; no other query of C has an **Operator** alias using the same operator and the same number of arguments; and either: op is a **Unary** and f has no argument, or op is a **Binary** and f has one argument.
- 2 • If it lists a **Bracket** alias: f is a query with at least one argument, and no other feature of C has a **Bracket** alias.
- 3 • If it includes a **convert** mark: it lists an **Operator** and f has one argument.

FROM CHAPTER 6: THE INHERITANCE RELATION**Syntax: Inheritance parts** 171

Inheritance \triangleq **Inherit_clause**⁺

Inherit_clause \triangleq **inherit** [**Non_conformance**] **Parent_list**

Non_conformance \triangleq {" **NONE** " }

Parent_list \triangleq {**Parent** ";" ... }⁺


```

Parent  $\triangleq$  Class_type [Feature_adaptation]
Feature_adaptation  $\triangleq$  [Undefine]
  [Redefine]
  [Rename]
  [New_exports]
  [Select]
end

```

Syntax (non-production): Feature adaptation 171
A `Feature_adaptation` part must include at least one of the optional components.

Definition: Parent part for a type, for a class 171
If a `Parent` part p of an `Inheritance` part lists a `Class_type` T , p is said to be a `Parent` part for T , and also for the base class of T .

Validity: Class ANY rule *VHCA* 173
Every system must include a non-generic class called *ANY*.

Validity: Universal Conformance principle *VHUC* 173
Every type conforms to *ANY*.

Definition: Unfolded Inheritance Part of a class 174
Any class C has an **Unfolded Inheritance Part** defined as follows:

- 1 • If C has an `Inheritance` part: that part.
- 2 • Otherwise: an `Inheritance` part of the form `inherit ANY`.

Definition: Multiple, single inheritance 175
A class has **multiple inheritance** if it has an Unfolded Inheritance Part with two or more `Parent` parts. It has **single inheritance** otherwise.

Definition: Inherit, heir, parent 176
A class C **inherits** from a type or class B if and only if C 's Unfolded Inheritance Part contains a Parent part for B .

B is then a **parent** of C (“parent type” or “parent class” if there is any ambiguity), and C an **heir** (or “heir class”) of B . Any type of base class C is also an heir of B (“heir type” in case of ambiguity).

Definition: Conforming, non-conforming parent 176
A parent B in an `Inheritance` part is **non-conforming** if and only if every Parent part for B in the clause appears in an Inherit_clause with a `Non_conformance` marker. It is **conforming** otherwise.

Definition: Ancestor types of a type, of a class 177
The **ancestor types** of a type CT of base class C include:

- 1 • CT itself.
- 2 • (Recursively) The result of applying CT 's generic substitution to the ancestor types of every parent type for C .

The ancestor types of a *class* are the ancestor types of its current type.

Definition: Ancestor, descendant 177
Class A is an **ancestor** of class B if and only if A is the base class of an ancestor type of B .

Class B is a **descendant** of class A if and only if A is an ancestor of B .

Definition: Proper ancestor, proper descendant 177
The **proper ancestors** of a class C are its ancestors other than C itself. The **proper descendants** of a class B are its descendants other than B itself.

Validity: Parent rule *VHPR* 178
The Unfolded Inheritance Part of a class D is valid if and only if it satisfies the following conditions:

- 1 • In every Parent part for a class B , B is not a descendant of D .
- 2 • No conforming parent is a frozen class.
- 3 • If two or more `Parent` parts are for classes which have a common ancestor A , D meets the conditions of the Repeated Inheritance Consistency constraint for A .
- 4 • At least one of the `Parent` parts is conforming.
- 5 • No two ancestor types of D are different generic derivations of the same class.
- 6 • Every `Parent` is generic-creation-ready.

Syntax: Rename clauses 183

```

Rename  $\triangleq$  rename Rename_list
Rename_list  $\triangleq$  {Rename_pair "," ... }+
Rename_pair  $\triangleq$  Feature_name as Extended_feature_name

```

Validity: Rename Clause rule *VHRC* 185
A `Rename_pair` of the form *old_name as new_name*, appearing in the `Rename` subclause of the `Parent` part for B in a class C , is valid if and only if it satisfies the following conditions:

- 1 • *old_name* is the final name of a feature f of B .
- 2 • *old_name* does not appear as the first element of any other `Rename_pair` in the same `Rename` subclause.
- 3 • *new_name* satisfies the Feature Name rule for C .
- 4 • The Alias of *new_name*, if present, is alias-valid for the version of f in C .

Semantics: Renaming principle 185
Renaming does not affect the semantics of an inherited feature.

Definition: Final name, extended final name, final name set 186
Every feature f of a class C has an **extended final name** in C , an Extended_feature_name, and a **final name**, a Feature_name, defined as follows:

- 1 • The final name is the identifier of the extended final name.
- 2 • If f is immediate in C , its extended final name is the Extended_feature_name under which C declares it.
- 3 • If f is inherited, f is obtained from a feature of a parent B of C . Let extended_parent_name be (recursively) the extended final name of that feature in B , and parent_name its final name of f in B . Then the extended final name of f in C is:
 - If the Parent part for B in C contains a Rename_pair of the form rename parent_name as new_name: new_name.
 - Otherwise: extended_parent_name.

The final names of all the features of a class constitute the **final name set** of a class.

Definition: Inherited name 186

The **inherited name** of a feature obtained from a feature f of a parent B is the final name of f in B .

Definition: Declaration for a feature 188

A Feature_declaration in a class C , listing a Feature_name fn , is a **declaration for** a feature f if and only if fn is the final name of f in C .

FROM CHAPTER 7: CLIENTS AND EXPORTS

Definition: Client relation between classes and types 192

A class C is a **client** of a type S if some ancestor of C is a simple client, an expanded client or a generic client of S .

Definition: Client relation between classes 193

A class C is a **client of a class** B if and only if C is a client of a type whose base class is B .

The same convention applies to the simple client, expanded client and generic client relations.

Definition: Supplier 193

A type or class S is a **supplier** of a class C if C is a client of S , with corresponding variants: simple, expanded, generic, indirect.

Definition: Simple client 194

A class C is a **simple client** of a type S if, in C , S is the type of some entity or expression or the Explicit_creation_type of a Creation_instruction, or is one of the Constraining_types of a formal generic parameter of C , or is involved in the Type of a Non_object_call or of a Manifest_type.

Definition: Expanded client 196

A class C is an **expanded client** of a type S if S is an expanded type and some attribute of C is of type S .

Definition: Generic client, generic supplier 199

A class C is a **generic client** of a type S if for some generically derived type T of the form $B[... , S, ...]$ one of the following holds:

- 1 • C is a client of T .
- 2 • T is a parent type of an ancestor of C .

Definition: Indirect client 200

A class A is an **indirect client** of a type S of base class B if there is a sequence of classes $C_1 = A, C_2, \dots, C_n = B$ such that $n > 2$ and every C_i is a client of C_{i+1} for $1 \leq i < n$.

The indirect forms of the simple client, expanded client and generic client relations are defined similarly.

Definition: Client set of a Clients part 207

The **client set** of a Clients part is the set of descendants of every class of the universe whose name it lists.

By convention, the client set of an absent Clients part includes all classes of the system.

Syntax: Clients 208

$Clients \triangleq \{ " \{ " Class_list " \} "$
 $Class_list \triangleq \{ Class_name " , " \dots \} ^+$

Syntax: Export adaptation 209

$New_exports \triangleq \{ export New_export_list "$
 $New_export_list \triangleq \{ New_export_item " ; " \dots \} ^+$
 $New_export_item \triangleq Clients [Header_comment]$
 $Feature_set$
 $Feature_set \triangleq Feature_list | all$
 $Feature_list \triangleq \{ Feature_name " , " \dots \} ^+$

Validity: Export List rule VLEL 210

A New_exports clause appearing in class C in a Parent part for a parent B , of the form

```
export
  { class_list1 } feature_set1
  ...
  { class_listn } feature_setn
```

is valid if and only if for every feature_set_i (for i in the interval $1..n$) that is a Feature_list (rather than all):

- 1 • Every element of the list is the final name of a feature of C inherited from B .
- 2 • No feature name appears more than once in any such list.

Definition: Client set of a feature 210

The **client set** of a feature f of a class C , of final name fname, includes the following classes (for all cases that match):

- 1 • If f is introduced or redeclared in C : the client set of the Feature_clause of the declaration for f in C .
- 2 • If f is inherited: the union of the client sets (recursively) of all its precursors from conforming parents.

- 3 • If the **Feature_set** of one or more **New_exports** clauses of **C** includes *fname* or **all**, the union of the client sets of their **Clients** parts.

Definition: Available for call, available 211

A feature *f* is **available for call**, or just **available** for short, to a class **C** or to a type based on **C**, if and only if **C** belongs to the client set of *f*.

Definition: Exported, selectively available, secret 211
The export status of a feature of a class is one of the following:

- 1 • The feature may be available to all classes. It is said to be **exported**, or **generally available**.
- 2 • The feature may be available to specific classes (other than **NONE** and **ANY**) only. In that case it is also available to the descendants of all these classes. Such a feature is said to be **selectively available** to the given classes and their descendants.
- 3 • Otherwise the feature is available only to **NONE**. It is then said to be **secret**.

Definition: Secret, public 214

A property of a class text is **secret** if and only if it involves any of the following, describing information on which client classes cannot rely to establish their correctness:

- 1 • Any feature that is not available to the given client, unless this is overridden by the next case.
- 2 • Any feature that is not available for creation to the given client, unless this is overridden by the previous case.
- 3 • The body and rescue clause of any feature, except for the information that the feature is external or **Once** and, in the last case, its once keys if any.
- 4 • For a query without formal arguments, whether it is implemented as an attribute or a function, except for the information that it is a constant attribute.
- 5 • Any Assertion_clause that (recursively) includes secret information.
- 6 • Any parent part for a non-conforming parent (and as a consequence the very presence of that parent).
- 7 • The information that a feature is frozen.

Any property of a class text that is not secret is **public**.

Definition: Incremental contract view, short form 215

The **incremental contract view** of a class, also called its **short form**, is a text with the same structure as the class but retaining only public properties.

Definition: Contract view, flat-short form 216

The **contract view** of a class, also called its **flat-short form**, is a text following the same conventions as the incremental contract view form but extended to include information about inherited as well as

immediate features, the resulting combined preconditions and postconditions and the unfolded form of the class invariant including inherited clauses.

FROM CHAPTER 8: ROUTINES

Definition: Formal argument, actual argument 219

Entities declared in a routine to represent information passed by callers are the routine's **formal arguments**.

The corresponding expressions in a particular call to the routine are the call's **actual arguments**.

Syntax: Formal argument and entity declarations

220

Formal_arguments \triangleq "(" Entity_declaration_list ")"
Entity_declaration_list \triangleq {Entity_declaration_group ";"
... }⁺
Entity_declaration_group \triangleq Identifier_list Type_mark
Identifier_list \triangleq {Identifier ";", ... }⁺

Validity: Formal Argument rule VRFA 220

Let *fa* be the **Formal_arguments** part of a routine *r* in a class **C**. Let *formals* be the concatenation of every **Identifier_list** of every **Entity_declaration_group** in *fa*. Then *fa* is valid if and only if no **Identifier e** appearing in *formals* is the final name of a feature of **C**.

Validity: Entity Declaration rule VRED 221

Let *el* be an **Entity_declaration_list**. Let *identifiers* be the concatenation of every **Identifier_list** of every **Entity_declaration_group** in *el*. Then *el* is valid if and only if no **Identifier** appears more than once in the list *identifiers*.

Syntax: Routine bodies 222

Deferred \triangleq **deferred**
Effective_routine \triangleq Internal | External
Internal \triangleq Routine_mark Compound
Routine_mark \triangleq **do** | **Once**
Once \triangleq **once** ["("Key_list ")"]
Key_list \triangleq {Manifest_string ";", ... }⁺

Definition: Once routine, once procedure, once function 223

A **once routine** is an **Internal** routine *r* with a **Routine_mark** of the **Once** form.

If *r* is a procedure it is also a **once procedure**; if *r* is a function, it is also a **once function**.

Syntax: Local variable declarations 225

Local_declarations \triangleq **local** [Entity_declaration_list]

Validity: Local Variable rule VRLV 226

Let *ld* be the **Local_declarations** part of a routine *r* in a class **C**. Let *locals* be the concatenation of every **Identifier_list** of every **Entity_declaration_group** in *ld*. Then *ld* is valid if and only if every **Identifier e** in *locals* satisfies the following conditions:

- 1 • No feature of **C** has *e* as its final name.

- 2 • No formal argument of *r* has *e* as its **Identifier**.

Definition: Local variable 226

The local variables of a routine include all **entities** declared in its **Local_declarations** part, if any, and, if it is a query, the predefined entity **Result**.

Syntax: Instructions 228

Compound \triangleq {Instruction ";" ... }*
 Instruction \triangleq Creation_instruction | Call | Assignment |
 Assigner_call | Conditional | Multi_branch | Loop | Debug
 | Precursor | Check | Retry

FROM CHAPTER 9: CORRECTNESS AND CONTRACTS

Syntax: Assertions 232

Precondition \triangleq **require** [else] Assertion
 Postcondition \triangleq **ensure** [then] Assertion [Only]
 Invariant \triangleq **invariant** Assertion
 Assertion \triangleq {Assertion_clause ";" ... }*
 Assertion_clause \triangleq [Tag_mark]
 Unlabeled_assertion_clause
 Unlabeled_assertion_clause \triangleq Boolean_expression |
 Comment
 Tag_mark \triangleq Tag ":"
 Tag \triangleq Identifier

Syntax (non-production): Assertion Syntax rule 233

An **Assertion** without a **Tag_mark** may not begin with any of the following:

- 1 • An opening parenthesis "(".
- 2 • An opening bracket "[".
- 3 • A non-**keyword** **Unary** operator that is also **Binary**.

Definition: Precondition, postcondition, invariant 234

The **precondition** and **postcondition** of a feature, or the **invariant** of a class, is the **Assertion** of, respectively, the corresponding **Precondition**, **Postcondition** or **Invariant** clause if present and non-empty, and otherwise the assertion **True**.

Definition: Contract, subcontract 236

Let *pre* and *post* be the precondition and postcondition of a feature *f*. The **contract** of *f* is the pair of assertions [*pre*, *post*].

A contract [*pre*', *post*'] is said to be a **subcontract** of [*pre*, *post*] if and only if *pre* implies *pre*' and *post*' implies *post*.

Validity: Precondition Export rule VAPE 237

A **Precondition** of a feature *r* of a class *S* is valid if and only if every feature *f* appearing in every **Assertion_clause** of its **unfolded form** *u* satisfies the following two conditions for every class *C* to which *r* is available:

- 1 • If *f* appears as **feature of a call** in *u* or any of its subexpressions, *f* is available to *C*.

- 2 • If *u* or any of its subexpressions uses *f* as creation procedure of a **Creation_expression**, *f* is available for **creation** to *C*.

Definition: Availability of an assertion clause 238

An **Assertion_clause** *a* of a routine **Precondition** or **Postcondition** is **available** to a class *B* if and only if all the features involved in the Equivalent Dot Form of *a* are **available** to *B*.

Syntax: "Old" postcondition expressions 239

Old \triangleq **old** Expression

Validity: Old Expression rule VAOX 239

An **Old** expression *oe* of the form **old** *e* is valid if and only if it satisfies the following conditions:

- 1 • It appears in a **Postcondition** part *post* of a feature.
- 2 • It does not involve **Result**.
- 3 • Replacing *oe* by *e* in *post* yields a valid **Postcondition**.

Semantics: Old Expression Semantics, associated variable, associated exception marker 240

The effect of including an **Old** expression *oe* in a **Postcondition** of an **effective feature** *f* is equivalent to replacing the semantics of its **Feature_body** by the effect of a call to a fictitious routine possessing a local variable *av*, called the **associated variable** of *oe*, and semantics defined by the following succession of steps:

- 1 • Evaluate *oe*.
- 2 • If this evaluation **triggers** an **exception**, record this event in an **associated exception marker** for *oe*.
- 3 • Otherwise, assign the value of *oe* to *av*.
- 4 • Proceed with the original semantics.

Semantics: Associated Variable Semantics 241

As part of the evaluation of a postcondition clause, the evaluation of the **associated variable** of an **Old** expression:

- 1 • **Triggers** an **exception** of type **OLD_EXCEPTION** if an **associated exception marker** has been recorded.
- 2 • Otherwise, yields the value to which the variable has been set.

Syntax: "Only" postcondition clauses 242

Only \triangleq **only** [Feature_list]

Validity: Only Clause rule VAON 243

An **Only** clause appearing in a **Postcondition** of a feature of a class *C* is valid if and only if every **Feature_name** *qn* appearing its **Feature_list** if any satisfies the following conditions:

- 1 • There is no other occurrence of *qn* in that **Feature_list**.
- 2 • *qn* is the **final name** of a **query** *q* of *C*, with no arguments.

- 3 • If C **redeclares** f from a **parent** B , q is not a feature of B .

Definition: Unfolded feature list of an Only clause 244

The **unfolded feature list** of an **Only** clause appearing in a **Postcondition** of a feature f in a class C is the **Feature_list** containing:

- 1 • All the **feature names** appearing in its **Feature_list** if any.
- 2 • If f is the **redeclaration** of one or more features, the **final names** in C of all the features whose names appear (recursively) in their **unfolded Only clauses**.

Definition: Unfolded Only clause 244

The **unfolded Only clause** of a feature f of a class C is a sequence of **Assertion_clause** components of the following form, one for every argument-less query q of C that does not appear in the **unfolded feature list** of the **Only** clause of its **Postcondition** if any:

$$q = (\text{old } q)$$

Definition: Hoare triple notation (total correctness) 247

In definitions of correctness notions for Eiffel constructs, the notation $\{P\} A \{Q\}$ (a mathematical convention, not a part of Eiffel) expresses that any execution of the **Instruction** or **Compound** A started in a state of the computation satisfying the assertion P will terminate in a state satisfying the assertion Q .

Semantics: Class consistency 247

A class C is **consistent** if and only if it satisfies the following conditions:

- 1 • For every **creation procedure** p of C :

$$\{pre_p\} do_p \{INV_C \text{ and then } post_p\} \quad 247$$
- 2 • For every feature f of C **exported generally** or **selectively**:

$$\{INV_C \text{ and then } pre_f\} do_f \{INV_C \text{ and then } post_f\}$$

where INV_C is the **invariant** of C and, for any feature f , pre_f is the **unfolded form** of the precondition of f , $post_f$ the **unfolded form** of its postcondition, and do_f its body.

Syntax: Check instructions 249

Check \triangleq **check** Assertion [Notes] **end**

Definition: Check-correct 250

An **effective** routine r is **check-correct** if, for every **Check** instruction c in r , any execution of c (as part of an execution of r) satisfies its **Assertion**.

Syntax: Variants 251

Variant \triangleq **variant** [Tag_mark] Expression

Validity: Variant Expression rule VAVE 251

A **Variant** is valid if and only if its **variant expression** is of type **INTEGER** or one of its **sized variants**.

Definition: Loop invariant and variant 251

The **Assertion** introduced by the **Invariant** clause of a loop is called its **loop invariant**. The **Expression** introduced by the **Variant** clause is called its **loop variant**.

Definition: Loop-correct 252

A routine is **loop-correct** if every loop it contains, with **loop invariant** INV , **loop variant** VAR , **Initialization** $INIT$, **Exit** condition $EXIT$ and body (**Compound** part of the **Loop_body**) $BODY$, satisfies the following conditions:

- 1 • $\{\text{true}\} INIT \{INV\}$
- 2 • $\{\text{true}\} INIT \{VAR \geq 0\}$
- 3 • $\{INV \text{ and then not } EXIT\} BODY \{INV\}$
- 4 • $\{INV \text{ and then not } EXIT \text{ and then } (VAR = v)\} BODY \{0 \leq VAR < v\}$

Definition: Correctness (class) 253

A class is **correct** if and only if it is **consistent** and every routine of the class is **check-correct**, **loop-correct** and **exception-correct**.

Definition: Local unfolded form of an assertion 254

The **local unfolded form** of an assertion a — a **Boolean_expression** — is the **Equivalent Dot Form** of the expression that would be obtained by applying the following transformations to a in order:

- 1 • Replace any **Only** clause by the corresponding **unfolded Only clause**.
- 2 • Replace any **Old** expression by its **associated variable**.
- 3 • Replace any clause of the **Comment** form by **True**.

Semantics: Evaluation of an assertion 255

To **evaluate** an assertion consists of computing the **value** of its **unfolded form**.

Semantics: Assertion monitoring 256

The execution of an Eiffel system may evaluate, or **monitor**, specific kinds of assertion, and loop **variants**, at specific stages:

- 1 • Precondition of a routine r : on starting a call to r , after argument evaluation and prior to executing any of the instructions in r 's body.
- 2 • Postcondition of a routine r : on successful (not interrupted by an exception) completion of a call to r , after executing any applicable instructions of r .
- 3 • Invariant of a class C : on both start and termination of a **qualified** call to a routine of C .
- 4 • Invariant of a loop: after execution of the **Initialization**, and after every execution (if any) of the **Loop_body**.
- 5 • Assertion in a **Check** instruction: on any execution of that instruction.

- 6 • Variant of a loop: as with the loop invariant.

Semantics: Assertion violation 256

An **assertion violation** is the occurrence at run time, as a result of assertion monitoring, of any of the following:

- An assertion (in the strict sense of the term) evaluating to false.
- A loop variant found to be negative.
- A loop variant found, after the execution of a Loop_body, to be no less than in its previous evaluation.

Semantics: Assertion semantics 257

In the absence of assertion violations, assertions have no effect on system execution other than through their evaluation as a result of assertion monitoring.

An assertion violation causes an exception of type `ASSERTION_VIOLATION` or one of its descendants.

Semantics: Assertion monitoring levels 258

An Eiffel implementation must provide facilities to enable or disable assertion monitoring according to some combinations of the following criteria:

- Statically (at compile time) or dynamically (at run time).
- Through control information specified within the Eiffel text or through outside elements such as a user interface or configuration files.
- For specific kinds as listed in the definition of assertion monitoring: routine preconditions, routine postconditions, class invariants, loop invariants, Check instructions, loop variants.
- For specific classes, specific clusters, or the entire system.

The following combinations must be supported:

- 1 • Statically disable all monitoring for the entire system.
- 2 • Statically enable precondition monitoring for an entire system.
- 3 • Statically enable precondition monitoring for specified classes.
- 4 • Statically enable all assertion monitoring for an entire system.

FROM CHAPTER 10: FEATURE ADAPTATION

Definition: Redeclare, redeclaration 263

A class **redeclares** an inherited feature if it redefines or effects it.

A declaration for a feature f is a **redeclaration** of f if it is either a redefinition or an effecting of f .

Definition: Unfolded form of an assertion 287

The **unfolded form** of an assertion a of local unfolded form ua in a class C is the following Boolean expression:

- 1 • If a is the invariant of C and C has n parents for some $n \geq 1$: up_1 and ... and up_n and then ua , where up_1, \dots, up_n are (recursively) the unfolded forms of the invariants of these parents, after application of any feature renaming specified by C 's corresponding Parent clauses.
- 2 • If a is the precondition of a redeclared feature f : the combined precondition for a .
- 3 • If a is the postcondition of a redeclared feature f : the combined postcondition for a .
- 4 • In all other cases: ua .

Definition: Assertion extensions 288

For a feature f of a class C :

- If C redeclares f with a non-empty Precondition (starting with **require else**), the **precondition extension** of f in C is the corresponding Assertion.
- If C redeclares f with a non-empty Postcondition (starting with **ensure then**), the **postcondition extension** of f in C is the corresponding Assertion.

In all other cases, the precondition extension of f in C is **False** and the postcondition extension of f in C is **True**.

Definition: Covariance-aware form of an assertion extension 289

The **covariance-aware form** of an inherited assertion a is:

- 1 • If the enclosing routine has one or more arguments x_1, \dots, x_n redefined covariantly to types U_1, \dots, U_n : the assertion $(\{x_1: U_1\} y_1 \text{ and } \dots \text{ and } \{x_n: U_n\} y_n) \text{ and then } a'$, where y_1, \dots, y_n are fresh names and a' is the result of substituting y_i for each corresponding x_i in a .
- 2 • Otherwise: a .

Definition: Combined precondition, postcondition 289

Consider a feature f redeclared in a class C . Let f_1, \dots, f_n ($n \geq 1$) be its versions in parents, pre_1, \dots, pre_n the covariance-aware forms of (recursively) the combined preconditions of these versions, and $post_1, \dots, post_n$ the covariance-aware forms of (recursively) their combined postconditions.

Let pre be the precondition extension of f if defined and not empty, otherwise **False**.

Let post be the postcondition extension of f if defined and not empty, otherwise **True**.

The **combined precondition** of f is the Assertion

$(pre_j \text{ or } \dots \text{ or } pre_n) \text{ or else } pre$

The combined postcondition of f is the **Assertion**

(old pre_j implies $post_j$)

and ... and

(old pre_n implies $post_n$)

and then $post$

Definition: Inherited as effective, inherited as deferred 291

An **inherited feature** is **inherited as effective** if it has at least one **precursor** that is an **effective feature**, and the corresponding **Parent** part does not **undefine** it.

Otherwise the feature is **inherited as deferred**.

Definition: Effect, effecting 291

A class **effects** an inherited feature f if and only if it **inherits f as deferred** and contains a **declaration for f** that defines an **effective feature**.

Definition: Redefine, redefinition 292

A class **redefines** an **inherited feature f** if and only if it contains a **declaration for f** that is not an **effecting of f** .

Such a declaration is then known as a **redefinition of f**

Definition: Name clash 297

A class has a **name clash** if it inherits two or more features from different **parents** under the same **final name**.

Syntax: Precursor 303

Precursor \triangleq **Precursor** [**Parent_qualification**] [**Actuals**]

Parent_qualification \triangleq "{" **Class_name** "}"

Definition: Relative unfolded form of a Precursor 303

In a class C , consider a **Precursor specimen p** appearing in the **redefinition** of a routine r **inherited** from a **parent class B** . Its **unfolded form relative to B** is an **Unqualified_call** of the form r' if p has no **Actuals**, or r' (**args**) if p has actual arguments **args**, where r' is a fictitious feature name added, with a **frozen** mark, as **synonym** for r in B .

Validity: Precursor rule **VDPR** 304

A **Precursor** is valid if and only if it satisfies the following conditions:

- 1 • It appears in the **Feature_body** of a **Feature_declaration** of a feature f .
- 2 • If the **Parent_qualification** part is present, its **Class_name** is the name of a **parent class P** of C .
- 3 • Among the features of C 's parents, limited to features of P if condition 2 applies, exactly one is an **effective feature redefined** by C into f . (The class to which this feature belongs is called the **applicable parent** of the **Precursor**.)
- 4 • The **unfolded form relative** to the applicable parent is, as an **Unqualified_call**, **argument-valid**.

- 5 • It is valid as an **Instruction** if and only if f is a **command**, and as an **Expression** if and only if f is a **query**.

Definition: Unfolded form of a Precursor 306

The **unfolded form** (absolute) of a valid **Precursor** is its **unfolded form relative** to its **applicable parent**.

Semantics: Precursor semantics 306

The effect of a **Precursor** is the effect of its **unfolded form**.

Syntax: Redefinition 307

Redefine \triangleq **redefine** **Feature_list**

Validity: Redefine Subclause rule **VDRS** 307

A **Redefine** subclause appearing in a **Parent part** for a class B in a class C is valid if and only if every **Feature_name $fname$** that it lists (in its **Feature_list**) satisfies the following conditions:

- 1 • $fname$ is the **final name** of a feature f of B .
- 2 • f was not **frozen** in B , and was not a **constant attribute**.
- 3 • $fname$ appears only once in the **Feature_list**.
- 4 • The **Features** part of C contains one **Feature_declaration** that is a **redeclaration** but not an **effecting of f** .
- 5 • If that redeclaration specifies a **deferred feature**, C inherits f **as deferred**.

Semantics: Redefinition semantics 308

The effect in a class C of **redefining** a feature f in a **Parent part for A** is that the **version** of f in C is, rather than its version in A , the feature described by the applicable declaration in C .

Syntax: Undefine clauses 308

Undefine \triangleq **undefine** **Feature_list**

Validity: Undefine Subclause rule **VDUS** 308

An **Undefine** subclause appearing in a **Parent part** for a class B in a class C is valid if and only if every **Feature_name $fname$** that it lists (in its **Feature_list**) satisfies the following conditions:

- 1 • $fname$ is the **final name** of a feature f of B .
- 2 • f was not **frozen** in B , and was not an **attribute**.
- 3 • f was **effective** in B .
- 4 • $fname$ appears only once in the **Feature_list**.
- 5 • Any **redeclaration** of f in C specifies a **deferred feature**.

Semantics: Undefinition semantics 308

The effect in a class C of **undefining** a feature f in a **Parent part for A** is to cause C to inherit from A , rather than the **version** of f in A , a **deferred** form of that version.

Definition: Effective, deferred feature 309

A feature f of a class C is an **effective feature** of C if and only if it satisfies either of the following conditions:

- 1 • C contains a declaration for f whose **Feature_body** is not of the **Deferred** form.
- 2 • f is an **inherited feature**, coming from a **parent** B of C where it is (recursively) effective, and C does not undefine it.

f is **deferred** if and only if it is not effective.

Definition: Effecting 309

A redeclaration into an **effective feature** of a feature **inherited as deferred** is said to **effect** that feature.

Deferred class property 310

A class that has at least one **deferred feature** must have a **Class_header** starting with the keyword **deferred**. The class is then said to be **deferred**.

Effective class property 311

A class whose features, if any, are all effective, is effective unless its **Class_header** starts with the keyword **deferred**.

Definition: Origin, seed 311

Every feature f of a class C has one or more features known as its **seeds** and one or more classes known as its **origins**, as follows:

- 1 • If f is **immediate** in C : f itself as seed; C as a origin.
- 2 • If f is **inherited**: (recursively) all the seeds and origins of its **precursors**.

Validity: Redeclaration rule **VDRD** 313

Let C be a class and g a feature of C . It is valid for g to be a **redeclaration** of a feature f inherited from a **parent** B of C if and only if the following conditions are satisfied.

- 1 • No **effective feature** of C other than f and g has the **same final name**.
- 2 • The **signature** of g **conforms to** the signature of f .
- 3 • The **Precondition** of g , if any, begins with **require else** (not just **require**), and its **Postcondition**, if any, begins with **ensure then** (not just **ensure**).
- 4 • If the redeclaration is a **redefinition** (rather than an **effecting**) the **Redefine** subclause of the **Parent** part for B lists in its **Feature_list** the **final name** of f in B .
- 5 • If f is **inherited as effective**, then g is also effective.
- 6 • If f is an **attribute**, g is an attribute, f and g are both **variable**, and their types are either both expanded or both non-expanded.
- 7 • f and g have either both no alias or the **same alias**.
- 8 • If both features are queries with associated **assigner commands** fp and gp , then gp is the **version of** fp in C .

Definition: Precursor (joined features) 315

A **precursor** of an inherited feature is a **version** of the feature in the **parent** from which it is inherited.

Definition: Transposition to a class or type 316

The **transposition** to a class C of a **specimen** s appearing in a **ancestor** A of C is the specimen obtained from s by replacing every expression by its **Equivalent Dot Form**, then:

- 1 • Replacing the arguments of any **Call** by (recursively) their transposition to C .
- 2 • If s is part of the declaration of a feature g **replicated** in C along a certain **repeated inheritance** path, replacing any **Feature_name** used as name of the **feature of an unqualified call** or as anchor of an **anchored type** by the name resulting from any renaming of the feature along that path.
- 3 • Replacing any **Feature_name** used as name of the feature of an unqualified call or as anchor of an anchored type, if case 2 does not apply, by the result of any renaming along applicable inheritance paths.
- 4 • In every **qualified call** of target t , replacing t by (recursively) its transposition t' to C and the feature of the call by (recursively) its transposition to the type of t' in C .
- 5 • In every **Non_object_call** of target type T , replacing T by (recursively) its transposition T' to C and the feature of the call by (recursively) its transposition to T' .
- 6 • For every entity e , other than an attribute, such that s includes a declaration for e , replacing every occurrence of e by a fresh identifier not used in C .
- 7 • If an ancestor B of C has a **parent type** P of base class A , replacing every occurrence of any generic parameter G of A by (recursively) the transposition to C of the application to G of P 's **generic substitution**.

The transposition to a type T of a **specimen** s appearing in an ancestor of the **base class** C of T is the result of applying the generic substitution of T to the class transposition of s to C .

Definition: Transposition 317

The **direct transposition** to a class B of a **specimen** s appearing in a **parent class** A of B is the specimen obtained from s by replacing every expression by its **Equivalent Dot Form**, then:

- 1 • Replacing the arguments of any **Call** by (recursively) their direct transposition to B .
- 2 • If s is part of the declaration of a feature g **replicated** in B along a certain **repeated inheritance** path, replacing the name of the feature of any **unqualified call** by the name of the feature as resulting from any renaming along that path.

- 3 • In every unqualified call of feature f whose feature name fn appears in a Rename_pair of the form fn as gn in a Parent part for A , such that case 2 does not apply, replacing fn by the identifier of gn .
- 4 • In every qualified call of target t , replacing t by (recursively) its class transposition t' to B and the feature of the call by (recursively) its transposition to the type of t' in B .
- 5 • In every Non_object_call of target type T , replacing T by (recursively) its class transposition T' to B and the feature of the call by (recursively) its transposition to T' .
- 6 • For every entity e , other than an attribute, such that s includes a declaration for e , replacing every occurrence of e by a fresh identifier not used in B .
- 7 • Replacing every occurrence of a formal generic parameter of A by the generic substitution of B 's parent type of base class A .

The **class transposition** to a class C of a specimen s appearing in an ancestor A of C is:

- 8 • If A and C are the same class: s .
- 9 • If A is a parent of an ancestor B of C : (recursively) the transposition to C of the direct transposition of s to B .

The **transposition** to a type T of a specimen s appearing in an ancestor of the base class C of T is the result of applying the generic substitution of T to the class transposition of s to C .

Definition: Unfolded redeclaration 318

Consider a feature f of a class A . The **unfolded redeclaration** of f in an heir C of A is a Feature_declaration defined as follows:

- 1 • If C redeclares f , the declaration of f in C .
- 2 • Otherwise, a Feature_declaration for a feature with the same extended name, the same signature as f and the same Assigner_mark if any, both transposed to C , and an Attribute_or_routine consisting solely of:
 - If f is deferred, a Feature_body of the Deferred kind.
 - If f is an effective routine, a **do** clause whose Compound reads just **Precursor** (if f is a procedure) or **Result := Precursor** (if f is a function), followed by the parenthesized list of formal arguments if any.
 - If f is an attribute, an attribute clause whose Compound reads just **Result := Precursor**.

Validity: Join rule VDJR 319

It is valid for a class C to inherit two different features under the same final name under and only under the following conditions:

- 1 • If both are inherited as effective, C redefines both into a common version.

- 2 • If both are inherited as deferred, the unfolded redeclaration in C of each of them is a valid redeclaration of the other.
- 3 • Otherwise, the unfolded redeclaration in C of the one inherited as effective is a valid redeclaration of the one inherited as deferred.

Semantics: Join Semantics rule 320

Joining in a class C two or more inherited features with the same final name under the terms of the Join rule yields a single feature of C defined as follows:

- 1 • If at least one of these features is effective: its unfolded redeclaration in C .
- 2 • Otherwise: the unfolded redeclaration in C of any of them.

FROM CHAPTER 11: TYPES

Syntax: Types 328

Type \triangleq Class_or_tuple_type | Formal_generic_name | Anchored
 Class_or_tuple_type \triangleq Class_type | Tuple_type
 Class_type \triangleq [Attachment_mark] Class_name [Actual_generics]
 Attachment_mark \triangleq "?" | "!"
 Anchored \triangleq [Attachment_mark] like Anchor
 Anchor \triangleq Feature_name | Current

Semantics: Direct instances and values of a type 329

The **direct instances** of a type T are the run-time objects resulting from: representing a manifest constant, manifest tuple, Manifest_type, agent or Address expression of type T ; applying a creation operation to a target of type T ; (recursively) cloning an existing direct instance of T .

The **values** of a type T are the possible run-time values of an entity or expression of type T .

Semantics: Instance of a type 330

The **instances** of a type TX are the direct instances of any type conforming to TX .

Semantics: Instance principle 331

- Any value of a type T is:
- If T is reference, either a reference to an instance of T or (unless T is attached) a void reference.
 - If T is expanded, an instance of T .

Definition: Instance, direct instance of a class 331

An instance of a class C is an instance of any type T based on C .

A direct instance of C is a direct instance of any type T based on C .

Base principle 332

Any type T proceeds, directly or indirectly, from a Class_or_tuple_type called its **base type**, and an underlying class called its **base class**.

The base class of a type is also the base class of its base type.

Base rule 332

The **base type** of any type is a **Class_or_tuple_type**, with no **Attachment_mark**.

The **base class** of any type other than a **Class_or_tuple_type** is (recursively) the base class of its base type.

The **direct instances** of a type are those of its base type.

Validity: Class Type rule *VTCT* 333

A **Class_type** is valid if and only if it satisfies the following two conditions:

- 1 • Its **Class_name** is the name of a class in the surrounding **universe**.
- 2 • If it has a “?” **Attachment_mark**, that class is not expanded.

Semantics: Type Semantics rule 333

To define the semantics of a type T it suffices to specify:

- 1 • Whether T is **expanded** or **reference**.
- 2 • Whether T , if reference, is **attached** or **detachable**.
- 3 • What is T 's **base type**.
- 4 • If T is a **Class_or_tuple_type**, what are its **base class** and its type parameters if any.

Definition: Base class and base type of an expression

334

Any expression e has a **base type** and a **base class**, defined as the **base type** and **base class** of the **type of** e .

Semantics: Non-generic class type semantics 335

A non-generic class C used as a type (of the **Class_type** category) has the same expansion status as C (i.e. it is expanded if C is an **expanded class**, reference otherwise). It is its own **base type** (after removal of any **Attachment_mark**) and **base class**.

Definition: Expanded type, reference type 337

A type T is **expanded** if and only if it is not a **Formal_generic_name** and the **base class** of its **deanchored form** is an **expanded class**.

T is a **reference type** if it is neither a **Formal_generic_name** nor **expanded**.

Definition: Basic type 338

The basic types are **BOOLEAN**, **CHARACTER** and its **sized variants**, **INTEGER** and its **sized variants**, **REAL** and its **sized variants** and **POINTER**.

Definition: Anchor, anchored type, anchored entity

339

The **anchor** of an anchored type **like anchor** is the **entity anchor**. A declaration of an entity with such a

type is an **anchored declaration**, and the entity itself is an **anchored entity**.

Definition: Anchor set; cyclic anchor 343

The **anchor set** of a type T is the set of **entities** containing, for every anchored type **like anchor involved** in T :

- **anchor**.
- (Recursively) the anchor set of the type of **anchor**.

An entity a of type T is a **cyclic anchor** if the anchor set of T includes a itself.

Definition: Types and classes involved in a type 343

The types **involved** in a type T are the following:

- T itself.
- If T is of the form $a T'$ where a is an **Attachment_mark**: (recursively) the types involved in T' .
- If T is a **generically derived Class_type** or a **Tuple_type**: all the types (recursively) involved in any of its actual parameters.

The **classes** involved in T are the **base classes** of the types involved in T .

Definition: Deanchored form of a type 344

The **deanchored form** of a type T in a class C is the type (**Class_or_tuple_type** or **Formal_generic**) defined as follows:

- 1 • If T is **like Current**: the **current type** of C .
- 2 • If T is **like anchor** where the type AT of **anchor** is not anchored: (recursively) the deanchored form of AT .
- 3 • If T is **like anchor** where the type AT of **anchor** is anchored but **anchor** is not a **cyclic anchor**: (recursively) the deanchored form of AT in C .
- 4 • If T is $a AT$, where a is an **Attachment_mark**: $a DT$, where DT is (recursively) the deanchored form of AT deprived of its **Attachment_mark** if any.
- 5 • If none of the previous cases applies: T after replacement of any actual parameter by (recursively) its deanchored form.

Validity: Anchored Type rule *VTAT* 345

It is valid to use an anchored type AT of the form **like anchor** in a class C if and only if it satisfies the following conditions:

- 1 • **anchor** is either **Current** or the final name of a query of C .
- 2 • **anchor** is not a **cyclic anchor**.
- 3 • The **deanchored form** UT of AT is valid in C .

The **base class** and **base type** of AT are those of UT .

Definition: Attached, detachable 346

A type is **detachable** if its deanchored form is a **Class_type** declared with the **? Attachment_mark**.

A type is **attached** if it is not detachable.

Semantics: Attached type semantics 347

Every run-time value of an attached type is non-void (attached to an object).

Definition: Stand-alone type 347

A **Type** is **stand-alone** if and only if it involves neither any **Anchored type** nor any **Formal_generic_name**.

FROM CHAPTER 12: GENERICITY**Syntax: Actual generic parameters** 350

Actual_generics \triangleq "[" **Type_list** "]"

Type_list \triangleq {**Type** "; ..."}⁺

Syntax: Formal generic parameters 351

Formal_generics \triangleq "[" **Formal_generic_list** "]"

Formal_generic_list \triangleq {**Formal_generic** "; ..."}⁺

Formal_generic \triangleq [**frozen**] **Formal_generic_name** [**Constraint**]

Formal_generic_name \triangleq [**?**] **Identifier**

Validity: Formal Generic rule *VCFG* 351

A **Formal_generics** part of a **Class_declaration** is valid if and only if every **Formal_generic_name** *G* in its **Formal_generic_list** satisfies the following conditions:

- 1 • *G* is different from the name of any class in the universe.
- 2 • *G* is different from any other **Formal_generic_name** appearing in the same **Formal_generics** part.

Definition: Generic class; constrained, unconstrained 352

Any class declared with a **Formal_generics** part (constrained or not) is a **generic class**.

If a formal generic parameter of a generic class is declared with a **Constraint**, the parameter is **constrained**; if not, it is **unconstrained**.

A generic class is itself **constrained** if it has at least one constrained parameter, **unconstrained** otherwise.

Definition: Generic derivation, non-generic type 352

The process of producing a type from a generic class by providing actual generic parameters is **generic derivation**.

A type resulting from a generic derivation is a **generically derived type**, or just **generic type**.

A type that is not generically derived is a **non-generic type**.

Definition: Self-initializing formal parameter 353

A **Formal_generic_parameter** is **self-initializing** if and only if its declaration includes the optional **?** mark.

Definition: Constraint, constraining types of a Formal_generic 353

The **constraint** of a formal generic parameter is its **Constraint** part if present, and otherwise **ANY**.

Its **constraining types** are all the types listed in its **Constraining_types** if present, and otherwise just **ANY**.

Syntax: Generic constraints 357

Constraint \triangleq ">" **Constraining_types** [**Constraint_creators**]

Constraining_types \triangleq **Single_constraint** | **Multiple_constraint**

Single_constraint \triangleq **Type** [**Renaming**]

Renaming \triangleq **Rename** **end**

Multiple_constraint \triangleq "{" **Constraint_list** "}"

Constraint_list \triangleq {**Single_constraint** "; ..."}⁺

Constraint_creators \triangleq **create** **Feature_list** **end**

Validity: Generic Constraint rule *VTGC* 357

A **Constraint** part appearing in the **Formal_generics** part of a class *C* is valid if and only if it satisfies the following conditions for every **Single_constraint** listing a type *T* in its **Constraining_types**:

- 1 • *T* involves no anchored type.
- 2 • If a **Renaming** clause **rename** *rename_list* **end** is present, a class definition of the form **class NEW inherit BT rename** *rename_list* **end** (preceded by **deferred** if the base class of *T* is deferred), where *BT* is the base class of *T*, would be valid.

Definition: Constraining creation features 358

If *G* is a formal generic parameter of a class, the **constraining creators of G** are the features of *G*'s **Constraining_types**, if any, corresponding after possible **Renaming** to the feature names listed in the **Constraining_creators** if present.

Validity: Generic Derivation rule *VTGD* 359

Let *C* be a generic class. A **Class_type CT** having *C* as base class is valid if and only if it satisfies the following conditions for every actual generic parameter *T* and every **Single_constraint U** appearing in the constraint for the corresponding formal generic parameter *G*:

- 1 • The number of **Type** components in *CT*'s **Actual_generics** list is the same as the number of **Formal_generic** parameters in the **Formal_generic_list** of *C*'s declaration.
- 2 • *T* conforms to the type obtained by applying to *U* the generic substitution of *CT*.
- 3 • If *C* is expanded, *CT* is generic-creation-ready.
- 4 • If *G* is a self-initializing formal parameter and *T* is attached, then *T* is a self-initializing type.

Definition: Generic-creation-ready type 360

A type of base class C is **generic-creation-ready** if and only if every actual generic parameter T of its deanchored form satisfies the following conditions:

- 1 • If the specification of the corresponding formal generic parameter includes a **Constraining creators**, the versions in T of the constraining creators for the corresponding formal parameter are creation procedures, available for creation to C , and T is (recursively) generic-creation-ready.
- 2 • If T is expanded, it is (recursively) generic-creation-ready.

Semantics: Generically derived class type semantics

363

A generically derived Class_type of the form $C [\dots]$, where C is a generic class, is expanded if C is an expanded class, reference otherwise. It is its own base type, and its base class is C .

Definition: Base type of a single-constrained formal generic 364

The base type of a constrained **Formal_generic_name** G having as its constraining types a **Single_constraint** listing a type T is:

- 1 • If T is a **Class_or_tuple_type**: T .
- 2 • Otherwise (T is a **Formal_generic_name**): the base type of T if it can be determined by (recursively) case 1, otherwise **ANY**.

Definition: Base type of an unconstrained formal generic 364

The base type of an unconstrained **Formal_generic_name** type is **ANY**.

Definition: Reference or expanded status of a formal generic 365

A **Formal_generic_name** represents a reference type or expanded type depending on the corresponding status of the associated actual generic parameter in a particular generic derivation.

Definition: Current type 365

Within a class text, the **current type** is the type obtained from the current class by providing as actual generic parameters, if required, the class's own formal generic parameters.

Definition: Features of a type 366

The features of a type are the features of its base class.

Definition: Generic substitution 367

Every type T defines a mapping σ from names to types known as its **generic substitution**:

- 1 • If T is generically derived, σ associates to every **Formal_generic_name** the corresponding actual parameter.
- 2 • Otherwise, σ is the identity substitution.

Generic Type Adaptation rule 367

The signature of an entity or feature f of a type T of base class C is the result of applying T 's generic substitution to the signature of f in C .

Definition: Generically constrained feature name 368

Consider a generic class C , a constrained **Formal_generic_name** G of C , a type T appearing as one of the **Constraining_types** for G , and a feature f of name $fname$ in the base class of T . The **generically constrained names** of f for G in C are:

- 1 • If one or more **Single_constraint** clauses for T include a **Rename** part with a clause $fname$ **as** $ename$, where the **Feature_name** part of $ename$ (an **Extended_feature_name**) is $gname$: all such $gname$.
- 2 • Otherwise: just $fname$.

Validity: Multiple Constraints rule *VTMC* 369

A feature of name $fname$ is applicable in a class C to a target x whose type is a **Formal_generic_name** G constrained by two or more types **CONST1**, **CONST2**, ..., if and only if it satisfies the following conditions:

- 1 • At least one of the **CONST_i** has a feature available to C whose generically constrained name for G in C is $fname$.
- 2 • If this is the case for two or more of the **CONST_i**, all the corresponding features are the same.

Definition: Base type of a multi-constraint formal generic type 369

The base type of a multiply constrained **Formal_generic_name** type is a type generically derived, with the same actual parameters as the current class, from a fictitious class with none of the optional parts except for **Formal_generics** and an **Inheritance** clause that lists all the constraining types as parents, with the given **Renaming** clause if any, and resolves any conflicts between potentially ambiguous features by further renaming them to new names not available to developers.

FROM CHAPTER 13: TUPLES**Syntax: Tuple types** 372

$\text{Tuple_type} \triangleq \text{TUPLE} [\text{Tuple_parameter_list}]$
 $\text{Tuple_parameter_list} \triangleq \text{"[" Tuple_parameters "]"}$
 $\text{Tuple_parameters} \triangleq \text{Type_list} \mid \text{Entity_declaration_list}$

Syntax: Manifest tuples 373

$\text{Manifest_tuple} \triangleq \text{"[" Expression_list "]"}$
 $\text{Expression_list} \triangleq \{\text{Expression} \text{" , " } \dots \}^*$

Definition: Type sequence of a tuple type 374

The **type sequence** of a tuple type is the sequence of types obtained by listing its parameters, if any, in the

order in which they appear, every labeled parameter being listed as many times as it has labels.

Definition: Value sequences associated with a tuple type 374

The **value sequences** associated with a tuple type T are sequences of values, each of the type appearing at the corresponding position in T 's **type sequence**.

FROM CHAPTER 14: CONFORMANCE

Definition: Compatibility between types 384

A type is **compatible** with another if it either **conforms** or **converts** to it.

Definition: Compatibility between expressions 384

An expression b is **compatible** with an expression a if and only if b either **conforms** or **converts** to a .

Definition: Expression conformance 384

An expression exp of type **SOURCE** **conforms to** an expression ent of type **TARGET** if and only if they satisfy the following conditions:

- 1 • **SOURCE** **conforms to** **TARGET**.
- 2 • If **TARGET** is attached, so is **SOURCE**.
- 3 • If **SOURCE** is expanded, its version of the function **cloned** from **ANY** is available to the **current class**.

Validity: Signature conformance VNCS 386

A **signature** $t = [B_1, \dots, B_n], [S]$ **conforms to** a signature $s = [A_1, \dots, A_n], [R]$ if and only if it satisfies the following conditions:

- 1 • Each of the two components of t has the same number of elements as the corresponding component of s .
- 2 • Each type in each of the two components of t **conforms to** the corresponding type in the corresponding component of s .
- 3 • Any B_i not identical to the corresponding A_i is **detachable**.

Definition: Covariant argument 387

In a **redeclaration** of a routine, a formal argument is **covariant** if its type differs from the type of the corresponding argument in at least one of the **parents'** **versions**.

Validity: General conformance VNCC 388

Let T and V be two types. V **conforms to** T if and only if one of the following conditions holds:

- 1 • V and T are identical.
- 2 • V **conforms directly** to T .
- 3 • V is **NONE** and T is a **detachable reference type**.
- 4 • V is $B [Y_1, \dots, Y_n]$ where B is a generic class, T is $B [X_1, \dots, X_n]$, and for every X_i the corresponding Y_i is identical to X_i or, if the corresponding formal

parameter does not specify **frozen**, conforms (recursively) to X_i .

5 • For some type U (recursively), V **conforms to** U and U conforms to T .

6 • T or V or both are anchored types appearing in the same class C , and the **deanchored form** of V in C (recursively) conforms to the deanchored form of T .

Definition: Conformance path 389

A **conformance path** from a type U to a type T is a sequence of types T_0, T_1, \dots, T_n ($n \geq 1$) such that T_0 is U , T_n is T , and every T_i (for $0 \leq i < n$) **conforms to** T_{i+1} . This notion also applies to **classes** by considering the associated **base classes**.

Validity: Direct conformance: reference types CN 390

A **Class type** CT of **base class** C **conforms directly** to a **reference type** BT if and only if it satisfies the following conditions:

- 1 • Applying CT 's **generic substitution** to one of the **conforming parents** of C yields BT .
- 2 • If BT is **attached**, so is CT .

Validity: Direct conformance: formal generic NCF 393

Let G be a formal generic parameter of a class C , which in the text of C may be used as a **Formal generic name** type. Then:

- 1 • No type **conforms directly** to G .
- 2 • G **conforms directly** to every type listed in its **constraint**, and to no other type.

Validity: Direct conformance: expanded types CE 396

No type **conforms directly** to an **expanded type**.

Validity: Direct conformance: tuple types VNCT 397

A **Tuple type** U , of type sequence us , **conforms directly** to a type T if and only if T satisfies the following conditions:

- 1 • T is a tuple type, of type sequence ts .
- 2 • The length of us is greater than or equal to the length of ts .
- 3 • For every element X of ts , the corresponding element of us conforms to X .

No type conforms directly to a tuple type except as implied by these conditions.

FROM CHAPTER 15: CONVERTIBILITY

Definition: Conversion procedure, conversion type 401

A procedure whose name appears in a **Converters** clause is a **conversion procedure**.

A type listed in a **Converters** clause is a **conversion type**.

Definition: Conversion query, conversion feature 405

A query whose name appears in a **Converters** clause is a **conversion query**.

A feature that is either a conversion procedure or a conversion query is a **conversion feature**.

No type may both **conform** and **convert** to another.

No type T may **convert** to another through both a **conversion procedure** and a **conversion query**.

That V **converts to** U and U **to** T does not imply that V **converts to** T .

Syntax: Converter clauses 410

Converters \triangleq **convert** Converter_list

Converter_list \triangleq {Converter ",...}+

Converter \triangleq Conversion_procedure | Conversion_query

Conversion_procedure \triangleq Feature_name "(" "{" Type_list
"}" ")"

Conversion_query \triangleq Feature_name ":" "{" Type_list "}"

Validity: Conversion Procedure rule *VYCP* 411

A **Conversion_procedure** listing a **Feature_name** fn and appearing in a class C with **current type** CT is valid if and only if it satisfies the following conditions, applicable to every type **SOURCE** listed in its **Type_list**:

- 1 • fn is the name of a **creation procedure** cp of C .
- 2 • If C is not generic, **SOURCE** does not **conform to** CT .
- 3 • If C is generic, **SOURCE** does not conform to the type obtained from CT by replacing every formal generic parameter by its **constraint**.
- 4 • **SOURCE**'s **base class** is different from the base class of any other **conversion type** listed for a **Conversion_procedure** in the **Converters** clause of C .
- 5 • The specification of the base class of **SOURCE** does not list a **conversion query** specifying a type of base class C .
- 6 • cp has exactly one formal argument, of a type **ARG**.
- 7 • **SOURCE** **conforms to** **ARG**.
- 8 • **SOURCE** **involves no anchored type**.

Validity: Conversion Query rule *VYQC* 413

A **Conversion_query** listing a **Feature_name** fn and appearing in a class C with **current type** CT is valid if and only if it satisfies the following conditions, applicable to every type **TARGET** listed in its **Type_list**:

- 1 • fn is the name of a query f of C .
- 2 • If C is not generic, CT does not **conform to** **TARGET**.
- 3 • If C is generic, the type obtained from CT by replacing every formal generic parameter by its **constraint** does not conform to **TARGET**.

4 • **TARGET**'s **base class** is different from the base class of any other **conversion type** listed for a **Conversion_query** in the **Converters** clause of C .

5 • The specification of the base class of **TARGET** does not list a **conversion procedure** specifying a type of base class C .

6 • f has no formal argument.

7 • The result type of f **conforms to** **TARGET**.

8 • **TARGET** **involves no anchored type**.

Definition: Converting to a class 414

A type T of **base class** CT **converts to** a class C if either:

- The **deanchored form** of T appears as **conversion type** for a procedure in the **Converters** clause of C .
- A type **based on** C appears as conversion type for a query in the **Converters** clause of CT .

Definition: Converting to and from a type 415

A type U of **base class** D **converts to** a **Class_type** T of base class C if and only if either:

- 1 • The **deanchored form** of U is the result of applying the **generic substitution** of the deanchored form of T to a **conversion type** for a procedure cp appearing in the **Converters** clause of C .
- 2 • The deanchored form of T is the result of applying the generic substitution of the deanchored form of U to a conversion type for a query cq appearing in the **Converters** clause of D .

A **Class_type** T **converts from** a type U if and only if U **converts to** T .

Definition: Converting “through” 415

A type U that **converts to** a type T :

- 1 • **Converts to** T **through a procedure** cp if case 1 of the definition of “converting to a type” applies.
- 2 • **Converts to** T **through a query** cq if case 2 of the definition applies.

These terms also apply to “**converting from**” specifications.

Semantics: Conversion semantics 416

Given an expression e of type U and a variable x of type T , where U **converts to** T , the effect of a **conversion attachment** of source e and target x is the same as the effect of either:

- 1 • If U **converts to** T **through a procedure** cp : the creation instruction **create** $x.cp(e)$.
- 2 • If U **converts to** T **through a query** cq : the assignment $x := e.cq$.

Definition: Explicit conversion 418

The Kernel Library class **TYPE** [G] provides a function

adapted alias “[]” (x : G): G

which can be used for any type T and any expression exp of a type U compatible with T to produce a T version of exp , written

$$\{T\} [exp]$$

If U converts to T , this expression denotes the result of converting exp to T , and is called an **explicit conversion**.

Validity: Expression convertibility VYEC 424

An expression exp of type U **converts to** an entity ent of type T if and only if U converts to T through a conversion feature $conv$ satisfying either of the following two conditions:

- 1 • $conv$ is precondition-free.
- 2 • exp statically satisfies the precondition.

Definition: Statically satisfied precondition 425

A feature precondition is **statically satisfied** if it satisfies any of the following conditions:

- 1 • It applies to a boolean, character, integer or real expression involving only constants, states that the expression equals a specific constant value or (in the last three cases) belongs to a specified interval, and holds for that value or interval.
- 2 • It applies to the type of an expression, states that it must be one of a specified set of types, and holds for that type.

Validity: Precondition-free routine VYPF 426

A feature r of a class C is **precondition-free** if it is either:

- 1 • Immediate in C , with either no Precondition clause or one consisting of a single Assertion_clause (introduced by require) whose Boolean_expression is the constant True.
- 2 • Inherited, and such that every precursor of r is (recursively) precondition-free, or r is redeclared in C with a Precondition consisting of a single Assertion_clause (introduced by require else) whose Boolean_expression is the constant True.

FROM CHAPTER 16: REPEATED INHERITANCE

Definition: Repeated inheritance, ancestor, descendant 434

Repeated inheritance occurs whenever (as a result of multiple inheritance) two or more of the ancestors of a class D have a common parent A .

D is then called a **repeated descendant** of A , and A a **repeated ancestor** of D .

Semantics: Repeated Inheritance rule 438

Let D be a class and B_1, \dots, B_n ($n \geq 2$) be parents of D based on classes having a common ancestor A . Let $f_1,$

\dots, f_n be features of these respective parents, all having as one of their seeds the same feature f of A . Then:

- 1 • Any subset of these features inherited by D under the same final name in D yields a single feature of D .
- 2 • Any two of these features inherited under a different name yield two features of D .

Definition: Sharing, replication 439

A repeatedly inherited feature is **shared** if case 1 of the Repeated Inheritance rule applies, and **replicated** if case 2 applies.

Validity: Call Sharing rule VMCS 458

It is valid for a feature f repeatedly inherited by a class D from an ancestor A , such that f is shared under repeated inheritance and not redeclared, to involve a feature g of A other than as the feature of a qualified call if and only if g is, along the corresponding inheritance paths, also shared.

Semantics: Replication Semantics rule 459

Let f and g be two features both repeatedly inherited by a class A and both replicated under the Repeated Inheritance rule, with two respective sets of different names: f_1 and f_2, g_1 and g_2 .

If the version of f in D is the original version from A and either contains an unqualified call to g or (if f is an attribute) is the target of an assignment whose source involves g , the f_1 version will use g_1 for that call or assignment, and the f_2 version will use g_2 .

Syntax: Select clauses 463

Select \triangleq select Feature_list

Validity: Select Subclause rule VMSS 463

A Select subclause appearing in the parent part for a class B in a class D is valid if and only if, for every Feature_name $fname$ in its Feature_list, $fname$ is the final name in D of a feature that has two or more potential versions in D , and $fname$ appears only once in the Feature_list.

Definition: Version 464

A feature g from a class D is a **version** of a feature f from an ancestor of D if f and g have a seed in common.

Definition: Multiple versions 464

A class D has n **versions** ($n \geq 2$) of a feature f of an ancestor A if and only if n of its features, all with different final names in D , are all versions of f .

Validity: Repeated Inheritance Consistency VMRC 466

It is valid for a class D to have two or more versions of a feature f of a proper ancestor A if and only if it satisfies one of the following conditions:

- 1 • There is at most one conformance path from D to A .
- 2 • There are two or more conformance paths, and the Parent clause for exactly one of them in D has a

Select clause listing the name of the version of f from the corresponding parent.

Definition: Dynamic binding version 468

For any feature f of a type T and any type U conforming to T , the **dynamic binding version** of f in U is the feature g of U defined as follows:

- 1 • If f has only one version in U , then g is that feature.
- 2 • If f has two or more versions in U , then the Repeated Inheritance Consistency constraint ensures that either exactly one conformance path exists from U to T , in which case g is the version of f in U obtained along that path, or that a **Select** subclause name a version of f , in which case g is that version.

Definition: Inherited features 470

Let D be a class. Let precursors be the list obtained by concatenating the lists of features of every parent of D ; this list may contain duplicates in the case of repeated inheritance. The list inherited of **inherited features** of D is obtained from precursors as follows:

- 1 • In the list precursors, for any set of two or more elements representing features that are repeatedly inherited in D under the same name, so that the Repeated Inheritance rule yields sharing, keep only one of these elements. The Repeated Inheritance Consistency constraint (sharing case) indicates that these elements must all represent the same feature, so that it does not matter which one is kept.
- 2 • For every feature f in the resulting list, if D undefines f , replace f by a deferred feature with the same signature, specification and header comment.
- 3 • In the resulting list, for any set of deferred features with the same final name in D , keep only one of these features, with assertions and header comment joined as per the Join Semantics rule. (Keep the signature, which the Join rule requires to be the same for all the features involved after possible redeclaration.)
- 4 • In the resulting list, remove any deferred feature such that the list contains an effective feature with the same final name. (This is the case in which a feature f , inherited as effective, effects one or more deferred features: of the whole group, only f remains.)
- 5 • All the features of the resulting list have different names; they are the inherited features of D in their parent forms. From this list, produce a new one by replacing any feature that D redeclares (through redefinition or effecting) with the result of the redeclaration, and retaining any other feature as it is.
- 6 • The result is the list inherited of inherited features of D .

Semantics: Join-Sharing Reconciliation rule 471

If a class inherits two or more features satisfying both the conditions of sharing under the Repeated Inheritance rule and those of the Join rule, the applicable semantics is the Repeated Inheritance rule.

Definition: Precursor 473

A **precursor** of an inherited feature of final name $fname$ is any parent feature — appearing in the list precursors obtained through case 1 of the definition of “Inherited features” — that the feature mergings resulting from the subsequent cases reduce into a feature of name $fname$.

Validity: Feature Name rule *VMFN* 474

It is valid for a feature f of a class C to have a certain final name if and only if it satisfies the following conditions:

- 1 • No other feature of C has that same feature name.
- 2 • If f is shared under repeated inheritance, its precursors all have either no Alias or the same alias.

Validity: Name Clash rule *VMNC* 475

The following properties govern the names of the features of a class C :

- 1 • It is invalid for C to introduce two different features with the same name.
- 2 • If C introduces a feature with the same name as a feature it inherits as effective, it must rename the inherited feature.
- 3 • If C inherits two features as effective from different parents and they have the same name, the class must also (except under sharing for repeated inheritance) remove the name clash through renaming.

FROM CHAPTER 17: CONTROL STRUCTURES

Semantics: Compound (non-exception) semantics 479

The effect of executing a Compound is:

- If it has zero instructions: to leave the state of the computation unchanged.
- If it has one or more instructions: to execute the first instruction of the Compound, then (recursively) to execute the Compound obtained by removing the first instruction.

This specification, the **non-exception semantics** of Compound, assumes that no exception is triggered. If the execution of any of the instructions triggers an exception, the Exception Semantics rule takes effect for the rest of the Compound’s instructions.

Syntax: Conditionals 481

Conditional \triangleq **if** Then_part_list [**Else_part**] **end**
Then_part_list \triangleq {Then_part **elseif** ...}⁺
Then_part \triangleq Boolean_expression **then** Compound

Else_part \triangleq else Compound

Definition: Secondary part 481

The **secondary part** of a **Conditional** possessing at least one **elseif** is the **Conditional** obtained by removing the initial “**if Then_part_list**” and replacing the first **elseif** of the remainder by **if**.

Definition: Prevailing immediately 481

The execution of a **Conditional** starting with **if condition_j** is said to **prevail immediately** if **condition_j** has value true.

Semantics: Conditional semantics 482

The effect of a **Conditional** is:

- If it **prevails immediately**: the effect of the first **Compound** in its **Then_part_list**.
- Otherwise, if it has at least one **elseif**: the effect (recursively) of its secondary part.
- Otherwise, if it has an **Else** part: the effect of the **Compound** in that **Else** part.
- Otherwise: no effect.

Definition: Inspect expression 484

The **inspect expression** of a **Multi_branch** is the expression appearing after the keyword **inspect**.

Syntax: Multi-branch instructions 485

Multi_branch \triangleq inspect Expression [When_part_list]

[Else_part] end

When_part_list \triangleq When_part⁺

When_part \triangleq when Choices then Compound

Choices \triangleq {Choice ", ...}⁺

Choice \triangleq Constant | Manifest_type | Constant_interval | Type_interval

Constant_interval \triangleq Constant " .. " Constant

Type_interval \triangleq Manifest_type " .. " Manifest_type

Definition: Interval 485

An **interval** is a **Constant_interval** or **Type_interval**.

Definition: Unfolded form of a multi-branch 486

To obtain the **unfolded form** of a **Multi_branch** instruction, apply the following transformations in the order given:

- 1 • Replace every **constant inspect value** by its **manifest value**.
- 2 • If the type **T** of the inspect expression is any **sized variant** of **CHARACTER**, **STRING** or **INTEGER**, replace every inspect value **v** by {**T**} **v**.
- 3 • Replace every **interval** by its **unfolded form**.

Definition: Unfolded form of an interval 486

The **unfolded form** of an **interval** **a..b** is the following (possibly empty) list:

- 1 • If **a** and **b** are constants, both of either a **character type**, a **string type** or an **integer type**, and of **manifest values** **va** and **vb**: the list made up of all values **i**, if

any, such that $va \leq i \leq vb$, using character, integer or lexicographical order respectively.

- 2 • If **a** and **b** are both of type **TYPE [T]** for some **T**, and have manifest values **va** and **vb**: the list containing every **Manifest_type** of the system conforming to **vb** and to which **va** conforms.
- 3 • If neither of the previous two cases apply: an empty list.

Validity: Interval rule **VOIN** 487

An **Interval** is valid if and only if its **unfolded form** is not empty.

Definition: Inspect values of a multi-branch 488

The **inspect values** of a **Multi_branch** instruction are all the values listed in the **Choices** parts of the instruction's **unfolded form**.

Validity: Multi-branch rule **VOMB** 488

A **Multi_branch** instruction is valid if and only if its unfolded form satisfies the following conditions.

- 1 • **Inspect values** are all valid.
- 2 • Inspect values are all **constants**.
- 3 • The **manifest values** of any two inspect values are different.
- 4 • If the **inspect expression** is of type **TYPE [T]** for some type **T**, all inspect values are types.
- 5 • If case 4 does not apply, the inspect expression is one of the sized variants of **INTEGER**, **CHARACTER** or **STRING**.

Semantics: Matching branch 489

During execution, a **matching branch** of a **Multi_branch** is a **When_part wp** of its **unfolded form**, satisfying either of the following for the value **val** of its **inspect expression**:

- 1 • $val \sim i$, where **i** is one of the non-**Manifest_type inspect values** listed in **wp**.
- 2 • **val** denotes a **Manifest_type** listed among the choices of **wp**.

Semantics: Multi-Branch semantics 490

Executing a **Multi_branch** with a **matching branch** consists of executing the **Compound** following the **then** in that branch. In the absence of matching branch:

- 1 • If the **Else_part** is present, the effect of the **Multi_branch** is that of the **Compound** appearing in its **Else_part**.
- 2 • Otherwise the execution **triggers** an **exception** of type **BAD_INSPECT_VALUE**.

Syntax: Loops

495

Loop \triangleq Initialization

[Invariant]

Exit_condition

Loop_body

[Variant]

end

Initialization \triangleq from CompoundExit_condition \triangleq until Boolean_expressionLoop_body \triangleq loop Compound**Semantics: Loop semantics**

496

The effect of a **Loop** is the effect of **executing** the **Compound** of its **Initialization**, then its **Loop_body**.

The effect of executing a **Loop_body** is:

- If the **Boolean_expression** of the **Exit_condition** evaluates to true: no effect (leave the state of the computation unchanged).
- Otherwise: the effect of **executing** the **Compound** clause, followed (recursively) by the effect of executing the **Loop_body** again in the resulting state.

Syntax: Debug instructions

498

Debug \triangleq debug [("Key_list")] Compound end**Semantics: Debug semantics**

498

A language processing tool must provide an option that makes it possible to enable or disable **Debug** instructions, both globally and for individual keys of a **Key_list**. Such an option may be settable for an entire system, or for individual classes, or both.

Letter case is not significant for a debug key.

The effect of a **Debug** instruction depends on the mode that has been set for the **current class**:

- If the **Debug** option is on generally, or if the instruction includes a **Key_list** and the option is on for at least one of the keys in the list, the effect of the **Debug** instruction is that of its **Compound**.
- Otherwise the effect is that of a null instruction.

FROM CHAPTER 18: ATTRIBUTES**Syntax: Attribute bodies**

501

Attribute \triangleq attribute Compound**Validity: Manifest Constant rule**

VQMC 503

A declaration of a feature f introducing a **manifest constant** is valid if and only if the **Manifest_constant** m used in the declaration matches the type T declared for f in one of the following ways:

- 1 • m is a **Boolean_constant** and T is **BOOLEAN**.
- 2 • m is a **Character_constant** and T is one of the **sized variants** of **CHARACTER** for which m is a valid value.
- 3 • m is an **Integer_constant** and T is one of the **sized variants** of **INTEGER** for which m is a valid value.

4 • m is a **Real_constant** and T is one of the **sized variants** of **REAL** for which m is a valid value.

5 • m is a **Manifest_string** and T is one of the **sized variants** of **STRING** for which m is a valid value.

6 • m is a **Manifest_type**, of the form $\{Y\}$ for some type Y , and T is **TYPE** $[X]$ for some **stand-alone type** X to which Y conforms.

FROM CHAPTER 19: OBJECTS, VALUES AND ENTITIES**Semantics: Type, generating type of an object; generator**

506

Every run-time object is a **direct instance** of exactly one **stand-alone type** of the system, called the **generating type** of the object, or just “the type of the object” if there is no ambiguity.

The **base class** of the generating type is called the object’s **generating class**, or **generator** for short.

Definition: Reference, void, attached, attached to

507

A **reference** is a value that is either:

- **Void**, in which case it provides no more information.
- **Attached**, in which case it gives access to an object. The reference is said to be **attached to** that object, and the object attached to the reference.

Semantics: Object principle

507

Every non-**void** value is either an object or a reference **attached** to an object.

Definition: Object semantics

508

Every run-time object has either **copy semantics** or **reference semantics**.

An object has copy semantics if and only if its **generating type** is an **expanded type**.

Definition: Non-basic class, non-basic type, field

508

Any class other than the **basic types** is said to be a **non-basic class**. Any type whose **base class** is non-basic is a **non-basic type**, and its instances are **non-basic objects**.

A **direct instance** of a non-basic type is a sequence of zero or more values, called **fields**. There is one field for every attribute of the type’s base class.

Definition: Subobject, composite object

509

Any **expanded field** of an object is a **subobject** of that object.

An object that has a **non-basic** subobject is said to be **composite**.

Definition: Entity, variable, read-only

512

An **entity** is an **Identifier**, or one of two reserved words (**Current** and **Result**), used in one of the following roles:

- 1 • **Final name** of an attribute of a class.

- 2 • Local variable of a routine or Inline_agent, including **Result** for a query.
- 3 • Formal argument of a routine or inline agent.
- 4 • Object Test local.
- 5 • **Current**, the predefined entity used to represent a reference to the current object (the target of the latest not yet completed routine call).

Names of non-constant attributes and local variables are **variable** entities, also called just **variables**. Constant attributes, formal arguments, Object Test locals and **Current** are **read-only** entities.

Syntax: Entities and variables 512

Entity \triangleq Variable | Read_only
 Variable \triangleq Variable_attribute | Local
 Variable_attribute \triangleq Feature_name
 Local \triangleq Identifier | **Result**
 Read_only \triangleq Formal | Constant_attribute | **Current**
 Formal \triangleq Identifier
 Constant_attribute \triangleq Feature_name

Validity: Entity rule *VEEN* 513

An occurrence of an entity *e* in the text of a class *C* (other than as the feature of a qualified call) is valid if and only if it satisfies one of the following conditions:

- 1 • *e* is **Current**.
- 2 • *e* is the final name of an attribute of *C*.
- 3 • *e* is the local variable **Result**, and the occurrence is in a Feature_body, Postcondition or Rescue part of an Attribute_or_routine text for a query or an Inline_agent whose signature includes a result type.
- 4 • *e* is **Result** appearing in the Postcondition of a constant attribute's declaration.
- 5 • *e* is listed in the Identifier_list of an Entity_declaration_group in a Local_declarations part of a feature or Inline_agent *fa*, and the occurrence is in a Local_declarations, Feature_body or Rescue part for *fa*.
- 6 • *e* is listed in the Identifier_list of an Entity_declaration_group in a Formal_arguments part for a routine *r*, and the occurrence is in a declaration for *r*.
- 7 • *e* is listed in the Identifier_list of an Entity_declaration_group in the Agent_arguments part of an Agent *a*, and the occurrence is in the Agent_body of *a*.
- 8 • *e* is the Object-Test Local of an Object_test, and the occurrence is in its scope.

Validity: Variable rule *VEVA* 514

A Variable entity *v* is valid in a class *C* if and only if it satisfies one of the following conditions:

- 1 • *v* is the final name of a variable attribute of *C*.

- 2 • *v* is the final name of a local variable of the immediately enclosing routine or agent.

Definition: Self-initializing type 515

A type is **self-initializing** if it is one of:

- 1 • A detachable type.
- 2 • A self-initializing formal parameter.
- 3 • An attached type (including expanded types and, as a special case of these, basic types) whose creation procedures include a version of default_create from ANY available for creation to *C*.

Semantics: Default Initialization rule 516

Every self-initializing type *T* has a **default initialization value** as follows:

- 1 • For a detachable type: a void reference.
- 2 • For a self-initializing attached type: an object obtained by creating an instance of *T* through default_create.
- 3 • For a self-initializing formal parameter: for every generic derivation, (recursively) the default initialization value of the corresponding actual generic parameter.
- 4 • For **BOOLEAN**: the boolean value false.
- 5 • For a sized variant of **CHARACTER**: null character.
- 6 • For a sized variant of **INTEGER**: integer zero.
- 7 • For a sized variant of **REAL**: floating-point zero.
- 8 • For **POINTER**: a null pointer.
- 9 • For **TYPED_POINTER**: an object representing a null pointer.

Definition: Self-initializing variable 517

A variable is **self-initializing** if one of the following holds:

- 1 • Its type is a self-initializing type.
- 2 • It is an attribute declared with an Attribute part such that the entity **Result** is properly set at the end of its Compound.

Definition: Evaluation position, precedes 517

An evaluation position is one of:

- In a Compound, one of its Instruction components.
- In an Assertion, one of its Assertion_clause components.
- In either case, a special **end position**.

A position *p* **precedes** a position *q* if they are both in the same Compound or Assertion, and either:

- *p* and *q* are both Instruction or Assertion_clause components, and *p* appears before *q* in the corresponding list.
- *q* is the end position and *p* is not.

Definition: Setter instruction 518

A **setter instruction** is an assignment or creation instruction.

If x is a **variable**, a setter instruction is a **setter for x** if its **assignment target** or **creation target** is x .

Definition: Properly set variable 518

At an **evaluation position** ep in a class C , a variable x is **properly set** if one of the following conditions holds:

- 1 • x is **self-initializing**.
- 2 • ep is an evaluation position of the **Compound** of a feature or **Inline_agent** of the **Internal** form, one of whose instructions **precedes** ep and is a **setter for x** .
- 3 • x is a variable attribute, and is (recursively) properly set at the **end position** of every **creation procedure** of C .
- 4 • ep is an evaluation position in a **Compound** that is part of an instruction ep' , itself belonging to a **Compound**, and x is (recursively) properly set at position ep' .
- 5 • ep is in a **Postcondition** of a routine or **Inline_agent** of the **Internal** form, and x is (recursively) properly set at the end position of its **Compound**.
- 6 • ep is an **Assertion_clause** containing **Result** in the **Postcondition** of a constant attribute

Validity: Variable Initialization rule *VEVI* 519

It is valid for an **Expression**, other than the target of an **Assigner_call**, to be also a **Variable** if it is **properly set** at the **evaluation position** defined by the closest enclosing **Instruction** or **Assertion_clause**.

Definition: Variable setting and its value 520

A **setting** for a variable x is any one of the following run-time events, defining in each case the **value** of the setting:

- 1 • Execution of a **setter for x** . (*Value*: the object **attached to x** by the setter, or a void reference if none.)
- 2 • If x is a **variable attribute** with an **Attribute** part: evaluation of that part, implying execution of its **Compound**. (*Value*: the object attached to **Result** at the **end position** of that **Compound**, or a void reference if none.)
- 3 • If the type T of x is **self-initializing**: assignment to x of T 's **default initialization value**. (*Value*: that initialization value.)

Definition: Execution context 521

At any time during execution, the current **execution context** for a variable is the period elapsed since:

- 1 • For an attribute: the creation of the **current object**.
- 2 • For a local variable: the start of execution of the **current routine**.

Semantics: Variable Semantics 521

The value produced by the run-time evaluation of a **variable x** is:

- 1 • If the **execution context** has previously executed at least one **setting for x** : the **value** of the latest such setting.
- 2 • Otherwise, if the type T of x is **self-initializing**: assignment to x of T 's **default initialization value**, causing a setting of x .
- 3 • Otherwise, if x is a **variable attribute** with an **Attribute** part: evaluation of that part, implying execution of its **Compound** and hence a setting for x .
- 4 • Otherwise, if x is **Result** in the **Postcondition** of a **constant attribute**: the **value** of the attribute.

Semantics: Entity Semantics rule 522

Evaluating an **entity** yields a **value** as follows:

- 1 • For **Current**: a value **attached to** the **current object**.
- 2 • For a formal argument of a routine or **Inline_agent**: the value of the corresponding actual at the time of the **current call**.
- 3 • For a **constant attribute**: the value of the associated **Manifest_constant** as determined by the Manifest Constant Semantics rule.
- 4 • For an **Object-Test Local**: as determined by the Object-Test Local Semantics rule.
- 5 • For a **variable**: as determined by the Variable Semantics rule.

FROM CHAPTER 20: CREATING OBJECTS**Semantics: Creation principle** 523

Any execution of a creation operation must produce an object that satisfies the invariant of its **generating class**.

Definition: Creation operation 524

A **creation operation** is a creation instruction or expression.

Validity: Creation Precondition rule *VGCP* 547

A **Precondition** of a routine r is **creation-valid** if and only if its **unfolded form** uf satisfies the following conditions:

- 1 • The predefined entity **Current** does not appear in uf .
- 2 • No **Unqualified_call** appears in uf .
- 3 • Every feature whose final name appears in the uf is available to every class to which r is **available for creation**.

Syntax: Creators parts 547

Creators \triangleq **Creation_clause**⁺
Creation_clause \triangleq **create** [**Clients**] [**Header_comment**]
Creation_procedure_list

$\text{Creation_procedure_list} \triangleq \{\text{Creation_procedure } ", \dots \}^+$
 $\text{Creation_procedure} \triangleq \text{Feature_name}$

Definition: Unfolded Creators part of a class 548

The **unfolded creators part** of a class C is a **Creators** defined as:

- 1 • If C has a **Creators** part c : c .
- 2 • If C is **deferred**: an empty **Creators** part.
- 3 • Otherwise, a **Creators** part built as follows, dc_name being the **final name** in C of its **version** of dc_name from ANY :

create
 dc_name

Validity: Creation Clause rule VGCC 548

A **Creation_clause** in the **unfolded creators part** of a class C is valid if and only if it satisfies the following conditions, the last four for every **Feature_name** cp_name in the clause's **Feature_list**:

- 1 • C is **effective**.
- 2 • cp_name appears only once in the **Feature_list**.
- 3 • cp_name is the final name of some procedure cp of C .
- 4 • cp is not a **once routine**.
- 5 • The precondition of cp , if any, is **creation-valid**.

Definition: Creation procedures of a class 550

The **creation procedures** of a class are all the features appearing in any **Creation_clause** of its **unfolded creators part**.

Creation procedure property 550

An **effective class** has at least one **creation procedure**.

Definition: Creation procedures of a type 550

The **creation procedures** of a type T are:

- 1 • If T is a **Formal_generic_name**, the **constraining creators for** T .
- 2 • Otherwise, the **creation procedures** of T 's **base class**.

Definition: Available for creation; general creation procedure 551

A creation procedure of a class C , listed in a **Creation_clause** cc of C 's **unfolded creators part**, is **available for creation** to the **descendants** of the classes given in the **Clients** restriction of cc , if present, and otherwise to all classes.

If there is no **Clients** restriction, the procedure is said to be a **general creation procedure**.

Syntax: Creation instructions 551

$\text{Creation_instruction} \triangleq \text{create } [\text{Explicit_creation_type}]$
 Creation_call
 $\text{Explicit_creation_type} \triangleq \{ \text{" Type " } \}$
 $\text{Creation_call} \triangleq \text{Variable } [\text{Explicit_creation_call}]$
 $\text{Explicit_creation_call} \triangleq \text{" . " Unqualified_call}$

Definition: Creation target, creation type 551

The **creation target** (or just "target" if there is no ambiguity) of a **Creation_instruction** is the **Variable** of its **Creation_call**.

The **creation type** of a creation instruction, denoting the type of the object to be created, is:

- The **Explicit_creation_type** appearing (between braces) in the instruction, if present.
- Otherwise, the type of the instruction's **target**.

Semantics: Creation Type theorem 552

The **creation type** of a creation instruction is always **effective**.

Definition: Unfolded form of a creation instruction 552

Consider a **Creation_instruction** ci of creation type CT . The **unfolded form** of ci is a creation instruction defined as:

- 1 • If ci has an **Explicit_creation_call**, then ci itself.
- 2 • Otherwise, a **Creation_instruction** obtained from ci by making the **Creation_call** explicit, using as **feature name** the **final name** in CT of CT 's **version** of ANY 's **default_create**.

Validity: Creation Instruction rule VGCI 553

A **Creation_instruction** of creation type CT , appearing in a class C , is valid if and only if it satisfies the following conditions:

- 1 • CT **conforms to** the **target's** type.
- 2 • The feature of the **Creation_call** of the instruction's **unfolded form** is **available for creation to** C .
- 3 • That **Creation_call** is **argument-valid**.
- 4 • CT is **generic-creation-ready**.

Validity: Creation Instruction properties VGCP 555

A **Creation_instruction** ci of creation type CT , appearing in a class C , is valid only if it satisfies the following conditions, assuming CT is not a **Formal_generic_name** and calling BCT the **base class** of CT and dc the **version** of ANY 's **default_create** in BCT :

- 1 • BCT is an **effective class**.
- 2 • If ci includes a **Type** part, the type it lists (which is CT) conforms to the type of the instruction's **target**.
- 3 • If ci has no **Creation_call**, then BCT either has no **Creators** part or has one that lists dc as one of the procedures **available to** C for creation.
- 4 • If BCT has a **Creators** part which doesn't list dc , then ci has a **Creation_call**.
- 5 • If ci has a **Creation_call** whose feature f is not dc , then BCT has a **Creators** part which lists f as one of the procedures **available to** C for creation.
- 6 • If ci has a **Creation_call**, that call is **argument-valid**.

If *CT* is a **Formal_generic_name**, the instruction is valid only if it satisfies the following conditions:

555

- 7 • *CT* denotes a **constrained generic parameter**.
- 8 • The **Constraint** for *CT* specifies one or more **procedures** as **constraining creators**.
- 9 • If *ci* has no **Creation_call**, one of the constraining creators is the **Constraint**'s version of **default_create** from *ANY*.
- 10 • If *ci* has a **Creation_call**, one of the constraining creators is the **feature** of the **Creation_call**.

Semantics: Creation Instruction Semantics 556

The effect of a creation instruction of **target** *x* and **creation type** *TC* is the effect of the following sequence of steps, in order:

- 1 • If there is **not enough memory available** for a new direct instance of *TC*, **trigger** an **exception of type** *NO_MORE_MEMORY* in the routine that attempted to execute the instruction. The remaining steps do not apply in this case.
- 2 • Create a new **direct instance** of *TC*, with **reference semantics** if *CT* is a **reference type** and **copy semantics** if *CT* is an **expanded type**.
- 3 • Call, on the resulting object, the **feature** of the **Unqualified_call** of the instruction's **unfolded form**.
- 4 • **Attach** *x* to the object.

Syntax: Creation expressions 561

Creation_expression $\hat{=}$ **create** **Explicit_creation_type** [**Explicit_creation_call**]

Definition: Properties of a creation expression 561

The **creation type** and **unfolded form** of a creation expression are defined as for a creation instruction.

Validity: Creation Expression rule *VGCE* 562

A **Creation_expression** of creation type *CT*, appearing in a class *C*, is valid if and only if it satisfies the following conditions:

- 1 • The **feature** of the **Creation_call** of the expression's **unfolded form** is **available for creation** to *C*.
- 2 • That **Creation_call** is argument-valid.
- 3 • *CT* is generic-creation-ready.

Validity: Creation Expression Properties *VGCX* 562

A **Creation_expression** *ce* of creation type *CT*, appearing in a class *C*, is valid only if it satisfies the following conditions, assuming *CT* is not a **Formal_generic_name** and calling *BCT* the **base class** of *CT* and *dc* the version of *ANY*'s **default_create** in *BCT*:

- 1 • *BCT* is an **effective class**.

- 2 • If *ce* has no **Explicit_creation_call**, then *BCT* either has no **Creators** part or has one that lists *dc* as one of the **procedures** **available** to *C* **for creation**.
- 3 • If *BCT* has a **Creators** part which doesn't list *dc*, then *ce* has an **Explicit_creation_call**.
- 4 • If *ce* has an **Explicit_creation_call** whose **feature** *f* is not *dc*, then *BCT* has a **Creators** part which lists *f* as one of the **procedures** **available** to *C* **for creation**.
- 5 • If *ce* has an **Explicit_creation_call**, that call is **argument-valid**.

If *CT* is a **Formal_generic_name**, the expression is valid only if it satisfies the following conditions:

562

- 6 • *CT* denotes a **constrained generic parameter**.
- 7 • The **Constraint** for *CT* specifies one or more **procedures** as **constraining creators**.
- 8 • If *ce* has no **Creation_call**, one of the constraining creators is the **Constraint**'s version of **default_create** from *ANY*.
- 9 • If *ce* has a **Creation_call**, one of the constraining creators is the **feature** of the **Creation_call**.

Semantics: Creation Expression Semantics 563

The value of a creation expression of creation type *TC* is — except if step 1 below **triggers** an **exception**, in which case the expression has no value — a value **attached** to a new object as can be obtained through the following sequence of steps:

- 1 • If there is **not enough memory available** for a new direct instance of *TC*, **trigger** an **exception of type** *NO_MORE_MEMORY* in the routine that attempted to execute the expression. In this case the expression has no value and the remaining steps do not apply.
- 2 • Create a new **direct instance** of *TC*, with **reference semantics** if *CT* is a **reference type** and **copy semantics** if *CT* is an **expanded type**.
- 3 • Call, on the resulting object, the **feature** of the **Unqualified_call** of the expression's **unfolded form**.

Definition: Garbage Collection, not enough memory available 564

Authors of Eiffel implementation are required to provide **garbage collection**, defined as a mechanism that can reuse for allocating new objects the memory occupied by unreachable objects, guaranteeing the following two properties:

- 1 • *Consistency*: the garbage collector never reclaims an object unless it is unreachable.
- 2 • *Completeness*: no allocation request for an object of a certain size *s* will fail if there exists an unreachable object of size $\geq s$.

Not enough memory available for a certain size s means that even after possible application of the garbage collection mechanism the memory available to the program is not sufficient for allocating an object of size s .

FROM CHAPTER 21: COMPARING AND DUPLICATING OBJECTS

Object comparison features from *ANY* 566

The features whose contract views appear below are provided by class *ANY*.

default_is_equal (*other*: **like** *Current*)

-- Is *other* attached to object field-by-field equal

-- to current object?

ensure

same_type: **Result implies** *same_type*

(*other*)

symmetric: **Result** =
other.default_is_equal (*Current*)

(*other*)

consistent: **Result implies** *is_equal*

(*other*)

is_equal (*other*: **? like** *Current*)

-- Is *other* attached to object considered equal

-- to current object?

ensure

same_type: **Result implies** *same_type*

(*other*)

symmetric: **Result** = *other.is_equal*

(*Current*)

consistent: *default_is_equal* (*other*)

implies Result

The original version of *is_equal* in *ANY* has the same effect as *default_is_equal*.

Syntax: Equality expressions 567

Equality \triangleq Expression Comparison Expression

Comparison \triangleq "=" | "/" | "~" | "/~"

Semantics: Equality Expression Semantics 567

The Boolean expression $e \sim f$ has value true if and only if the values of e and f are both attached and such that $e.is_equal(f)$ holds.

The Boolean expression $e = f$ has value true if and only if the values of e and f are one of:

- 1 • Both void.
- 2 • Both attached to the same object with reference semantics.
- 3 • Both attached to objects with copy semantics, and such that $e \sim f$ holds.

Semantics: Inequality Expression Semantics 568

The expression $e \neq f$ has value true if and only if $e = f$ has value false.

The expression $e \sim\sim f$ has value true if and only if $e \sim f$ has value false.

Copying and cloning features from *ANY* 569

The features whose contract views appear below are provided by class *ANY* as secret features.

copy (*other*: **? like** *Current*)

-- Update current object using fields of object

-- attached to *other*, to yield equal objects.

require

exists: *other* \neq Void

same_type: *other.same_type* (*Current*)

ensure

equal: *is_equal* (*other*)

frozen *default_copy* (*other*: **? like** *Current*)

-- Update current object using fields of object

-- attached to *other*, to yield identical objects.

require

exists: *other* \neq Void

same_type: *other.same_type* (*Current*)

ensure

equal: *default_is_equal* (*other*)

frozen *cloned*: **like** *Current*

-- New object equal to current object

-- (relies on *copy*)

ensure

equal: *is_equal* (*Result*)

frozen *default_cloned*: **like** *Current*

-- New object equal to current object

-- (relies on *default_copy*)

ensure

equal: *default_is_equal* (*Result*)

The original versions of *copy* and *cloned* in *ANY* have the same effect as *default_copy* and *default_cloned* respectively.

Deep equality, copying and cloning 570

The feature *is_deep_equal* of class *ANY* makes it possible to compare object structures recursively; the features *deep_copy* and *deep_cloned* duplicate an object structure recursively.

FROM CHAPTER 22: ATTACHING VALUES TO ENTITIES

Definition: Reattachment, source, target 588

A **reattachment** operation is one of:

- 1 • An **Assignment** $x := y$; then y is the attachment's source and x its target.
- 2 • The run-time association, during the execution of a routine call, of an actual argument (the source) to the corresponding formal argument (the target).

Syntax: Assignments 589

Assignment \triangleq Variable $:=$ Expression

Validity: Assignment rule VBAR 590

An **Assignment** is valid if and only if its source expression is compatible with its target entity.

Semantics: Reattachment principle 599

After a reattachment to a target entity t of type TT , the object attached to t , if any, is of a type conforming to TT .

Semantics: Attaching an entity, attached entity 600

Attaching an entity e to an object O is the operation ensuring that the value of e becomes **attached to** O .

Semantics: Reattachment Semantics 600

The effect of a reattachment of source expression $source$ and target entity $target$ is the effect of the first of the following steps whose condition applies:

- 1 • If $source$ converts to $target$: perform a conversion attachment from $source$ to $target$.
- 2 • If the value of $source$ is a void reference: make $target$'s value void as well.
- 3 • If the value of $source$ is attached to an object with copy semantics: create a clone of that object, if possible, and attach $target$ to it.
- 4 • If the value of $source$ is attached to an object with reference semantics: attach $target$ to that object.

Semantics: Assignment Semantics 601

The effect of a reassignment $x := y$ is determined by the Reattachment Semantics rule, with source y and target x .

Definition: Dynamic type 606

The **dynamic type** of an expression x , at some instant of execution, is the type of the object to which x is attached, or **NONE** if x is void.

Definition: Polymorphic expression; dynamic type and class sets 606

An expression that has two or more possible dynamic types is said to be **polymorphic**.

The set of possible dynamic types for an expression x is called the **dynamic type set** of x . The set of base classes of these types is called the **dynamic class set** of x .

Syntax: Assigner calls 609

Assigner_call \triangleq Expression $:=$ Expression

Validity: Assigner Call rule VBAC 610

An **Assigner_call** of the form $target := source$, where $target$ and $source$ are expressions, is valid if and only if it satisfies the following conditions:

- 1 • $source$ is compatible with $target$.
- 2 • The Equivalent Dot Form of $target$ is a qualified **Object_call** whose feature has an assigner command.

Semantics: Assigner Call semantics 610

The effect of an **Assigner_call** $target := source$, where the Equivalent Dot Form of $target$ is $x.f$ or $x.f(args)$ and f has an assigner command p , is, respectively, $x.p(source)$ or $x.p(source, args)$.

FROM CHAPTER 23: FEATURE CALL

Validity: Call Use rule VUCN 623

A **Call** of feature f denotes:

- 1 • If f is a query (attribute or a function): an expression.
- 2 • If f is a procedure: an instruction.

Syntax: Feature calls 626

Call \triangleq Object_call | Non_object_call
Object_call \triangleq [Target "."] Unqualified_call
Unqualified_call \triangleq Feature_name [Actuals]
Target \triangleq Local | Read_only | Call | Parenthesized_target
Parenthesized_target \triangleq "(" Expression ")"
Non_object_call \triangleq "{" Type "}" "." Unqualified_call

Syntax: Actual arguments 626

Actuals \triangleq "(" Actual_list ")"
Actual_list \triangleq {Expression "," ... }⁺

Definition: Unqualified, qualified call 627

An **Object_call** is **qualified** if it has a **Target**, **unqualified** otherwise.

Definition: Target of a call 628

Any **Object_call** has a **target**, defined as follows:

- 1 • If it is qualified: its **Target component**.
- 2 • If it is unqualified: **Current**.

Definition: Target type of a call 629

Any **Call** has a **target type**, defined as follows:

- 1 • For an **Object_call**: the type of its target. (In the case of an **Unqualified_call** this is the current type.)
- 2 • For a **Non_object_call** having a type T as its **Type** part: T .

Definition: Feature of a call 629

For any **Call** the "**feature of the call**" is defined as follows:

- 1 • For an **Unqualified_call**: its **Feature_name**.

- 2 • For a qualified call or Non_object_call: (recursively) the feature of its Unqualified_call part.

Definition: Imported form of a Non_object_call 630

The **imported form** of a Non_object_call of Type T and feature *f* appearing in a class *C* is the Unqualified_call built from the original Actuals if any and, as feature of the call, a fictitious new feature added to *C* and consisting of the following elements:

- 1 • A name different from those of other features of *C*.
- 2 • A Declaration_body obtained from the Declaration_body of *f* by replacing every type by its deanchored form, then applying the generic substitution of *T*.

Validity: Non-Object Call rule *VUNO* 631

A Non_object_call of Type T and feature *fname* in a class *C* is valid if and only if it satisfies the following conditions:

- 1 • *fname* is the final name of a feature *f* of *T*.
- 2 • *f* is available to *C*.
- 3 • *f* is either a constant attribute or an external feature whose assertions, if any, use neither **Current** nor any unqualified calls.
- 4 • The call's imported form is a valid Unqualified_call.

Semantics: Non-Object Call Semantics 631

The effect of a Non_object_call is that of its imported form.

Validity: Export rule *VUEX* 632

An Object_call appearing in a class *C*, with *fname* as the feature of the call, is **export-valid** for *C* if and only if it satisfies the following conditions.

- 1 • *fname* is the final name of a feature of the target type of the call.
- 2 • If the call is qualified, that feature is available to C. The export status of a feature *f*:
 - Constrains all qualified calls *x.f (...)*, including those in which the type of *x* is the current type, or is **Current** itself.
 - Does not constrain unqualified calls.

Validity: Argument rule *VUAR* 634

An export-valid call of target type *ST* and feature *fname* appearing in a class *C* where it denotes a feature *sf* is **argument-valid** if and only if it satisfies the following conditions:

- 1 • The number of actual arguments is the same as the number of formal arguments declared for *sf*.
- 2 • Every actual argument of the call is compatible with the corresponding formal argument of *sf*.

Validity: Target rule *VUTA* 635

An Object_call is **target-valid** if and only if either:

- 1 • It is unqualified.
- 2 • Its target is an attached expression.

Validity: Class-Level Call rule *VUCC* 636

A call of target type *ST* is **class-valid** if and only if it is export-valid, argument-valid and target-valid.

Definition: Void-Unsafe 636

A language processing tool may, as a temporary migration facility, provide an option that waives the target validity requirement in class validity. Systems processed under such an option are **void-unsafe**.

Definition: Target Object 637

The **target object** of an execution of an Object_call is:

- 1 • If the call is qualified: the object attached to its target.
- 2 • If it is unqualified: the current object.

Semantics: Failed target evaluation of a void-unsafe system 638

In the execution of an (invalid) system compiled in void-unsafe mode through a language processing tool offering such a migration option, an attempt to execute a call triggers, if it evaluates the target to a void reference, an exception of type VOID_TARGET.

Definition: Dynamic feature of a call 639

Consider an execution of a call of feature *fname* and target object O. Let *ST* be its target type and *DT* the type of *O*. The **dynamic feature** of the call is the dynamic binding version in *DT* of the feature of name *fname* in *ST*.

Definition: Freshness of a once routine call 644

During execution, a call whose feature is a once routine *r* is **fresh** if and only if every feature call started so far satisfies any of the following conditions:

- 1 • It did not use *r* as dynamic feature.
- 2 • It was in a different thread, and *r* has the once key "**THREAD**" or no once key.
- 3 • Its target was not the current object, and *r* has the once key "**OBJECT**".
- 4 • After it was started, a call was executed to one of the refreshing features of onces from *ANY*, including among the keys to be refreshed at least one of the once keys of *r*.

Definition: Latest applicable target and result of a non-fresh call 645

The **latest applicable target** of a non-fresh call to a once routine *df* to a target object O is the last value to which it was attached in the call to *df* most recently started on:

- 1 • If *df* has the once key "**OBJECT**": *O*.

- 2 • Otherwise, if *df* has the once key "*THREAD*" or no once key: any target in the current thread.
- 3 • Otherwise: any target in any thread.

If *df* is a function, the **latest applicable result** of the call is the last value returned by a fresh call using as target object its latest applicable target.

Semantics: Once Routine Execution Semantics 646

The effect of executing a once routine *df* on a target object *O* is:

- 1 • If the call is fresh: that of a non-once call made of the same elements, as determined by Non-once Routine Execution Semantics.
- 2 • If the call is not fresh and the last execution of *f* on the latest applicable target triggered an exception: to trigger again an identical exception. The remaining cases do not then apply.
- 3 • If the call is not fresh and *df* is a procedure: no further effect.
- 4 • If the call is not fresh and *df* is a function: to attach the local variable **Result** to the latest applicable result of the call.

Semantics: Current object, current routine 649

At any time during the execution of a system there is a **current object** *CO* and a **current routine** *cr* defined as follows:

- 1 • At the start of the execution: *CO* is the root object and *cr* is the root procedure.
- 2 • If *cr* executes a qualified call: the call's target object becomes the new current object, and its dynamic feature becomes the new current routine. When the qualified call terminates, the earlier current object and routine resume their roles.
- 3 • If *cr* executes an unqualified call: the current object remains the same, and the dynamic feature of the call becomes the current routine for the duration of the call as in case 2.
- 4 • If *cr* starts executing any construct whose semantics does not involve a call: the current object and current routine remain the same.

Semantics: Current Semantics 651

The value of the predefined entity **Current** at any time during execution is the current object if the current routine belongs to an expanded class, and a reference to the current object otherwise.

Semantics: Non-Once Routine Execution Semantics

652

The effect of executing a non-once routine *df* on a target object *O* is the effect of the following sequence of steps:

- 1 • If *df* has any local variables, including **Result** if *df* is a function, save their current values if any call to *df* has been started but not yet terminated.
- 2 • Execute the body of *df*.
- 3 • If the values of any local variables have been saved in step 1, restore the variables to their earlier values.

Semantics: General Call Semantics 653

The effect of an Object call of feature *sf* is, in the absence of any exception, the effect of the following sequence of steps:

- 1 • Determine the target object *O* through the applicable definition.
- 2 • Attach **Current** to *O*.
- 3 • Determine the dynamic feature *df* of the call through the applicable definition.
- 4 • For every actual argument *a*, if any, in the order listed: obtain the value *v* of *a*; then if the type of *a* converts to the type of the corresponding formal in *sf*, replace *v* by the result of the applicable conversion. Let *arg_values* be the resulting sequence of all such *v*.
- 5 • Attach every formal argument of *df* to the corresponding element of *arg_values* by applying the Reattachment Semantics rule.
- 6 • If the call is qualified and class invariant monitoring is on, evaluate the class invariant of *O*'s base type on *O*.
- 7 • If precondition monitoring is on, evaluate the precondition of *df*.
- 8 • If *df* is not an attribute, not a once routine and not external, apply Non-Once Routine Execution Semantics to *O* and *df*.
- 9 • If *df* is a once routine, apply the Once Routine Execution Semantics to *O* and *df*.
- 10 • If *df* is an external routine, execute that routine on the actual arguments given, if any, according to the rules of the language in which it is written.
- 11 • If *df* is a self-initializing attribute and has not yet been initialized, initialize it through the Default Initialization rule.
- 12 • If the call is qualified and class invariant monitoring is on, evaluate the class invariant of *O*'s base type on *O*.
- 13 • If postcondition monitoring is on, evaluate the postcondition of *df*.

An exception occurring during any of these steps causes the execution to skip the remaining parts of this process and instead handle the exception according to the Exception Semantics rule.

Definition: Type of a Call used as expression 655

Consider a call denoting an expression. Its **type** with respect to a type *CT* of base class *C* is:

- 1 • For an unqualified call, its feature *f* being a query of *CT*: the result type of the version of *f* in *C*, adapted through the generic substitution of *CT*.
- 2 • For a qualified call *a.e* of Target *a*: (recursively) the type of *e* with respect to the type of *a*.
- 3 • For a Non_object_call: (recursively) the type of its imported form.

Semantics: Call Result 656

Consider a Call *c* whose feature is a query. An execution of *c* according to the General Call Semantics yields a **call result** defined as follows, where *O* is the target object determined at step 1 of the rule and *df* the dynamic feature determined at step 3:

- 1 • If *df* is a non-external, non-once function: the value attached to the local variable Result of *df* at the end of step 2 of Non-Once Routine Execution Semantics.
- 2 • If *df* is a once function: the value attached to Result as a result of the application of Once Routine Execution Semantics.
- 3 • If *df* is an attribute: the corresponding field in *O*.
- 4 • If *df* is an external function: the result returned by the function according to the external language's rule.

Semantics: Value of a call expression 656

The **value** of a Call *c* used as an expression is, at any run-time moment, the result of executing *c*.

FROM CHAPTER 24: ERADICATING VOID CALLS**Syntax: Object test** 658

$\text{Object_test} \triangleq \text{"\{ Identifier \}:" Type \}" \text{Expression}$

Definition: Object-Test Local 659

The **Object-Test Local** of an Object_test is its Identifier component.

Validity: Object Test rule *VUOT* 659

An Object_test *ot* of the form $\{x: T\} \text{exp}$ is valid if and only if it satisfies the following conditions:

- 1 • *x* does not have the same lower name as any feature of the enclosing class, or any formal argument or local variable of any enclosing feature or Inline_agent, or, if *ot* appears in the scope of any other Object_test, its Object-Test Local.
- 2 • *T* is an attached type.

Definition: Conjunctive, disjunctive, implicative; Term, semistrict term 660

Consider an Operator_expression *e* of boolean type, which after resolution of any ambiguities through precedence rules can be expressed as $a_1 \ \& \ a_2 \ \&\dots \ \& \ a_n$

for $n \geq 1$, where $\&$ represents boolean operators and every a_i , called a **term**, is itself a valid Boolean_expression. Then *e* is:

- **Conjunctive** if every $\&$ is either **and** or **and then**.
- **Disjunctive** if every $\&$ is either **or** or **or else**.
- **Implicative** if $n = 2$ and $\&$ is **implies**.

A term a_i is **semistrict** if in the corresponding form it is followed by a semistrict operator.

Definition: Scope of an Object-Test Local 661

The scope of the Object-Test Local of an Object_test *ot* includes any applicable program element from the following:

- 1 • If *ot* is a semistrict term of a conjunctive expression: any subsequent terms.
- 2 • If *ot* is a term of an implicative expression: the next term.
- 3 • If **not** *ot* is a semistrict term of a disjunctive expression *e*: any subsequent terms.
- 4 • If *ot* is a term of a conjunctive expression serving as the Boolean_expression in the Then_part in a Conditional: the corresponding Compound.
- 5 • If **not** *ot* is a term of a disjunctive expression serving as the Boolean_expression in the Then_part in a Conditional: any subsequent Then_part and Else_clause.
- 6 • If **not** *ot* is a term of a disjunctive expression serving as the Exit_condition in a Loop: the Loop_body.
- 7 • If *ot* is a term of a conjunctive expression used as Unlabeled_assertion_clause in a Precondition: the subsequent components of the Attribute_or_routine.
- 8 • If *ot* is a term of a conjunctive expression used as Unlabeled_assertion_clause in a Check: the subsequent components of its enclosing Compound.

Semantics: Object Test semantics 661

The value of an Object_test $\{x: T\} \text{exp}$ is true if the value of *exp* is attached to an instance of *T*, false otherwise.

Semantics: Object-Test Local semantics 662

For an Object_test $\{x: T\} \text{exp}$, the value of *x*, defined only over its scope, is the value of *exp* at the time of the Object_test's evaluation.

Definition: Read-only void test 662

A **read-only void test** is a Boolean_expression of one of the forms $e = \text{Void}$ and $e / = \text{Void}$, where *e* is a read-only entity.

Definition: Scope of a read-only void test 662

The **scope** of a read-only void test appearing in a class text, for *e* of type *T*, is the scope that the Object-Test Local *ot* would have if the void test were replaced by:

- 1 • For $e = \text{Void}$: **not** ($\{ot: T\} e$).
- 2 • For $e \neq \text{Void}$: $\{ot: T\} e$.

Definition: Certified Attachment Pattern 663

A **Certified Attachment Pattern** (or **CAP**) for an expression exp whose type is detachable is an occurrence of exp in one of the following contexts:

- 1 • exp is an Object-Test Local and the occurrence is in its scope.
- 2 • exp is a read-only entity and the occurrence is in the scope of a void test involving exp .

Definition: Attached expression 664

An expression exp of type T is **attached** if it satisfies any of the following conditions:

- 1 • T is attached.
- 2 • T is expanded.
- 3 • exp appears in a Certified Attachment Pattern for exp .

FROM CHAPTER 25: TYPING-RELATED PROPERTIES**Definition: Catcall** 665

A **catcall** is a run-time attempt to execute a **Call**, such that the feature of the call is not applicable to the target of the call.

Validity: Descendant Argument rule VUDA 667

Consider a call of target type ST and feature $fname$ appearing in a class C . Let sf be the feature of final name $fname$ in ST . Let DT be a type conforming to ST , and df the version of sf in DT . The call is **descendant-argument-valid** for DT if and only if it satisfies the following conditions:

- 1 • The call is argument-valid.
- 2 • Every actual argument conforms, after conversion to the corresponding formal argument of sf if applicable, to the corresponding formal argument of df .

Validity: Single-level Call rule VUSC 668

A call of target x is **system-valid** if for any element D of the dynamic class set of x it is export-valid for D and descendant-argument-valid for D .

FROM CHAPTER 26: EXCEPTION HANDLING**Definition: Failure, exception, trigger** 690

Under certain circumstances, the execution or evaluation of a construct specimen may be unable to proceed as defined by the construct's semantics. It is then said to result in a **failure**.

If, during the execution of a feature, the execution of one of its components fails, this prevents continuing its

execution normally; such an event is said to **trigger** an **exception**.

Syntax: Rescue clauses 701

$\text{Rescue} \triangleq \text{rescue Compound}$
 $\text{Retry} \triangleq \text{retry}$

Validity: Rescue clause rule VXRC 701

It is valid for an Attribute_or_routine to include a **Rescue** clause if and only if its Feature_body is an Attribute or an Effective_routine of the Internal form.

Validity: Retry rule VXRT 701

A **Retry** instruction is valid if and only if it appears in a **Rescue** clause.

Definition: Exception-correct 702

A routine is **exception-correct** if any branch of the **Rescue** clause not terminating with a **Retry** ensures the invariant.

Semantics: Default Rescue Original Semantics 702

Class **ANY** introduces a non-frozen procedure default_rescue with no argument and a null effect.

Definition: Rescue block 703

Any Internal or Attribute feature f of a class C has a **rescue block**, a **Compound** defined as follows, where rc is C 's version of **ANY**'s default_rescue:

- 1 • If f has a **Rescue** clause: the **Compound** contained in that clause.
- 2 • If r is not rc and has no **Rescue** clause: a **Compound** made of a single instruction: an **Unqualified_call** to rc .
- 3 • If r is rc and has no **Rescue** clause: an empty **Compound**.

Semantics: Exception Semantics 704

An exception triggered during an execution of a feature f causes, if it is neither ignored nor continued, the effect of the following sequence of events.

- 1 • Attach the value of last_exception from **ANY** to a direct instance of a descendant of the Kernel Library class **EXCEPTION** corresponding to the type of the exception.
- 2 • Unlike in the non-exception semantics of **Compound**, do not execute the remaining instructions of f .
- 3 • If the recipient of the exception is f , execute the rescue block of f .
- 4 • If case 3 applies and the rescue block executes a **Retry**, this terminates the processing of the exception. Execution continues with a new execution of the **Compound** in the Feature_body of f .
- 5 • If neither case 3 nor case 4 applies (in particular in case 3 if the rescue block executes to the end without executing a **Retry**), this terminates the processing of

the current exception and the current execution of f , causing a failure of that execution. If the execution of f was caused by a call to f from another feature, trigger an exception of type *ROUTINE_FAILURE* in the calling routine, to be handled (recursively) according to the present rule. If there is no such calling feature, f is the root procedure; terminate its execution as having failed.

Definition: Type of an exception 705

The **type** of a triggered exception is the generating type of the object to which the value of *last_exception* is attached per step 1 of the Expression Semantics rule.

Semantics: Exception Cases 706

The triggering of an exception in a feature f called by a feature *caller* results in the setting of the following properties, accessible through features of the exception class instance to which the value of *last_exception* is attached, as per the following table, where:

- The **Recipient** is either f or *caller*.
- “**Type**” indicates the type of the exception (a descendant of *EXCEPTION*).
- If f is the root procedure, executed during the original system creation call, the value of *caller* as given below does not apply.

Type	Recipient
Exception during evaluation [Type of exception as triggered] of invariant on entry	<i>caller</i>
Invariant violation on entry <i>INVARIANT_ENTRY_VIOLATION</i>	<i>caller</i>
Exception during evaluation [Type of exception as triggered] of precondition	<i>caller</i>
Exception during evaluation of <u>Old expression</u> on entry of <u>Old expression</u> on entry	See <u>Old Expression Semantics</u>
Precondition violation <i>PRECONDITION_VIOLATION</i>	<i>caller</i>
Exception in body [Type of exception as triggered]	f
Exception during evaluation [Type of exception as triggered] of invariant on exit	f
Invariant violation on exit <i>INVARIANT_EXIT_VIOLATION</i>	f
Exception during evaluation [Type of exception as triggered] of postcondition on exit	f
Postcondition violation <i>POSTCONDITION_VIOLATION</i>	f

Semantics: Exception Properties 707

The value of the query *original* of class *EXCEPTION*, applicable to *last_exception*, is an *EXCEPTION* reference determined as follows after the triggering of an exception of type *TEX*:

- 1 • If *TEX* does not conform to *ROUTINE_FAILURE*: a reference to the current *EXCEPTION* object.
- 2 • If *TEX* conforms to *ROUTINE_FAILURE*: the previous value of *original*.

Definition: Ignoring, continuing an exception 709

It is possible, through routines of the Kernel Library class *EXCEPTION*, to ensure that exceptions of certain types be:

- **Ignored**: lead to no change of non-exception semantics.
- **Continued**: lead to execution of a programmer-specified routine, then to continuation of the execution according to non-exception semantics.

FROM CHAPTER 27: AGENTS, ITERATION AND INTROSPECTION

Definition: Operands of a call 723

The **operands** of a call include its target (explicit in a qualified call, implicit in an unqualified call), and its arguments if any.

Definition: Operand position 723

The target of a call has **position** 0. The i -th actual argument, for any applicable i , has **position** i .

Definition: Construction time, call time 725

The **construction time** of an agent object is the time of evaluation of the agent expression defining it.

Its **call time** is when a call to its associated operation is executed.

Syntactical forms for a call agent 733

A call agent is of the form

agent *agent_body*

where *agent_body* is a **Call**, qualified (as in $x.r(\dots)$) or unqualified (as in $f(\dots)$) with the following possible variants:

- You may replace any argument by a question mark **?**, making the argument open.
- You may replace the target, by $\{TYPE\}$ where *TYPE* is the name of a type, making the target open.
- You may remove the argument list (\dots) altogether, making all arguments open.

Syntax: Agents 751

Agent \triangleq **Call_agent** | **Inline_agent**

Call_agent \triangleq **agent** **Call_agent_body**

Inline_agent \triangleq **agent** [**Formal_arguments**] [**Type_mark**]
[**Attribute_or_routine**] [**Agent_actuals**]

Syntax: Call agent bodies

752

$\text{Call_agent_body} \triangleq \text{Agent_qualified} \mid \text{Agent_unqualified}$
 $\text{Agent_qualified} \triangleq \text{Agent_target} \cdot \cdot \text{Agent_unqualified}$
 $\text{Agent_unqualified} \triangleq \text{Feature_name} [\text{Agent_actuals}]$
 $\text{Agent_target} \triangleq \text{Entity} \mid \text{Parenthesized} \mid \text{Manifest_type}$
 $\text{Agent_actuals} \triangleq \text{"(" Agent_actual_list \text{"}"}$
 $\text{Agent_actual_list} \triangleq \{\text{Agent_actual} \text{";" } \dots\}^+$
 $\text{Agent_actual} \triangleq \text{Expression} \mid \text{Placeholder}$
 $\text{Placeholder} \triangleq [\text{Manifest_type} \text{"?"}$

Definition: Target type of an call agent

754

The **target type** of a **Call_agent** is:

- 1 • If there is no **Agent_target**, the **current type**.
- 2 • If there is an **Agent_target** and it is an **Entity** or **Parenthesized**, its type.
- 3 • If there is an **Agent_target** and it is a **Manifest_type**, the type that it lists (in braces).

Validity: Call Agent rule

VPCA 754

A **Call_agent** involving a **Feature_name** fn , appearing in a class C , with target type TO , is valid if and only if it satisfies the following conditions:

- 1 • fn is the **name of** a feature f of TO .
- 2 • If there is an **Agent_target**, f is **export-valid** for TO in C .
- 3 • If the **Agent_actuals** part is present, the number of elements in its **Agent_actual_list** is equal to the number of formals of f .
- 4 • Any **Agent_actual** of the **Expression** kind is of a type **compatible with** the type of the corresponding formal in f .

Definition: Associated feature of an inline agent

755

Every inline agent ia of a class C has an **associated feature**, defined as a fictitious routine f of C , such that:

- 1 • The name of f is chosen not to conflict with any other feature name in C and its descendants.
- 2 • The formal arguments of f are those of ia .
- 3 • f is **secret** (**available for call** to no class).
- 4 • The **Attribute_or_routine** part of f is defined by the **Attribute_or_routine** part of ia .
- 5 • f is a **function** if ia has a **Type_mark** (its return type being given by the **Type** in that **Type_mark**), a **procedure** otherwise.

Validity: Inline Agent rule

VPIA 755

An **Inline_agent** a of **associated feature** f , is valid in the text of a class C if and only if it satisfies the following conditions:

- 1 • f , if added to C , would be valid.
- 2 • f is not **deferred**.

Validity: Inline Agent Requirements

VPIR 756

An **Inline_agent** a must satisfy the following conditions:

- 1 • No formal argument or **local variable** of a has the **same name** as a feature of the enclosing class.
- 2 • Every **entity** appearing in the **Routine** part of a is the name of one of: a formal argument of a ; a local variable of a ; a feature of the enclosing class; **Current**.
- 3 • The **Feature_body** of a 's **Routine** is not of the **Deferred** form.

Definition: Call-agent equivalent of an inline agent

757

The **call-agent equivalent** of an inline agent ia is the **Call_agent agent** f

where f is the **associated feature** of ia .

Semantics: Semantics of inline agents

757

The semantic properties of an inline agent are those of its **call-agent equivalent**.

Semantics: Use of Result in an inline function agent

758

In an agent of the **Inline_agent** form denoting a function, the local variable **Result** denotes the result of the agent itself.

Definition: Open and closed operands

758

The **open operands** of a **Call_agent** include:

- 1 • Any **Agent_actual** that is a **Placeholder**.
- 2 • The **Agent_target** if it is present and is a **Manifest_type**.

The **closed operands** include all non-open **operands**.

Definition: Open and closed operand positions

759

The **open operand positions** of an **Agent** are the **operand positions** of its open operands, and the **closed operand positions** those of its closed operands.

Definition: Type of an agent expression

759

Consider a **Call_agent** a , with a **target** of type TO . Let $i1, \dots, im$ ($m \geq 0$) be its **open operand positions**, if any, and let T_{i1}, \dots, T_{im} be the types of f 's formal arguments at positions $i1, \dots, im$ (taking T_{i1} to be TO if $i1 = 0$).

The type of a is:

- **PROCEDURE** [TO , **TUPLE** [T_{i1} , ..., T_{im}]] if f is a **procedure**.
- **FUNCTION** [TO , **TUPLE** [T_{i1} , ..., T_{im}], R] if f is a **function** of result type R other than **BOOLEAN**.
- **PREDICATE** [TO , **TUPLE** [T_{i1} , ..., T_{im}]] if f is a function of result type **BOOLEAN**.

Semantics: Agent Expression semantics 759

The value of an agent expression a at a certain *construction time* yields a reference to an instance $D0$ of the type of a , containing information identifying:

- The associated feature of a .
- Its open operand positions.
- The values of its closed operands at the time of evaluation.

Semantics: Effect of executing call on an agent 760

Let $D0$ be an agent object with associated feature f and open positions $i1, \dots, im$ ($m \geq 0$). The information in $D0$ enables a call to the procedure *call*, executed at any **call time** posterior to $D0$'s construction time, with target $D0$ and (if required) actual arguments a_{i1}, \dots, a_{im} to perform the following:

- Produce the same effect as a call to f , using the closed operands at the closed operand positions and a_{i1}, \dots, a_{im} , evaluated at call time, at the open operand positions.
- In addition, if f is a function, setting the value of the query *last_result* for $D0$ to the result returned by such a call.

FROM CHAPTER 28: EXPRESSIONS**Syntax: Expressions** 761

Expression \triangleq Basic_expression | Special_expression
 Basic_expression \triangleq Read_only | Local | Call | Precursor |
 Equality | Parenthesized | Old | Operator_expression |
 Bracket_expression | Creation_expression
 Special_expression \triangleq Manifest_constant | Manifest_tuple |
 Agent | Object_test | Once_string | Address
 Parenthesized \triangleq "(" Expression ")"
 Address \triangleq "\$" Variable
 Once_string \triangleq once Manifest_string
 Boolean_expression \triangleq Basic_expression |
 Boolean_constant | Object_test

Definition: Subexpression, operand 764

The **subexpressions** of an expression e are e itself and (recursively) all the following expressions:

- 1 • For a **Parenthesized** (a) or a **Parenthesized_target** ((a)): the subexpressions of a .
- 2 • For an **Equality** or **Binary_expression** $a \ \$ \ b$, where $\$$ is an operator: the subexpressions of a and of b .
- 3 • For a **Unary_expression** $\diamond a$, where \diamond is an operator: the subexpressions of a .
- 4 • For a **Call**: the subexpressions of the **Actuals** part, if any, of its **Unqualified_part**.
- 5 • For a **Precursor**: the subexpressions of its unfolded form.
- 6 • For an **Agent**: the subexpression of its **Agent_actuals** if any.

- 7 • For a **qualified** call: the subexpressions of its **target**.

- 8 • For a **Bracket_expression** $f [a1, \dots a_n]$: the subexpressions of f and those of all of $a1, \dots a_n$.

- 9 • For an **Old** expression **old** a : a .

- 10 • For a **Manifest_tuple** $[a1, \dots a_n]$: the subexpressions of all of $a1, \dots a_n$.

In cases 2 and 3, the **operands** of e are a and (in case 2) b .

Semantics: Parenthesized Expression Semantics 765

If e is an expression, the value of the **Parenthesized** (e) is the value of e .

Syntax: Operator expressions 766

Operator_expression \triangleq Unary_expression |
 Binary_expression
 Unary_expression \triangleq Unary Expression
 Binary_expression \triangleq Expression Binary Expression

Operator precedence levels 768

- 13 • (Dot notation, in **qualified** and non-object calls)

- 12 **old** (In postconditions)

not + - Used as **unary**

All free unary operators

- 11 All free binary operators.

- 10 \wedge (Used as binary: power)

- 9 $* / // \backslash$ (As binary: multiplicative arithmetic operators)

- 8 $+ -$ Used as **binary**

- 7 $..$ (To define an interval)

- 6 $= / = \sim / \sim < > < = > =$ (As binary: relational operators)

- 5 **and and then** (Conjunctive boolean operators)

- 4 **or or else xor** (Disjunctive boolean operators)

- 3 **implies** (Implicative boolean operator)

- 2 $[]$ (Manifest tuple delimiter)

- 1 $;$ (Optional semicolon between an **Assertion_clause** and the next)

Definition: Parenthesized Form of an expression 769

The **parenthesized form** of an expression is the result of rewriting every subexpression of one of the forms below, where $\$$ and \ddagger are different binary operators, \diamond and \clubsuit different unary operators, and a, b, c arbitrary operands, as follows:

- 1 • For $a \ \$ \ b \ \$ \ c$ where $\$$ is not the power operator \wedge : ($a \ \$ \ b$) $\ \$ \ c$ (left associativity).

- 2 • For $a \ \wedge \ b \ \wedge \ c$: $a \ \wedge \ (b \ \wedge \ c)$ (right associativity).

- 3 • For $a \S b \ddagger c$: $(a \S b) \ddagger c$ if the precedence of \ddagger is lower than the precedence of \S or the same, and $a \S (b \ddagger c)$ otherwise.
- 4 • For $\diamond \clubsuit a$: $\diamond (\clubsuit a)$
- 5 • For $\diamond a \S b$: $(\diamond a) \S b$
- 6 • For $a \S \diamond b$: $a \S (\diamond b)$
- 7 • For a subexpression e to which none of the previous patterns applies: e unchanged.

Definition: Target-converted form of a binary expression 771

The **target-converted form** of a **Binary_expression** $x \S y$, where the one-argument feature of alias \S in the **base class** of x has the **Feature_name** f , is:

- 1 • If the declaration of f includes a **convert** mark and the type **TY** of y is not **compatible with** the type of the formal argument of f : $(\{TY\} [x]) \S y$.
- 2 • Otherwise: the original expression, $x \S y$.

Validity: Operator Expression rule VWOE 772

A **Unary_expression** x or **Binary_expression** $x \S y$, for some operator \S , is valid if and only if it satisfies the following conditions:

- 1 • A feature of the **base class** of x is declared as **alias** " \S ".
- 2 • The expression's **Equivalent Dot Form** is a valid **Call**.

Semantics: Expression Semantics (strict case) 773

The **value** of an **Expression**, other than a **Binary_expression** whose **Binary** is **semistrict**, is the **value** of its **Equivalent Dot Form**.

Definition: Semistrict operators 774

A **semistrict operator** is any one of the three operators **and then, or else** and **implies**, applied to **operands** of type **BOOLEAN**.

Semantics: Operator Expression Semantics (semistrict cases) 777

For a and b of type **BOOLEAN**:

- The value of **a and then b** is: if a has value false, then false; otherwise the value of b .
- The value of **a or else b** is: if a has value true, then true; otherwise the value of b .
- The value of **a implies b** is: if a has value false, then true; otherwise the value of b .

Syntax: Bracket expressions 778

Bracket_expression \triangleq **Bracket_target** "[" **Actuals** "]"
Bracket_target \triangleq **Target** | **Once_string** |
Manifest_constant | **Manifest_tuple**

Validity: Bracket Expression rule VWBR 780

A **Bracket_expression** $x [i]$ is valid if and only if it satisfies the following conditions:

- 1 • A feature of the **base class** of x is declared as **alias** "["].
- 2 • The expression's **Equivalent Dot Form** is a valid **Call**.

Definition: Equivalent Dot Form of an expression 780

Any **Expression** e has an **Equivalent Dot Form**, not involving (in any of its **subexpressions**) any **Bracket_expression** or **Operator_expression**, and defined as follows, where C denotes the **base class** of x , pe denotes the **Parenthesized Form** of e , and x', y', c' denote the **Equivalent Dot Forms** (obtained recursively) of x, y, c :

- 1 • If pe is a **Unary_expression** $x' \cdot f$, where f is the **Feature_name** of the no-argument feature of alias \S in C .
- 2 • If pe is a **Binary_expression** of **target-converted form** $x \S y$: $x' \cdot f(y')$ where f is the **Feature_name** of the one-argument feature of alias \S in C .
- 3 • If pe is a **Bracket_expression** $x [y]$: $x' \cdot f(y')$ where f is the **Feature_name** of the feature declared as **alias** "["] in C .
- 4 • If pe has no **subexpression** other than itself: pe .
- 5 • In all other cases: (recursively) the result of replacing every **subexpression** of e by its **Equivalent Dot Form**.

Validity: Boolean Expression rule VWBE 781

A **Basic_expression** is valid as a **Boolean_expression** if and only if it is of type **BOOLEAN**.

Validity: Identifier rule VWID 782

An **Identifier** appearing in an expression in a class C , other than as the **feature of a qualified Call**, must be the **name** of a feature of C , or a **local variable** of the enclosing feature or inline agent if any, or a formal argument of the enclosing feature or inline agent if any, or the **Object-Test Local** of an **Object_test**.

Definition: Type of an expression 783

The type of an **Expression** e is:

- 1 • For the predefined **Read-only Current**: the **current type**.
- 2 • For a routine's **Formal** argument: the type declared for e .
- 3 • For an **Object-Test local**: its declared type.
- 4 • For **Result**, appearing in the text of a query f : the result type of f .
- 5 • For a **local variable** other than **Result**: the type declared for e .
- 6 • For a **Call**: the type of e as determined by the **Expression Call Type** definition with respect to the current type.

- 7 • For a **Precursor**: (recursively) the type of its unfolded form.
- 8 • For an **Equality**: *BOOLEAN*.
- 9 • For a **Parenthesized** (*f*): (recursively) the type of *f*.
- 10 • For **old** *f*: (recursively) the type of *f*.
- 11 • For an **Operator_expression** or **Bracket_expression**: (recursively) the type of the Equivalent Dot Form of *e*.
- 12 • For a **Manifest_constant**: as given by the definition of the type of a manifest constant.
- 13 • For a **Manifest_tuple** [*a₁*, ... *a_n*] (*n* ≥ 0): *TUPLE* [*T₁*, ... *T_n*] where each *T_i* is (recursively) the type of *a_i*.
- 14 • For an **Agent**: as given by the definition of the type of an agent expression.
- 15 • For an **Object_test**: *BOOLEAN*.
- 16 • For a **Once_string**: *STRING*.
- 17 • For an **Address** *\$v*: *TYPED_POINTER* [*T*] where *T* is (recursively) the type of *v*.
- 18 • For a **Creation_expression**: the Explicit_creation_type.

FROM CHAPTER 29: CONSTANTS

Syntax: Constants 787

Constant \triangleq **Manifest_constant** | **Constant_attribute**
Constant_attribute \triangleq **Feature_name**

Validity: Constant Attribute rule VWCA 787

A **Constant_attribute** appearing in a class *C* is valid if and only if its **Feature_name** is the final name of a constant attribute of *C*.

Syntax: Manifest constants 788

Manifest_constant \triangleq [**Manifest_type**] **Manifest_value**
Manifest_type \triangleq "{" **Type** "**"**
Manifest_value \triangleq **Boolean_constant** |
Character_constant |
Integer_constant |
Real_constant |
Manifest_string |
Manifest_type
Sign \triangleq "+" | "-"
Integer_constant \triangleq [**Sign**] **Integer**
Character_constant \triangleq "" | "Character""
Boolean_constant \triangleq *True* | *False*
Real_constant \triangleq [**Sign**] **Real**

Syntax (non-production): Sign Syntax rule 788

If present, the **Sign** of an **Integer_constant** or **Real_constant** must immediately precede the associated **Integer** or **Real**, with no intervening tokens or components (such as breaks or comments).

Syntax (non-production): Character Syntax rule 788

The quotes of a **Character_constant** must immediately precede and follow the **Character**, with no intervening tokens or components (such as breaks or comments).

Definition: Type of a manifest constant 790

The type of a **Manifest_constant** of **Manifest_value** *mv* is:

- 1 • For **{*T*}** *mv*, with the optional **Manifest_type** present: *T*. The remaining cases assume this optional component is absent, and only involve *mv*.
- 2 • If *mv* is a **Boolean_constant**: *BOOLEAN*.
- 3 • If *mv* is a **Character_constant**: *CHARACTER*.
- 4 • If *mv* is an **Integer_constant**: *INTEGER*.
- 5 • If *mv* is a **Real_constant**: *REAL*.
- 6 • If *mv* is a **Manifest_string**: *STRING*.
- 7 • If *mv* is a **Manifest_type** **{*T*}**: *TYPE* [*T*].

Validity: Manifest-Type Qualifier rule VWMQ 791

It is valid for a **Manifest_constant** to be of the form **{*T*}** *v* (with the optional **Manifest_type** qualifier present) if and only if the type *U* of *v* (as determined by cases 2 to 7 of the definition of the type of a manifest constant) is one of *CHARACTER*, *STRING*, *INTEGER* and *REAL*, and *T* is one of the sized variants of *U*.

Semantics: Manifest Constant Semantics 791

The **value** of a **Manifest_constant** *c* listing a **Manifest_value** *v* is:

- 1 • If *c* is of the form **{*T*}** *v* (with the optional **Manifest_type** qualifier present): the value of type *T* denoted by *v*.
- 2 • Otherwise (*c* is just *v*): the value denoted by *v*.

Definition: Manifest value of a constant 792

The **manifest value** of a constant is:

- 1 • If it is a **Manifest_constant**: its value.
- 2 • If it is a constant attribute: (recursively) the manifest value of the **Manifest_constant** listed in its declaration.

Syntax: Manifest strings 795

Manifest_string \triangleq **Basic_manifest_string** | **Verbatim_string**
Basic_manifest_string \triangleq "" | **String_content** ""
String_content \triangleq {**Simple_string** **Line_wrapping_part** ...}⁺
Verbatim_string \triangleq **Verbatim_string_opener** **Line_sequence** **Verbatim_string_closer**
Verbatim_string_opener \triangleq "" | [**Simple_string**]
Open_bracket
Verbatim_string_closer \triangleq **Close_bracket** [**Simple_string**]'
""
Open_bracket \triangleq "[" | "{"
Close_bracket \triangleq "]" | "}"

Syntax (non-production): Line sequence 796

A specimen of Line_sequence is a sequence of one or more Simple_string components, each separated from the next by a single New_line.

Syntax (non-production): Manifest String rule 796

In addition to the properties specified by the grammar, every Manifest_string must satisfy the following properties:

- 1 • The Simple_string components of its String_content or Line_sequence may not include a double quote character except as part of the character code `%"` (denoting a double quote).
- 2 • A Verbatim_string_opener or Verbatim_string_closer may not contain any break character.

Definition: Line_wrapping_part 797

A Line_wrapping_part is a sequence of characters consisting of the following, in order: `%` (percent character); zero or more blanks or tabs; New_line; zero or more blanks or tabs; `%` again.

Semantics: Manifest string semantics 798

The value of a Basic_manifest_string is the sequence of characters that it includes, in the order given, excluding any line wrapping parts, and with any character_code replaced by the corresponding character.

Validity: Verbatim String rule *VWVS* 800

A Verbatim_string is valid if and only if it satisfies the following conditions, where α is the (possibly empty) Simple_string appearing in its Verbatim_string_opener:

- 1 • The Close_bracket is `]` if the Open_bracket is `[`, and `}` if the Open_bracket is `{`.
- 2 • Every character in α is printable, and not a double quote `"`.
- 3 • If α is not empty, the string's Verbatim_string_closer includes a Simple_string identical to α .

Semantics: Verbatim string semantics 800

The value of a Line_sequence is the string obtained by concatenating the characters of its successive lines, with a “new line” character inserted between any adjacent ones.

The value of a Verbatim_string using braces `{ }` as Open_bracket and Close_bracket is the value of its Line_sequence.

The value of a Verbatim_string using braces `[]` as Open_bracket and Close_bracket is the value of the left-aligned form of its Line_sequence.

Definition: Prefix, longest break prefix, left-aligned form 804

A prefix of a string s is a string p of some length n ($n \geq 0$) such that the first n characters of s are the corresponding characters of p .

The longest break prefix of a sequence of strings ls is the longest string bp containing no characters other than spaces and tabs, such that bp is a prefix of every string in ls . (The longest break prefix is always defined, although it may be an empty string.)

The left-aligned form of a sequence of strings ls is the sequence of strings obtained from the corresponding strings in ls by removing the first n characters, where n is the length of the longest break prefix of ls ($n \geq 0$).

FROM CHAPTER 30: BASIC TYPES**Definition: Basic types and their sized variants** 817

A basic type is any of the types defined by the following ELKS classes:

- BOOLEAN.
- CHARACTER, CHARACTER_8, CHARACTER_32, together called the “sized variants of CHARACTER”.
- INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_64, NATURAL, NATURAL_8, NATURAL_16, NATURAL_32, NATURAL_64, together called the “sized variants of INTEGER”.
- REAL, REAL_32, REAL_64, together called the “sized variants of REAL”.
- POINTER.

Definition: Sized variants of STRING 817

The sized variants of STRING are STRING, STRING_8 and STRING_32.

Semantics: Boolean value semantics 819

Class BOOLEAN covers the two truth values.

The reserved words True and False denote the corresponding constants.

Semantics: Character types 819

The reference class CHARACTER_GENERAL describes properties of characters independently of the character code.

The expanded class CHARACTER_32 describes Unicode characters; the expanded class CHARACTER_8 describes 8-bit (ASCII-like) characters.

The expanded class CHARACTER describes characters with a length and encoding settable through a compilation option. The recommended default is Unicode.

Semantics: Integer types 820

The reference class *INTEGER_GENERAL* describes integers, signed or not, of arbitrary length. The expanded classes *INTEGER_xx*, for *xx* = 8, 16, 32 or 64, describe signed integers stored on *xx* bits. The expanded classes *NATURAL_xx*, for *xx* = 8, 16, 32 or 64, describe unsigned integers stored on *xx* bits.

The expanded classes *INTEGER* and *NATURAL* describe integers, respectively signed and unsigned, with a length setttable through a compilation option. The recommended default is 64 bits in both cases.

Semantics: Floating-point types 821

The reference class *REAL_GENERAL* describes floating-point numbers with arbitrary precision. The expanded classes *REAL_xx*, for *xx* = 32 or 64, describe IEEE floating-point numbers with *xx* bits of precision.

The expanded class *REAL* describes floating-point numbers with a precision setttable through a compilation option. The recommended default is 64 bits.

Semantics: Address semantics 821

The expanded class *POINTER* describes addresses of data beyond the control of Eiffel systems.

FROM CHAPTER 31: INTERFACING WITH C, C++ AND OTHER ENVIRONMENTS

Syntax: External routines 829

```
External  $\triangleq$  external External_language [External_name]
External_language  $\triangleq$  Unregistered_language |
Registered_language
Unregistered_language  $\triangleq$  Manifest_string
External_name  $\triangleq$  alias Manifest_string
```

Semantics: Address semantics 835

The value of an *Address* expression is an address enabling foreign software to access the associated *Variable*.

Syntax: Registered languages 837

```
Registered_language  $\triangleq$  C_external | C++_external |
DLL_external
```

Syntax: External signatures 839

```
External_signature  $\triangleq$  signature
[External_argument_types] [: External_type]
External_argument_types  $\triangleq$  "(" External_type_list ")"
External_type_list  $\triangleq$  {External_type "," ...}*
External_type  $\triangleq$  Simple_string
```

Validity: External Signature rule *VZES* 839

An *External_signature* in the declaration of an external *routine* *r* is valid if and only if it satisfies the following conditions:

- 1 • Its *External_type_list* contains the same number of elements as *r* has formal arguments.
- 2 • The final optional component (*:* *External_type*) if present if and only if *r* is a *function*.

A *language processing tool* may delegate enforcement of these requirements to non-Eiffel tools on the chosen *platform*.

Semantics: External signature semantics 840

An *External_signature* specifies that the associated external routine:

- Expects arguments of number and types as given by the *External_argument_types* if present, and no arguments otherwise.
- Returns a result of the *External_type* appearing after the colon, if present, and otherwise no result.

Syntax: External file use 840

```
External_file_use  $\triangleq$  use External_file_list
External_file_list  $\triangleq$  {External_file "," ...}+
External_file  $\triangleq$  External_user_file | External_system_file
External_user_file  $\triangleq$  "'" Simple_string "'"
External_system_file  $\triangleq$  "<"Simple_string ">"
```

Validity: External File rule *VZEF* 841

An *External_file* is valid if and only if its *Simple_string* satisfies the following conditions:

- 1 • When interpreted as a file name according to the conventions of the underlying *platform*, it denotes a file.
- 2 • The file is accessible for reading.
- 3 • The file's content satisfies the rules of the applicable foreign language.

A *language processing tool* may delegate enforcement of these conditions to non-Eiffel tools on the chosen *platform*.

Semantics: External file semantics 842

An *External_file_use* in an external *routine* declaration specifies that foreign language tools, to process the routine (for example to compile its original code), require access to the listed files.

Syntax: C externals 843

```
C_external  $\triangleq$  "'" C
[inline]
[External_signature] [External_file_use]
'"'
```

Validity: C external rule *VZCC* 846

A *C_external* for the declaration of an external *routine* *r* is valid if and only if it satisfies the following conditions:

- 1 • At least one of the optional *inline* and *External_signature* components is present.

An **alpha_numeric character** is alpha_betic or numeric.

A **printable character** is any of the characters listed as printable in the definition of the character set (Unicode or extended ASCII).

Definition: Break character, break 881

A **break character** is one of the following characters:

- Blank (also known as space).
- Tab.
- New Line (also known as Line Feed).
- Return (also known as Carriage Return).

A **break** is a sequence of one or more break characters that is not part of a **Character_constant**, of a **Manifest_string** or of a **Simple_string component** of a **Comment**.

Semantics: Break semantics 881

Breaks serve a purely **syntactical** role, to separate **tokens**. The effect of a break is independent of its makeup (its precise use of spaces, tabs and newlines). In particular, the separation of a class text into lines has no effect on its semantics.

Definition: Expected, free comment 882

A comment is **expected** if it appears in a **construct** as part of the style guidelines for that construct. Otherwise it is **free**.

Syntax (non-production): “Blanks or tabs”, new line 883

A **specimen** of **Blanks_or_tabs** is any non-empty sequence of **characters**, each of which is a blank or a tab.

A specimen of **New_line** is a New Line.

Syntax: Comments 883

Comment \triangleq “**---**” {**Simple_string Comment_break ...**}*
Comment_break \triangleq **New_line** [**Blanks_or_tabs**] “**---**”

Syntax (non-production): Free Comment rule 884

It is permitted to include a **free comment** between any two successive **components** of a **specimen** of a **construct** defined by a BNF-E **production**, except if excluded by specific syntax rules.

Header comment rule 884

A feature **Header_comment** is an abbreviation for a **Note** clause of the form

note

what: *Explanation*

where *Explanation* is a **Verbatim_string** with [**]** and [**]** as **Open_bracket** and **Close_bracket** and a **Line_sequence** made up of the successive lines (**Simple_string**) of the comment, each deprived of its first characters up to and including the first two consecutive dash characters, and of the space immediately following them if any.

Definition: Symbol, word 885

A **symbol** is either a **special symbol** of the language, such as the semicolon “**;**” and the “**.**” of dot notation, or a **standard operator** such as “**+**” and “*****”.

A **word** is any token that is not a symbol. Examples of words include identifiers, **keywords**, **free operators** and non-symbol operators such as **or else**.

Syntax (non-production): Break rule 885

It is permitted to write two adjacent **tokens** without an intervening **break** if and only if they satisfy one of the following conditions:

- 1 • One is a **word** and the other is a **symbol**.
- 2 • They are both symbols, and their concatenation is not a symbol.

Semantics: Letter Case rule 886

Letter case is significant for the following **constructs**: **Character_constant** and **Manifest_string** except for special character codes, **Comment**.

For all other constructs, letter case is not significant: changing a **letter** to its lower-case or upper-case counterpart does not affect the semantics of a **specimen** of the construct.

Definition: Reserved word, keyword 888

The following names are **reserved words** of the language.

agent	alias	all	and	as
assign	attribute			
check	class	convert	create	Current
debug	deferred			
do	else	elseif	end	ensure
expandedexport				
external	False	feature	from	frozenif
implies				
inherit	inspect	invariant	like	local
loop	not			
note	obsolete	old	once	only or
Precursor				
redefine	rename	require	rescue	Result
retry	select			
separate	then	True	TUPLE	undefine
until	variant			
Void	when	xor		

The reserved words that serve as purely syntactical markers, not carrying a direct semantic value, are called **keywords**; they appear in the above list in all lower-case letters.

Syntax (non-production): Double Reserved Word rule 889

The **reserved words** **and then** and **or else** are each made of two **components** separated by one or more

blanks (but no other break characters). Every other reserved word is a sequence of letters with no intervening break character.

Definition: Special symbol 890

A **special symbol** is any of the following character sequences:

```
-- ; ; , ? ! ' " $ . -> :=
= /= ~ /~ ( ) ( | ) [ ] { }
```

Syntax (non-production): Identifier 891

An **Identifier** is a sequence of one or more alpha numeric characters of which the first is a letter.

Validity: Identifier rule *VIID* 891

An **Identifier** is valid if and only if it is not one of the language's reserved words.

Definition: Predefined operator 892

A **predefined operator** is one of:

```
= /= ~ /~
```

Definition: Standard operator 892

A **standard unary operator** is one of:

```
+ -
```

A **standard binary operator** is any one of the following one- or two-character symbols:

```
+ - * / ^ < >
<= >= // \ \ ..
```

Definition: Operator symbol 893

An **operator symbol** is any non-alpha numeric printable character that satisfies any of the following properties:

- 1 • It does not appear in any of the special symbols.
- 2 • It appears in any of the standard (unary or binary) operators but is neither a dot `.` nor an equal sign `=`.
- 3 • It is a tilde `~`, percent `%`, question mark `?`, or exclamation mark `!`.

Definition: Free operator 893

A **free operator** is sequence of one or more characters satisfying the following properties:

- 1 • It is not a special symbol, standard operator or predefined operator.
- 2 • Every character in the sequence is an operator symbol.
- 3 • Every subsequence that is not a standard operator or predefined operator is distinct from all special symbols.

A **Free unary** is a free operator that is distinct from all standard unary operators.

A **Free binary** is a free operator that is distinct from all standard binary operators.

Syntax (non-production): Manifest character 895

A **manifest character** — specimen of construct **Character** — is one of the following:

- 1 • Any key associated with a printable character, except for the percent key `%`.
- 2 • The sequence `%k`, where *k* is a one-key code taken from the list of special characters.
- 3 • The sequence `%/code/`, where *code* is an unsigned integer in any of the available forms — decimal, binary, octal, hexadecimal — corresponding to a valid character code in the chosen character set.

Special characters and their codes 897

Character	Code	Mnemonic name
@	%A	A t-sign
BS	%B	B ackspace
^	%C	C ircumflex
\$	%D	D ollar
FF	%F	F orm feed
\	%H	B ackslas H
~	%L	TiL de
NL (LF)	%N	N ewline
`	%Q	B ack Q uote
CR	%R	C arriage R eturn
#	%S	S harp
HT	%T	H orizontal T ab
NUL	%U	N Ull
	%V	V ertical bar
%	%%	P ercent
'	%'	S ingle quote
"	%"	D ouble quote
[%(O pening bracket
]	%)	C losing bracket
{	%<	O pening brace
}	%>	C losing brace

Syntax (non-production): Percent variants 898

The percent forms of **Character** are available for the manifest characters of a **Character_constant** and of the **Simple_string** components of a **Manifest_string**, but not for any other token.

Semantics: Manifest character semantics 898

The value of a **Character** is:

- 1 • If it is a printable character *c* other than `%`: *c*.
- 2 • If it is of the form `%k` for a one-key code *k*: the corresponding character as given by the table of special characters.
- 3 • If it is of the form `%/code/`: the character of code *code* in the chosen character set.

Syntax (non-production): String, simple string 899

A **string** — specimen of construct **String** — is a sequence of zero or more manifest characters.

A **simple string** — specimen of `Simple_string` — is a `String` consisting of at most one line (that is to say, containing no embedded new-line manifest character).

Semantics: String semantics 899

The value of a `String` or `Simple_string` is the sequence of the values of its characters.

Syntax: Integers 900

`Integer` \triangleq [`Integer_base`] `Digit_sequence`
`Integer_base` \triangleq "0" `Integer_base_letter`
`Integer_base_letter` \triangleq "b" | "c" | "x" | "B" | "C" | "X"
`Digit_sequence` \triangleq `Digit`⁺
`Digit` \triangleq "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" |
 "a" | "b" | "c" | "d" | "e" | "f" |
 "A" | "B" | "C" | "D" | "E" | "F" | "_"

Validity: Integer rule *VIII* 901

An `Integer` is valid if and only if it satisfies the following conditions:

- 1 • It contains no breaks.
- 2 • Neither the first nor the last `Digit` of the `Digit_sequence` is an underscore “_”.
- 3 • If there is no `Integer_base` (decimal integer), every `Digit` is either one of the decimal digits **0** to **9** (zero to nine) or an underscore.
- 4 • If there is an `Integer_base` of the form **0b** or **0B** (binary integer), every `Digit` is either **0**, **1** or an underscore.
- 5 • If there is an `Integer_base` of the form **0c** or **0C** (octal integer), every `Digit` is either one of the octal digits **0** to **7** or an underscore.

Semantics: Integer semantics 901

The value of an `Integer` is the integer constant denoted in ordinary mathematical notation by the `Digit_sequence`, without its underscores if any, in the corresponding base: binary if the `Integer` starts with **0b** or **0B**, octal if it starts with **0c** or **0C**, hexadecimal if it starts with **0x** or **0X**, decimal otherwise.

Syntax (non-production): Real number 902

A **real** — specimen of `Real` — is made of the following elements, in the order given:

- An optional decimal `Integer`, giving the integral part.
- A required “.” (dot).
- An optional decimal `Integer`, giving the fractional part.
- An optional exponent, which is the letter *e* or *E* followed by an optional `Sign` (+ or –) and a decimal `Integer`.

No intervening character (blank or otherwise) is permitted between these elements. The integral and fractional parts may not both be absent.

Semantics: Real semantics 902

The value of a `Real` is the real number that would be expressed in ordinary mathematical notation as $i.f10^e$, where *i* is the integral part, *f* the fractional part and *e* the exponent (or, in each case, zero if the corresponding part is absent).

K

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

Syntax in alphabetical order

K.1 OVERVIEW

This appendix gives the entire syntax specification, with the various productions appearing in alphabetical order of construct names, each with the number of the page where it appeared.

K.2 SYNTAX

Actual_generics	\triangleq "[" Type_list "]"	350
Actual_list	\triangleq {Expression "," ...} ⁺	626
Actuals	\triangleq "(" Actual_list ")"	626
Address	\triangleq "\$" Variable	761
Agent	\triangleq Call_agent Inline_agent	751
Agent_actual	\triangleq Expression Placeholder	752
Agent_actual_list	\triangleq {Agent_actual "," ...} ⁺	752
Agent_actuals	\triangleq "(" Agent_actual_list ")"	752
Agent_qualified	\triangleq Agent_target "." Agent_unqualified	752
Agent_target	\triangleq Entity Parenthesized Manifest_type	752
Agent_unqualified	\triangleq Feature_name [Agent_actuals]	752
Alias	\triangleq alias "" Alias_name "" [convert]	151
Alias_name	\triangleq Operator Bracket	151
Anchor	\triangleq Feature_name Current	328
Anchored	\triangleq [Attachment_mark] like Anchor	328
Assertion	\triangleq {Assertion_clause "," ...} [*]	232
Assertion_clause	\triangleq [Tag_mark] Unlabeled_assertion_clause	232
Assigner_call	\triangleq Expression ":@" Expression	609
Assigner_mark	\triangleq assign Feature_name	155
Assignment	\triangleq Variable ":@" Expression	589

Class_declaration	≙ [Notes] Class_header [Formal_generics] [Obsolete] [Inheritance] [Creators] [Converters] [Features] [Invariant] [Notes] end	119
Class_header	≙ [Header_mark] class Class_name	124
Class_list	≙ {Class_name "," ...}+	208
Class_name	≙ Identifier	110
Class_or_tuple_type	≙ Class_type Tuple_type	328
Class_type	≙ [Attachment_mark] Class_name [Actual_generics]	328
Clients	≙ "{" Class_list "}"	208
Close_bracket	≙ "]" "}"	795
Comment	≙ "--" {Simple_string Comment_break ...}*	883
Comment_break	≙ New_line [Blanks_or_tabs] "--"	883
Comparison	≙ "=" "/=" "~" "/~"	567
Compound	≙ {Instruction ";" ...}*	228
Conditional	≙ if Then_part_list [Else_part] end	481
Constant	≙ Manifest_constant Constant_attribute	787
Constant_attribute	≙ Feature_name	513
Constant_attribute	≙ Feature_name	787
Constant_interval	≙ Constant ".." Constant	485
Constraining_types	≙ Single_constraint Multiple_constraint	357
Constraint	≙ "->" Constraining_types [Constraint_creators]	357
Constraint_creators	≙ create Feature_list end	357
Constraint_list	≙ {Single_constraint "," ...}+	357
Conversion_procedure	≙ Feature_name "(" {" Type_list "}" ")"	410
Conversion_query	≙ Feature_name ":" {" Type_list "}"	410
Converter	≙ Conversion_procedure Conversion_query	410
Converter_list	≙ {Converter "," ...}+	410
Converters	≙ convert Converter_list	410
Creation_call	≙ Variable [Explicit_creation_call]	551

Creation_clause	≙	create [Clients] [Header_comment] Creation_procedure_list	547
Creation_expression	≙	create Explicit_creation_type [Explicit_creation_call]	561
Creation_instruction	≙	create [Explicit_creation_type] Creation_call	551
Creation_procedure	≙	Feature_name	547
Creation_procedure_list	≙	{Creation_procedure "," ...}+	547
Creators	≙	Creation_clause ⁺	547
Debug	≙	debug ["("Key_list ")"] Compound end	498
Declaration_body	≙	[Formal_arguments] [Query_mark] [Feature_value]	141
Deferred	≙	deferred	222
Digit	≙	"0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "a" "b" "c" "d" "e" "f" "A" "B" "C" "D" "E" "F" "_"	900
Digit_sequence	≙	Digit ⁺	900
DLL_external	≙	' " ' dll [windows] DLL_identifier [DLL_index] [External_signature] [External_file_use] ' " '	857
DLL_identifier	≙	Simple_string	857
DLL_index	≙	Integer	857
Effective_routine	≙	Internal External	222
Else_part	≙	else Compound	481
Entity	≙	Variable Read_only	512
Entity_declaration_group	≙	Identifier_list Type_mark	220
Entity_declaration_list	≙	{Entity_declaration_group ";" ...}+	220
Equality	≙	Expression Comparison Expression	567
Exit_condition	≙	until Boolean_expression	495
Explicit_creation_call	≙	"." Unqualified_call	551
Explicit_creation_type	≙	"{" Type "}"	551
Explicit_value	≙	"=" Manifest_constant	141
Expression	≙	Basic_expression Special_expression	761
Expression_list	≙	{Expression "," ...}*	373
Extended_feature_name	≙	Feature_name [Alias]	151
External	≙	external External_language [External_name]	829
External_argument_types	≙	"(" External_type_list ")"	839

Inherit_clause	△	inherit [Non_conformance] Parent_list	171
Inheritance	△	Inherit_clause ⁺	171
Initialization	△	from Compound	495
Inline_agent	△	agent [Formal_arguments] [Type_mark] [Attribute_or_routine] [Agent_actuals]	751
Instruction	△	Creation_instruction Call Assignment Assigner_call Conditional Multi_branch Loop Debug Precursor Check Retry	228
Integer	△	[Integer_base] Digit_sequence	900
Integer_base	△	"0" Integer_base_letter	900
Integer_base_letter	△	"b" "c" "x" "B" "C" "X"	900
Integer_constant	△	[Sign] Integer	788
Internal	△	Routine_mark Compound	222
Invariant	△	invariant Assertion	232
Key_list	△	{Manifest_string "," ...} ⁺	222
Local	△	Identifier Result	513
Local_declarations	△	local [Entity_declaration_list]	225
Loop	△	Initialization [Invariant] Exit_condition Loop_body [Variant] end	495
Loop_body	△	loop Compound	495
Manifest_constant	△	[Manifest_type] Manifest_value	788
Manifest_string	△	Basic_manifest_string Verbatim_string	795
Manifest_tuple	△	"[" Expression_list "]"	373
Manifest_type	△	"{" Type "}"	788
Manifest_value	△	Boolean_constant Character_constant Integer_constant Real_constant Manifest_string Manifest_type	788
Message	△	Manifest_string	129
Multi_branch	△	inspect Expression [When_part_list] [Else_part] end	485
Multiple_constraint	△	"{" Constraint_list "}"	357
New_export_item	△	Clients [Header_comment] Feature_set	209
New_export_list	△	{New_export_item ";" ...} ⁺	209

New_exports	△ export New_export_list	209
New_feature	△ [frozen] Extended_feature_name	141
New_feature_list	△ {New_feature ", " ... } ⁺	141
Non_conformance	△ {" <i>NONE</i> "}	171
Non_object_call	△ {" Type " } " . " Unqualified_call	626
Note_entry	△ Note_name Note_values	123
Note_item	△ Identifier Manifest_constant	123
Note_list	△ {Note_entry "; " ... } [*]	123
Note_name	△ Identifier ":"	123
Note_values	△ {Note_item ", " ... } ⁺	123
Notes	△ note Note_list	123
Object_call	△ [Target ". "] Unqualified_call	626
Object_test	△ {" Identifier ":" Type " } Expression	658
Obsolete	△ obsolete Message	129
Old	△ old Expression	239
Once	△ once ["(" Key_list ")"]	222
Once_string	△ once Manifest_string	761
Only	△ only [Feature_list]	242
Open_bracket	△ "[" "{"	795
Operator	△ Unary Binary	154
Operator_expression	△ Unary_expression Binary_expression	766
Parent	△ Class_type [Feature_adaptation]	171
Parent_list	△ {Parent "; " ... } ⁺	171
Parent_qualification	△ {" Class_name "}	303
Parenthesized	△ "(" Expression ")"	761
Parenthesized_target	△ "(" Expression ")"	626
Placeholder	△ [Manifest_type] "?"	752
Postcondition	△ ensure [then] Assertion [Only]	232
Precondition	△ require [else] Assertion	232
Precursor	△ Precursor [Parent_qualification] [Actuals]	303
Query_mark	△ Type_mark [Assigner_mark]	141
Read_only	△ Formal Constant_attribute Current	513
Real_constant	△ [Sign] Real	788
Redefine	△ redefine Feature_list	307
Registered_language	△ C_external C++_external DLL_external	837

Rename	≙ rename Rename_list	183
Rename_list	≙ {Rename_pair "," ...}+	183
Rename_pair	≙ Feature_name as Extended_feature_name	183
Renaming	≙ Rename end	357
Rescue	≙ rescue Compound	701
Retry	≙ retry	701
Routine_mark	≙ do Once	222
Select	≙ select Feature_list	463
Sign	≙ "+" "-"	788
Single_constraint	≙ Type [Renaming]	357
Special_expression	≙ Manifest_constant Manifest_tuple Agent Object_test Once_string Address	761
String_content	≙ {Simple_string Line_wrapping_part ...}+	795
Tag	≙ Identifier	232
Tag_mark	≙ Tag ":"	232
Target	≙ Local Read_only Call Parenthesized_target	626
Then_part	≙ Boolean_expression then Compound	481
Then_part_list	≙ {Then_part elseif ...}+	481
Tuple_parameter_list	≙ "[" Tuple_parameters "]"	372
Tuple_parameters	≙ Type_list Entity_declaration_list	372
Tuple_type	≙ TUPLE [Tuple_parameter_list]	372
Type	≙ Class_or_tuple_type Formal_generic_name Anchored	328
Type_interval	≙ Manifest_type .. Manifest_type	485
Type_list	≙ {Type "," ...}+	350
Type_mark	≙ ":" Type	141
Unary	≙ not "+" "-" Free_unary	154
Unary_expression	≙ Unary Expression	766
Undefine	≙ undefine Feature_list	308
Unlabeled_assertion_clause	≙ Boolean_expression Comment	232
Unqualified_call	≙ Feature_name [Actuals]	626
Unregistered_language	≙ Manifest_string	829
Variable	≙ Variable_attribute Local	512
Variable_attribute	≙ Feature_name	512
Variant	≙ variant [Tag_mark] Expression	251
Verbatim_string	≙ Verbatim_string_opener Line_sequence Verbatim_string_closer	795

Verbatim_string_closer	≙	Close_bracket [Simple_string] ' "'	795
Verbatim_string_opener	≙	' "' [Simple_string] Open_bracket	795
When_part	≙	when Choices then Compound	485
When_part_list	≙	When_part ⁺	485

L

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of “Eiffel: The Language”. Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

Reserved words, special symbols, operator precedence

L.1 OVERVIEW

This chapter lists the reserved words — including keywords for the external interface sublanguages —, the reserved special (non-alphabetic) symbols, and the precedence of operators appearing in expressions.

L.2 RESERVED WORDS

Following are the sixty-two reserved words of Eiffel, in alphabetical order.

Recall the distinction between *reserved words* and their special case, *keywords*. Reserved words include all the names (listed below) that cannot be used as identifiers for classes, features or entities. Some reserved words carry a meaning of their own, such as *Current* which denotes an expression and *TUPLE* which denotes a type. These are typeset in italics, with a first letter in upper case (all letters upper-case in the case of a type or class name). Reserved words that do *not* by themselves denote anything but just serve as syntactic markers, such as **do** or **if**, are called keywords and appear in boldface.

Every reserved word (keyword or not) has an entry in the index, with a reference to the page of the corresponding syntax productions, if any.

agent	alias	all	and	as	assign	attribute
check	class	convert	create	<i>Current</i>	debug	deferred
do	else	elseif	end	ensure	expanded	export
external	<i>False</i>	feature	from	frozen	if	implies
inherit	inspect	invariant	like	local	loop	not
note	obsolete	old	once	only	or	<i>Precursor</i>
redefine	rename	require	rescue	<i>Result</i>	retry	select
separate	then	<i>True</i>	<i>TUPLE</i>	undefine	until	variant
<i>Void</i>	when	xor				

L.3 SPECIAL SYMBOLS

The following table shows all the special symbols of the language, together with the page of the syntax productions where they appear. *← This table appeared first on page 890.*

Symbol	Name	Role	Pages
--	Double dash	Introduces comments.	
;	Semicolon	Separates instructions, declarations, assertion clauses...; always optional.	
,	Comma	Separates elements in lists of of entities or expressions.	
:	Colon	Separates the Type_mark in a declaration, a Tag_mark in an Assertion_clause , and a Note_name term in a Notes clause.	
?: :!	Colon-question, colon-exclamation	Separate the Type_mark in a declaration.	
'	Single quote	Encloses manifest constants.	
"	Double quote	Encloses manifest strings.	
%	Percent	Introduces special character codes.	
/	Slash	In a special character code, introduces a character through its code.	
+ -	Plus and minus	Signs of integer and real constants. (Also permitted as prefix and infix operators, appearing in a separate table.)	
\$	Dollar	Address operator for passing the address of an Eiffel feature or expression to a routine (usually external).	
%	Percent	Introduces a special character code.	
/	Slash	In a special character, introduces a character by its numerical code.	
.	Dot	Separates target from feature in a feature call or creation call. Separates integer from fractional part in a real number.	
->	Arrow	Introduces the constraint of a constrained formal generic parameter.	
:=	Receives	Assignment operator.	
= /=	Equal, not-equal signs	Equality and non-equality operators.	
~ /~	Tilde, slash-tilde	Object equality and non-equality operators.	
()	Parentheses	Group subexpressions in operator expressions; enclose formal and actual arguments of routines.	
()	Target parentheses	Enclose a constant or non-atomic expression used as target of a call in dot or bracked notation.	
[]	Brackets	Enclose formal and actual generic parameters to classes; enclose items of a manifest tuple; specify that a feature has a Bracket alias.	
{ }	Braces	Enclose types in various contexts: Clients part, Feature_clause or New_export_list , Creation_type .	

L.4 OPERATORS AND THEIR PRECEDENCE

Operator precedence levels	
13	. (Dot notation, in <u>qualified</u> and non-object calls)
12	old (In postconditions) not + - Used as unary All free unary operators
11	All free binary operators.
10	^ (Used as binary: power)
9	* / // \ (As binary: multiplicative arithmetic operators)
8	+ - Used as binary
7	.. (To define an interval)
6	= /= ~ /~ < > <= >= (As binary: relational operators)
5	and and then (Conjunctive boolean operators)
4	or or else xor (Disjunctive boolean operators)
3	implies (Implicative boolean operator)
2	[] (Manifest tuple delimiter)
1	; (Optional semicolon between an <u>Assertion_clause</u> and the next)

L.5 KEYWORDS AND SYMBOLS OF SPECIAL INTERFACE SUBLANGUAGES

Here are the keywords used in the special interface sublanguages. These are not Eiffel keywords, but special words that may appear in strings denoting external languages and their special mechanisms.

C	C++	data_member	delete
Fortran95	include	inline	Java
macro	new	static	struct

The following symbols may appear in such strings:

Symbol	Name	Role
:	Colon	Introduces the result type in a function signature.
()	Parentheses	Enclose argument types in a function signature.
"	Double quote	Encloses a file name (may have to be written "%" as part of a manifest string).
\$	Dollar	Introduces an Eiffel entity in an inline C text.

< >	Angle brackets	Enclose the name of a system include file.
[]	Square brackets	Enclose macro and DLL specifications.

M

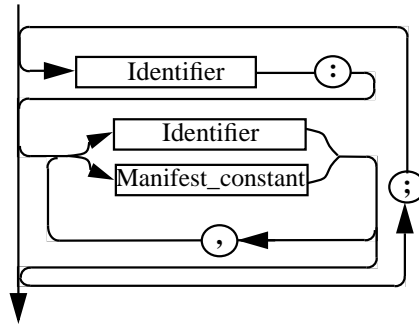
Draft 5.10, 25 August 2006 (Santa Barbara).

Extracted from ongoing work on future third edition of "Eiffel: The Language".
Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or
second printing (Prentice Hall, 1991). Do not reproduce or distribute.

Syntax diagrams (not done)

Copyright © Bertrand Meyer, 1986-2005. Draft 5.10, 25 August 2006 (Santa Barbara).

Notes



Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

PART VII: THE LANGUAGE STANDARD

This last part of the book contains the draft ECMA Eiffel standard.

It contains no new material but only extracts from part **II**. The elements retained are:

Standard ECMA-367-3 (Draft)

Revision 367-3-002, 25 August 2006

Eiffel Analysis, Design and Programming Language

Standard

List of changes

25 August 2006: Fixed call-agent-equivalent definition, 8.27.12

25 August 2006: Integrated still incomplete changes of clause 10 with notion of transposition.

Standard
ECMA-367

1st Edition - June 2005

**Eiffel Analysis, Design and
Programming Language**

Brief history

Eiffel was originally designed, as a method of software construction and a notation to support that method, in 1985. The first implementation, from Eiffel Software (then Interactive Software Engineering Inc.), was commercially released in 1986. The principal designer of the first versions of the language was Bertrand Meyer. Other people closely involved with the original definition included Jean-Marc Nerson. The language was originally described in Eiffel Software technical documents that were expanded to yield Meyer's book *Eiffel: The Language* in 1990-1991. The two editions of *Object-Oriented Software Construction* (1988 and 1997) also served to describe the concepts. (For bibliographical references on the documents cited see 3.6.) As usage of Eiffel grew, other Eiffel implementations appeared, including Eiffel/S and Visual Eiffel from Object Tools, Germany, EiffelStudio and Eiffel Envision from Eiffel Software, and SmartEiffel from LORIA, France.

Eiffel today is used throughout the world for industrial applications in banking and finance, defense and aerospace, health care, networking and telecommunications, computer-aided design, game programming, and many other application areas. Eiffel is particularly suited for mission-critical developments in which programmer productivity and product quality are essential. In addition Eiffel is a popular medium for teaching programming and software engineering in universities.

In 2002 Ecma International formed Technical Group 4 (Eiffel) of Technical Committee 39 (Programming and Scripting Languages). The Eiffel Analysis, Design and Programming Language Standard provides a precise definition of the language and ensures interoperability between implementations. The first of these benefits is of particular interest to implementors of Eiffel compilers and environments, who can rely on it as the reference on which to base their work; the second, to Eiffel users, for whom the Standard delivers a guarantee of compatibility between the products of different providers and of trust in the future of Eiffel.

TG4 devised this Standard from June 2002 to April 2005, starting from material from the original and revised versions of the book *Standard Eiffel* (latest revision of *Eiffel: The Language*). During that period the Technical Group conducted fifteen face-to-face meetings and numerous phone meetings, in addition to extensive technical correspondence. The members of the committee have been: Karine Arnout (ETH, Zurich); Éric Bezault (Axa Rosenberg, Orinda); Paul Cohen (Generic, Stockholm), Dominique Colnet (LORIA, Nancy); Mark Howard (Axa Rosenberg, Orinda); Alexander Kogtenkov (Eiffel Software, Moscow); Bertrand Meyer (Eiffel Software, Santa Barbara, and ETH, Zurich); Christine Mingins (Monash University, Melbourne); Roger Osmond (EMC, Boston); Emmanuel Stapf (Eiffel Software, Santa Barbara); Kim Waldén (Generic, Stockholm).

Observers having attended one or more of the meetings include Cyril Adrian (LORIA), Volkan Arslan (ETH), Paul Crismer (Groupe S, Brussels), Jocelyn Fiat (Eiffel Software, France), Randy John (Axa Rosenberg), Ian King (Eiffel Software), Philippe Ribet (LORIA), Julian Rogers (Eiffel Software), Bernd Schoeller (ETH), David Schwartz (Axa Rosenberg), Zoran Simic (Axa Rosenberg), Raphael Simon (Eiffel Software), Olivier Zendra (LORIA). The committee acknowledges the contributions of many people including David Hollenberg, Marcel Satchell, Richard O'Keefe and numerous others listed in the acknowledgments of the book *Standard Eiffel*.

The editor of the standard is Bertrand Meyer. Emmanuel Stapf is the convener of TG4 (succeeding Christine Mingins, 2002-2003) and its secretary (succeeding Karine Arnout, 2002-2004).

The final version of the document was prepared by Éric Bezault, Mark Howard, Alexander Kogtenkov, Bertrand Meyer and Emmanuel Stapf.

Table of contents

1 Scope	1
1.1 Overview	1
1.2 "The Standard"	1
1.3 Aspects covered	1
1.4 Aspects not covered	1
2 Conformance	1
2.1 Definition	1
2.2 Compatibility and non-default options	2
2.3 Departure from the Standard	2
3 Normative references	2
3.1 Earlier Eiffel language specifications	2
3.2 Eiffel Kernel Library	2
3.3 Floating point number representation	3
3.4 Character set: Unicode	3
3.5 Character set: ASCII	3
3.6 Phonetic alphabet	3
4 Definitions	3
5 Notational conventions	3
5.1 Standard elements	3
5.2 Normative elements	3
5.3 Rules on definitions	4
5.4 Use of defined terms	4
5.5 Unfolded forms	4
5.6 Language description	5
5.7 Validity: "if and only if" rules	5
6 Acronyms and abbreviations	5
6.1 Name of the language	5
6.2 Pronunciation	5
7 General description	5
7.1 Design principles	5
7.2 Object-oriented design	6
7.3 Classes	7
7.4 Types	10

7.5 Assertions	11
7.6 Exceptions	14
7.7 Genericity	15
7.8 Inheritance	16
7.9 Polymorphism and dynamic binding	18
7.10 Combining genericity and inheritance	20
7.11 Deferred classes	21
7.12 Tuples and agents	22
7.13 Type- and void-safety	23
7.14 Putting a system together	24
8 Language specification	24
8.1 General organization	24
8.2 Syntax, validity and semantics	25
8.2.1 Definition: Syntax, BNF-E	25
8.2.2 Definition: Component, construct, specimen	25
8.2.3 Construct Specimen convention	25
8.2.4 Construct Name convention	25
8.2.5 Definition: Terminal, non-terminal, token	25
8.2.6 Definition: Production	26
8.2.7 Kinds of production	26
8.2.8 Definition: Aggregate production	26
8.2.9 Definition: Choice production	26
8.2.10 Definition: Repetition production, separator	26
8.2.11 Basic syntax description rule	27
8.2.12 Definition: Non-production syntax rule	27
8.2.13 Textual conventions	27
8.2.14 Definition: Validity constraint	28
8.2.15 Definition: Valid	28
8.2.16 Validity: General Validity rule	28
8.2.17 Definition: Semantics	28
8.2.18 Definition: Execution terminology	28
8.2.19 Semantics: Case Insensitivity principle	29
8.2.20 Definition: Upper name, lower name	29
8.2.21 Syntax (non-production): Semicolon Optionality rule	29
8.3 The architecture of Eiffel software	30
8.3.1 Definition: Cluster, subcluster, contains directly, contains	30
8.3.2 Definition: Terminal cluster, internal cluster	30
8.3.3 Definition: Universe	31
8.3.4 Validity: Class Name rule	31
8.3.5 Semantics: Class name semantics	31
8.3.6 Definition: System, root type name, root procedure name	31
8.3.7 Definition: Type dependency	31
8.3.8 Validity: Root Type rule	32

8.3.9 Validity: Root Procedure rule	32
8.3.10 Definition: Root type, root procedure, root class	32
8.3.11 Semantics: System execution	33
8.4 Classes	33
8.4.1 Definition: Current class	33
8.4.2 Syntax : Class declarations	33
8.4.3 Syntax : Notes	33
8.4.4 Semantics: Notes semantics	34
8.4.5 Syntax : Class headers	34
8.4.6 Definition: Expanded, frozen, deferred, effective class	34
8.4.7 Validity: Class Header rule	34
8.4.8 Syntax : Obsolete marks	35
8.4.9 Semantics: Obsolete semantics	35
8.5 Features	35
8.5.1 Definition: Inherited, immediate; origin; redeclaration; introduce	35
8.5.2 Syntax : Feature parts	36
8.5.3 Feature categories: overview	36
8.5.4 Syntax : Feature declarations	36
8.5.5 Syntax : New feature lists	36
8.5.6 Syntax : Feature bodies	37
8.5.7 Validity: Feature Body rule	37
8.5.8 Definition: Variable attribute	37
8.5.9 Definition: Constant attribute	38
8.5.10 Definition: Routine, function, procedure	38
8.5.11 Definition: Command, query	38
8.5.12 Definition: Signature, argument signature of a feature	38
8.5.13 Feature principle	38
8.5.14 Syntax : Feature names	39
8.5.15 Syntax (non-production): Alias Syntax rule	39
8.5.16 Definition: Operator feature, bracket feature, identifier-only	39
8.5.17 Definition: Identifier of a feature name	40
8.5.19 Definition: Same feature name, same operator, same alias	40
8.5.20 Syntax : Operators	40
8.5.21 Syntax : Assigner marks	41
8.5.22 Validity: Assigner Command rule	41
8.5.23 Definition: Synonym	42
8.5.24 Definition: Unfolded form of a possibly multiple declaration	42
8.5.25 Validity: Feature Declaration rule	42
8.5.26 Validity: Alias Validity rule	43
8.6 The inheritance relation	44
8.6.1 Syntax : Inheritance parts	44
8.6.2 Syntax (non-production): Feature adaptation	44
8.6.3 Definition: Parent part for a type, for a class	44

8.6.4 Definition: Multiple, single inheritance	45
8.6.5 Definition: Inherit, heir, parent	45
8.6.6 Definition: Conforming, non-conforming parent	45
8.6.7 Definition: Ancestor types of a type, of a class	45
8.6.8 Definition: Ancestor, descendant	46
8.6.9 Definition: Proper ancestor, proper descendant	46
8.6.10 Validity: Class ANY rule	46
8.6.11 Validity: Universal Conformance principle	46
8.6.12 Definition: Unfolded Inheritance Part of a class	46
8.6.13 Validity: Parent rule	46
8.6.14 Syntax : Rename clauses	47
8.6.15 Validity: Rename Clause rule	47
8.6.16 Semantics: Renaming principle	48
8.6.17 Definition: Final name, extended final name, final name set	48
8.6.18 Definition: Inherited name	48
8.6.19 Definition: Declaration for a feature	48
8.7 Clients and exports	49
8.7.1 Definition: Client relation between classes and types	49
8.7.2 Definition: Client relation between classes	49
8.7.3 Definition: Indirect client	49
8.7.4 Definition: Supplier	49
8.7.5 Definition: Simple client	49
8.7.6 Definition: Expanded client	50
8.7.7 Definition: Generic client, generic supplier	50
8.7.8 Definition: Client set of a Clients part	50
8.7.9 Syntax : Clients	51
8.7.10 Syntax : Export adaptation	51
8.7.11 Validity: Export List rule	51
8.7.12 Definition: Client set of a feature	51
8.7.13 Definition: Available for call, available	52
8.7.14 Definition: Exported, selectively available, secret	52
8.7.15 Definition: Secret, public	52
8.7.16 Definition: Incremental contract view, short form	53
8.7.17 Definition: Contract view, flat-short form	53
8.8 Routines	54
8.8.1 Definition: Formal argument, actual argument	54
8.8.2 Syntax : Formal argument and entity declarations	54
8.8.3 Validity: Formal Argument rule	55
8.8.4 Validity: Entity Declaration rule	55
8.8.5 Syntax : Routine bodies	55
8.8.6 Definition: Once routine, once procedure, once function	55
8.8.7 Syntax : Local variable declarations	55
8.8.8 Validity: Local Variable rule	55
8.8.9 Definition: Local variable	56

8.8.10 Syntax : Instructions	56
8.9 Correctness and contracts	57
8.9.1 Syntax : Assertions	57
8.9.2 Syntax (non-production): Assertion Syntax rule	57
8.9.3 Definition: Precondition, postcondition, invariant	57
8.9.4 Definition: Contract, subcontract	58
8.9.5 Validity: Precondition Export rule	58
8.9.6 Definition: Availability of an assertion clause	58
8.9.7 Syntax : “Old” postcondition expressions	58
8.9.8 Validity: Old Expression rule	59
8.9.9 Semantics: Old Expression Semantics, associated variable, associated exception marker	59
8.9.10 Semantics: Associated Variable Semantics	60
8.9.11 Syntax : “Only” postcondition clauses	60
8.9.12 Definition: Unfolded feature list of an Only clause	60
8.9.13 Validity: Only Clause rule	61
8.9.14 Definition: Unfolded form of an Only clause	61
8.9.15 Definition: Hoare triple notation (total correctness)	61
8.9.16 Semantics: Class consistency	61
8.9.17 Syntax : Check instructions	62
8.9.18 Definition: Check-correct	62
8.9.19 Syntax : Variants	62
8.9.20 Validity: Variant Expression rule	62
8.9.21 Definition: Loop invariant and variant	62
8.9.22 Definition: Loop-correct	62
8.9.23 Definition: Correctness (class)	62
8.9.24 Definition: Local unfolded form of an assertion	62
8.9.25 Semantics: Assertion monitoring	63
8.9.26 Semantics: Evaluation of an assertion	63
8.9.27 Semantics: Assertion violation	63
8.9.28 Semantics: Assertion semantics	64
8.9.29 Semantics: Assertion monitoring levels	64
8.10 Feature adaptation	64
8.10.1 Definition: Redeclare, redeclaration	64
8.10.2 Definition: Unfolded form of an assertion	64
8.10.3 Definition: Assertion extensions	65
8.10.4 Definition: Covariance-aware form of an assertion extension	65
8.10.5 Definition: Combined precondition, postcondition	65
8.10.6 Definition: Inherited as effective, inherited as deferred	66
8.10.7 Definition: Effect, effecting	66
8.10.8 Definition: Redefine, redefinition	66
8.10.9 Definition: Name clash	67
8.10.10 Syntax : Precursor	67
8.10.11 Definition: Relative unfolded form of a Precursor	67

8.10.12	Validity: Precursor rule	67
8.10.13	Definition: Unfolded form of a Precursor	68
8.10.14	Semantics: Precursor semantics	68
8.10.15	Syntax : Redefinition	68
8.10.16	Validity: Redefine Subclause rule	68
8.10.17	Semantics: Redefinition semantics	68
8.10.18	Syntax : Undefine clauses	68
8.10.19	Validity: Undefine Subclause rule	68
8.10.20	Semantics: Undefinition semantics	69
8.10.21	Definition: Effective, deferred feature	69
8.10.22	Definition: Effecting	69
8.10.23	Deferred class property	69
8.10.24	Effective class property	69
8.10.25	Definition: Origin, seed	69
8.10.26	Validity: Redeclaration rule	70
8.10.27	Definition: Precursor (joined features)	71
8.10.28	Validity: Join rule	71
8.10.29	Semantics: Join Semantics rule	72
8.11	Types	72
8.11.1	Syntax : Types	72
8.11.2	Semantics: Direct instances and values of a type	73
8.11.3	Semantics: Instance of a type	73
8.11.4	Semantics: Instance principle	73
8.11.5	Definition: Instance, direct instance of a class	73
8.11.6	Base principle	73
8.11.7	Base rule	74
8.11.8	Validity: Class Type rule	74
8.11.9	Semantics: Type Semantics rule	74
8.11.10	Definition: Base class and base type of an expression	74
8.11.11	Semantics: Non-generic class type semantics	74
8.11.12	Definition: Expanded type, reference type	74
8.11.13	Definition: Basic type	75
8.11.14	Definition: Anchor, anchored type, anchored entity	75
8.11.15	Definition: Anchor set; cyclic anchor	75
8.11.16	Definition: Types and classes involved in a type	76
8.11.17	Definition: Constant type	76
8.11.18	Definition: Deanchored form of a type	76
8.11.19	Validity: Anchored Type rule	77
8.11.20	Definition: Attached, detachable	77
8.11.21	Semantics: Attached type semantics	77
8.11.22	Definition: Stand-alone type	78
8.12	Genericity	78
8.12.1	Syntax : Actual generic parameters	78
8.12.2	Syntax : Formal generic parameters	78

8.12.3	Validity: Formal Generic rule	78
8.12.4	Definition: Generic class; constrained, unconstrained	79
8.12.5	Definition: Generic derivation, non-generic type	79
8.12.6	Definition: Self-initializing formal	79
8.12.7	Definition: Constraint, constraining types of a Formal_generic	79
8.12.8	Syntax : Generic constraints	80
8.12.9	Validity: Generic Constraint rule	80
8.12.10	Definition: Constraining creation features	80
8.12.11	Validity: Generic Derivation rule	80
8.12.12	Definition: Generic-creation-ready type	81
8.12.13	Semantics: Generically derived class type semantics	81
8.12.14	Definition: Base type of a single-constrained formal generic	82
8.12.15	Definition: Base type of an unconstrained formal generic	82
8.12.16	Definition: Reference or expanded status of a formal generic	82
8.12.17	Definition: Current type	82
8.12.18	Definition: Features of a type	82
8.12.19	Definition: Generic substitution	82
8.12.20	Generic Type Adaptation rule	82
8.12.21	Definition: Generically constrained feature name	83
8.12.22	Validity: Multiple Constraints rule	83
8.12.23	Definition: Base type of a multi-constraint formal generic type	83
8.13	Tuples	83
8.13.1	Syntax : Tuple types	83
8.13.2	Syntax : Manifest tuples	84
8.13.3	Definition: Type sequence of a tuple type	84
8.13.4	Definition: Value sequences associated with a tuple type	84
8.14	Conformance	84
8.14.1	Definition: Compatibility between types	85
8.14.2	Definition: Compatibility between expressions	85
8.14.3	Definition: Expression conformance	85
8.14.4	Validity: Signature conformance	85
8.14.5	Definition: Covariant argument	86
8.14.6	Validity: General conformance	86
8.14.7	Definition: Conformance path	87
8.14.8	Validity: Direct conformance: reference types	87
8.14.9	Validity: Direct conformance: formal generic	87
8.14.10	Validity: Direct conformance: expanded types	87
8.14.11	Validity: Direct conformance: tuple types	88
8.15	Convertibility	88
8.15.1	Definition: Conversion procedure, conversion type	88
8.15.2	Definition: Conversion query, conversion feature	88
8.15.6	Syntax : Converter clauses	88
8.15.7	Validity: Conversion Procedure rule	89

8.15.8 Validity: Conversion Query rule	89
8.15.9 Definition: Converting to a class	89
8.15.10 Definition: Converting to and from a type	90
8.15.11 Definition: Converting “through”	90
8.15.12 Semantics: Conversion semantics	90
8.15.13 Definition: Explicit conversion	90
8.15.14 Validity: Expression convertibility	91
8.15.15 Definition: Statically satisfied precondition	91
8.15.16 Validity: Precondition-free routine	91
8.16 Repeated inheritance	92
8.16.1 Definition: Repeated inheritance, ancestor, descendant	92
8.16.2 Semantics: Repeated Inheritance rule	93
8.16.3 Definition: Sharing, replication	93
8.16.4 Validity: Call Replication rule	93
8.16.5 Semantics: Replication Semantics rule	93
8.16.6 Syntax : Select clauses	93
8.16.7 Validity: Select Subclause rule	94
8.16.8 Definition: Version	94
8.16.9 Definition: Multiple versions	94
8.16.10 Validity: Repeated Inheritance Consistency constraint	94
8.16.11 Definition: Dynamic binding version	94
8.16.12 Definition: Inherited features	94
8.16.13 Semantics: Join-Sharing Reconciliation rule	95
8.16.14 Definition: Precursor	95
8.16.15 Validity: Feature Name rule	95
8.16.16 Validity: Name Clash rule	95
8.17 Control structures	96
8.17.1 Semantics: Compound (non-exception) semantics	96
8.17.2 Syntax : Conditionals	96
8.17.3 Definition: Secondary part	97
8.17.4 Definition: Prevailing immediately	97
8.17.5 Semantics: Conditional semantics	97
8.17.6 Definition: Inspect expression	97
8.17.7 Syntax : Multi-branch instructions	97
8.17.8 Definition: Interval	97
8.17.9 Definition: Unfolded form of a multi-branch	97
8.17.10 Definition: Unfolded form of an interval	98
8.17.11 Validity: Interval rule	98
8.17.12 Definition: Inspect values of a multi-branch	98
8.17.13 Validity: Multi-branch rule	98
8.17.14 Semantics: Matching branch	99
8.17.15 Semantics: Multi-Branch semantics	99
8.17.16 Syntax : Loops	99
8.17.17 Semantics: Loop semantics	99

8.17.18 Syntax : Debug instructions	100
8.17.19 Semantics: Debug semantics	100
8.18 Attributes	100
8.18.1 Syntax : Attribute bodies	100
8.18.2 Validity: Manifest Constant rule	100
8.19 Objects, values and entities	101
8.19.1 Semantics: Type, generating type of an object; generator	101
8.19.2 Definition: Reference, void, attached, attached to	101
8.19.3 Semantics: Object principle	101
8.19.4 Definition: Object semantics	101
8.19.5 Definition: Non-basic class, non-basic type, field	102
8.19.6 Definition: Subobject, composite object	102
8.19.7 Definition: Entity, variable, read-only	102
8.19.8 Syntax : Entities and variables	102
8.19.9 Validity: Entity rule	102
8.19.10 Validity: Variable rule	103
8.19.11 Definition: Self-initializing type	103
8.19.12 Semantics: Default Initialization rule	103
8.19.13 Definition: Self-initializing variable	104
8.19.14 Definition: Evaluation position, precedes	104
8.19.15 Definition: Setter instruction	105
8.19.16 Definition: Properly set variable	105
8.19.17 Validity: Variable Initialization rule	105
8.19.18 Definition: Variable setting and its value	106
8.19.19 Definition: Execution context	106
8.19.20 Semantics: Variable Semantics	106
8.19.21 Semantics: Entity Semantics rule	107
8.20 Creating objects	107
8.20.1 Semantics: Creation principle	107
8.20.2 Definition: Creation operation	108
8.20.3 Validity: Creation Precondition rule	108
8.20.4 Syntax : Creators parts	108
8.20.5 Definition: Unfolded Creators part of a class	108
8.20.6 Validity: Creation Clause rule	109
8.20.7 Definition: Creation procedures of a class	109
8.20.8 Creation procedure property	109
8.20.9 Definition: Creation procedures of a type	109
8.20.10 Definition: Available for creation; general creation procedure	110
8.20.11 Syntax : Creation instructions	110
8.20.12 Definition: Creation target, creation type	110
8.20.13 Semantics: Creation Type theorem	110
8.20.14 Definition: Unfolded form of a creation instruction	110
8.20.15 Validity: Creation Instruction rule	110

8.20.16	Validity: Creation Instruction properties	111
8.20.17	Semantics: Creation Instruction Semantics	112
8.20.18	Syntax : Creation expressions	112
8.20.19	Definition: Properties of a creation expression	112
8.20.20	Validity: Creation Expression rule	113
8.20.21	Validity: Creation Expression Properties	113
8.20.22	Semantics: Creation Expression Semantics	113
8.21	Comparing and duplicating objects	114
8.21.1	Object comparison features from <i>ANY</i>	114
8.21.2	Syntax : Equality expressions	115
8.21.3	Semantics: Equality Expression Semantics	115
8.21.4	Semantics: Inequality Expression Semantics	115
8.21.5	Copying and cloning features from <i>ANY</i>	115
8.21.6	Deep equality, copying and cloning	116
8.22	Attaching values to entities	116
8.22.1	Definition: Reattachment, source, target	117
8.22.2	Syntax : Assignments	117
8.22.3	Validity: Assignment rule	117
8.22.4	Semantics: Reattachment principle	117
8.22.5	Semantics: Attaching an entity, attached entity	117
8.22.6	Semantics: Reattachment Semantics	118
8.22.7	Semantics: Assignment Semantics	118
8.22.8	Definition: Dynamic type	119
8.22.9	Definition: Polymorphic expression; dynamic type and class sets	119
8.22.10	Syntax : Assigner calls	119
8.22.11	Validity: Assigner Call rule	119
8.22.12	Semantics: Assigner Call semantics	119
8.23	Feature call	119
8.23.1	Validity: Call Use rule	120
8.23.2	Syntax : Feature calls	120
8.23.3	Syntax : Actual arguments	120
8.23.4	Definition: Unqualified, qualified call	120
8.23.5	Definition: Target of a call	120
8.23.6	Definition: Target type of a call	121
8.23.7	Definition: Feature of a call	121
8.23.8	Definition: Imported form of a <i>Non_object_call</i>	121
8.23.9	Validity: Non-Object Call rule	121
8.23.10	Semantics: Non-Object Call Semantics	122
8.23.11	Validity: Export rule	122
8.23.13	Validity: Argument rule	122
8.23.14	Validity: Target rule	123
8.23.15	Validity: Class-Level Call rule	123
8.23.16	Definition: Void-Unsafe	123

8.23.17 Definition: Target Object	123
8.23.18 Semantics: Failed target evaluation of a void-unsafe system	123
8.23.19 Definition: Dynamic feature of a call	123
8.23.20 Definition: Freshness of a once routine call	124
8.23.21 Definition: Latest applicable target of a non-fresh call	124
8.23.22 Semantics: Once Routine Execution Semantics	124
8.23.23 Semantics: Current object, current routine	125
8.23.24 Semantics: Current Semantics	125
8.23.25 Semantics: Non-Once Routine Execution Semantics	125
8.23.26 Semantics: General Call Semantics	125
8.23.27 Definition: Type of a Call used as expression	126
8.23.28 Semantics: Call Result	126
8.23.29 Semantics: Value of a call expression	126
8.24 Eradicating void calls	126
8.24.1 Syntax : Object test	127
8.24.2 Definition: Object-Test Local	127
8.24.3 Validity: Object Test rule	127
8.24.4 Definition: Conjunctive, disjunctive, implicative;Term, semistrict term	127
8.24.5 Definition: Scope of an Object-Test Local	127
8.24.6 Semantics: Object Test semantics	128
8.24.7 Semantics: Object-Test Local semantics	128
8.24.8 Definition: Read-only void test	128
8.24.9 Definition: Scope of a read-only void test	128
8.24.10 Definition: Certified Attachment Pattern	129
8.24.11 Definition: Attached expression	129
8.25 Typing-related properties	129
8.25.1 Definition: Catcall	129
8.25.2 Validity: Descendant Argument rule	130
8.25.3 Validity: Single-level Call rule	130
8.25.4 Definition: System-valid, valid	130
8.25.5 Definition: Dynamic type set	130
8.26 Exception handling	130
8.26.1 Definition: Failure, exception, trigger	130
8.26.2 Syntax : Rescue clauses	131
8.26.3 Validity: Rescue clause rule	131
8.26.4 Validity: Retry rule	131
8.26.5 Definition: Exception-correct	131
8.26.6 Semantics: Default Rescue Original Semantics	131
8.26.7 Definition: Rescue block	131
8.26.8 Semantics: Exception Semantics	132
8.26.9 Definition: Type of an exception	132
8.26.10 Semantics: Exception Cases	133

8.26.11 Semantics: Exception Properties	133
8.26.12 Definition: Ignoring, continuing an exception	134
8.27 Agents, iteration and introspection	134
8.27.1 Definition: Operands of a call	134
8.27.2 Definition: Operand position	134
8.27.3 Definition: Construction time, call time	134
8.27.4 Syntactical forms for a call agent	134
8.27.5 Syntax : Agents	135
8.27.6 Syntax : Call agent bodies	135
8.27.7 Definition: Target type of an agent call	135
8.27.8 Validity: Call Agent rule	135
8.27.9 Definition: Associated feature of an inline agent	135
8.27.10 Validity: Inline Agent rule	136
8.27.11 Validity: Inline Agent Requirements	136
8.27.12 Definition: Call-agent equivalent of an inline agent	136
8.27.13 Semantics: Semantics of inline agents	136
8.27.14 Semantics: Use of Result in an inline function agent	136
8.27.15 Definition: Open and closed operands	136
8.27.16 Definition: Open and closed operand positions	136
8.27.17 Definition: Type of an agent expression	136
8.27.18 Semantics: Agent Expression semantics	137
8.27.19 Semantics: Effect of executing <i>call</i> on an agent	137
8.28 Expressions	137
8.28.1 Syntax : Expressions	137
8.28.2 Definition: Subexpression, operand	137
8.28.3 Semantics: Parenthesized Expression Semantics	138
8.28.4 Syntax : Operator expressions	138
8.28.5 Operator precedence levels	138
8.28.6 Definition: Parenthesized Form of an expression	138
8.28.7 Definition: Target-converted form of a binary expression	139
8.28.8 Validity: Operator Expression rule	139
8.28.9 Semantics: Expression Semantics (strict case)	139
8.28.10 Definition: Semistrict operators	140
8.28.11 Semantics: Operator Expression Semantics (semistrict cases)	140
8.28.12 Syntax : Bracket expressions	140
8.28.13 Validity: Bracket Expression rule	140
8.28.14 Definition: Equivalent Dot Form of an expression	140
8.28.15 Validity: Boolean Expression rule	141
8.28.16 Validity: Identifier rule	141
8.28.17 Definition: Type of an expression	141
8.29 Constants	142
8.29.1 Syntax : Constants	142
8.29.2 Validity: Constant Attribute rule	142

8.29.3 Syntax : Manifest constants	142
8.29.4 Syntax (non-production): Sign Syntax rule	142
8.29.5 Syntax (non-production): Character Syntax rule	142
8.29.6 Definition: Type of a manifest constant	143
8.29.7 Validity: Manifest-Type Qualifier rule	143
8.29.8 Semantics: Manifest Constant Semantics	143
8.29.9 Definition: Manifest value of a constant	144
8.29.10 Syntax : Manifest strings	144
8.29.11 Syntax (non-production): Line sequence	145
8.29.12 Syntax (non-production): Manifest String rule	145
8.29.13 Definition: Line_wrapping_part	145
8.29.14 Semantics: Manifest string semantics	145
8.29.15 Validity: Verbatim String rule	145
8.29.16 Semantics: Verbatim string semantics	146
8.29.17 Definition: Prefix, longest break prefix, left-aligned form	146
8.30 Basic types	146
8.30.1 Definition: Basic types and their sized variants	146
8.30.2 Definition: Sized variants of STRING	146
8.30.3 Semantics: Boolean value semantics	147
8.30.4 Semantics: Character types	147
8.30.5 Semantics: Integer types	147
8.30.6 Semantics: Floating-point types	147
8.30.7 Semantics: Address semantics	147
8.31 Interfacing with C, C++ and other environments	147
8.31.1 Syntax : External routines	147
8.31.2 Validity: Address rule	148
8.31.3 Address Type rule	148
8.31.4 Semantics: Address semantics	148
8.31.5 Syntax : Registered languages	149
8.31.6 Syntax : External signatures	149
8.31.7 Validity: External Signature rule	149
8.31.8 Semantics: External signature semantics	149
8.31.9 Syntax : External file use	149
8.31.10 Validity: External File rule	150
8.31.11 Semantics: External file semantics	150
8.31.12 Syntax : C externals	150
8.31.13 Validity: C external rule	151
8.31.14 Semantics: C Inline semantics	151
8.31.15 Syntax : C++ externals	151
8.31.16 Validity: C++ external rule	152
8.31.17 Semantics: C++ Inline semantics	152
8.31.18 Syntax : DLL externals	152
8.31.19 Validity: External DLL rule	152
8.31.20 Semantics: External DLL semantics	152

8.32 Lexical components	152
8.32.1 Syntax (non-production): Character, character set	153
8.32.2 Definition: Letter, alpha_betic, numeric, alpha_numeric, printable	153
8.32.3 Definition: Break character, break	153
8.32.4 Semantics: Break semantics	154
8.32.5 Definition: Expected, free comment	154
8.32.6 Syntax (non-production): "Blanks or tabs", new line	154
8.32.7 Syntax : Comments	154
8.32.8 Syntax (non-production): Free Comment rule	154
8.32.9 Header comment rule	154
8.32.10 Definition: Symbol, word	155
8.32.11 Syntax (non-production): Break rule	155
8.32.12 Semantics: Letter Case rule	155
8.32.13 Definition: Reserved word, keyword	155
8.32.14 Syntax (non-production): Double Reserved Word rule	156
8.32.15 Definition: Special symbol	156
8.32.16 Syntax (non-production): Identifier	156
8.32.17 Validity: Identifier rule	156
8.32.18 Definition: Predefined operator	156
8.32.19 Definition: Standard operator	156
8.32.20 Definition: Operator symbol	156
8.32.21 Definition: Free operator	157
8.32.22 Syntax (non-production): Manifest character	157
8.32.23 Special characters and their codes	158
8.32.24 Syntax (non-production): Percent variants	159
8.32.25 Semantics: Manifest character semantics	159
8.32.26 Syntax (non-production): String, simple string	159
8.32.27 Semantics: String semantics	159
8.32.28 Syntax : Integers	159
8.32.29 Validity: Integer rule	159
8.32.30 Semantics: Integer semantics	160
8.32.31 Syntax (non-production): Real number	160
8.32.32 Semantics: Real semantics	160

1 Scope

1.1 Overview

This document provides the full reference for the Eiffel language.

Eiffel is a method of software construction and a language applicable to the analysis, design, implementation and maintenance of software systems. This Standard covers only the language, with an emphasis on the implementation aspects. As a consequence, the word “Eiffel” in the rest of this document is an abbreviation for “the Eiffel language”.

1.2 “The Standard”

The language definition proper — “the Standard” — is contained in Partition 8 of this document, with the exception of text appearing between markers *Informative text* and *End*; such text only plays an explanatory role for human readers.

1.3 Aspects covered

The Standard specifies:

- The form of legal basic constituents of Eiffel texts, or *lexical* properties of the language.
- The structure of legal Eiffel texts made of lexically legal constituents, or *syntax* properties.
- Supplementary restrictions imposed on syntactically legal Eiffel texts, or *validity* properties.
- The computational effect of executing valid Eiffel texts, or *semantic* properties.
- Some requirements on a conforming implementation of Eiffel, such as the ability to produce certain forms of automatic documentation.

1.4 Aspects not covered

The Standard does not specify:

- The requirements that a computing environment must meet to support the translation, execution and other handling of Eiffel texts.
- The semantic properties of an Eiffel text if it or its data exceed the capacity of a particular computing environment.
- The mechanisms for translating Eiffel texts into a form that can be executed in a computing environment.
- The mechanisms for starting the execution of the result of such a translation.
- Other mechanisms for handling Eiffel texts and interacting with users of the language.

The specification of Partition 8 consists of precise specification elements, originating with the book *Standard Eiffel* where these elements are accompanied by extensive explanations and examples. The elements retained are:

- Definitions of technical terms and Eiffel concepts.
- Syntax specifications.
- Validity constraints (with their codes, such as *VVBG*).
- Semantic specifications.

2 Conformance

2.1 Definition

An implementation of the Eiffel language is **conformant** if and only if in its default operating mode, when provided with a candidate software text, it:

- Can, if the text and all its elements satisfy the lexical, syntax and validity rules of the Standard, execute the software according to the semantic rules of the Standard, or generate code for a computing environment such that, according to the specification of that environment, the generated code represents the semantics of the text according to these rules.
- Will, if any element of the text violates any lexical, syntactical or validity rule of the Standard, report an error and perform no semantic processing (such as generating executable code, or directly attempting to execute the software).

2.2 Compatibility and non-default options

Implementations may provide options that depart in minor ways from the rules of this Standard, for example to provide compatibility with earlier versions of the implementation or of the language itself. Such options are permitted if and only if:

- Per 2.1, they are not the default.
- The implementation includes documentation that states that all such options are nonconformant.

2.3 Departure from the Standard

Material reasons, such as bugs or lack of time, may lead to the release of an implementation that supports most of the Standard but misses a few rules and hence is not yet conformant according to the definition of 2.1. In such a case the implementation shall include documentation that:

- States that the implementation is not conformant.
- Lists all the known causes of non-conformance.
- Provides an estimate of the date or version number for reaching full conformance.

3 Normative references

3.1 Earlier Eiffel language specifications

Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, second printing, 1992 (first printing: 1991).

Bertrand Meyer: *Standard Eiffel* (revision of preceding entry), ongoing, 1997-present, at <http://www.inf.ethz.ch/~meyer/ongoing/etl>.

Bertrand Meyer: *Object-Oriented Software Construction*, Prentice Hall: first edition, 1988; second edition, 1997.

3.2 Eiffel Kernel Library

The terms "ELKS" and "Kernel Library", as used in this Standard, refer to the latest version of the Eiffel Library Kernel Standard. A preliminary version is available from the NICE consortium:

NICE consortium: *The Eiffel Library Kernel Standard, 2001 Vintage*.

The Standard assumes that ELKS includes at least the following classes:

- Classes representing fundamental language-related concepts: *ANY*, *DISPOSABLE*, *NONE*, *TYPE*, *TYPED_POINTER*.
- Classes representing basic types and strings: *BOOLEAN*, *CHARACTER*, *CHARACTER_8*, *CHARACTER_32*, *INTEGER*, *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *NATURAL*, *NATURAL_8*, *NATURAL_16*, *NATURAL_32*, *NATURAL_64*, *POINTER*, *REAL*, *REAL_32*, *REAL_64*, *STRING*, *STRING_8*, *STRING_32*.
- Classes representing fundamental data structures: *ARRAY*, *TUPLE*.
- Agent-related classes: *FUNCTION*, *PREDICATE*, *PROCEDURE*, *ROUTINE*.

- Exception-related classes: [ASSERTION_VIOLATION](#), [ATTACHED_TARGET_VIOLATION](#), [EXCEPTION](#), [INSPECT_RANGE_VIOLATION](#), [INVARIANT_ENTRY_VIOLATION](#), [INVARIANT_EXIT_VIOLATION](#), [MEMORY_ALLOCATION_FAILURE](#), [OLD_VIOLATION](#), [POSTCONDITION_VIOLATION](#), [PRECONDITION_VIOLATION](#), [ROUTINE_FAILURE](#).

The clauses referring to these classes list the features they need.

3.3 Floating point number representation

IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989). Also known as ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*.

3.4 Character set: Unicode

The Unicode Consortium: *The Unicode Standard*, Version 4.1, at <http://www.unicode.org/versions/Unicode4.1.0/>.

3.5 Character set: ASCII

ISO 14962:1997: *Space data and information transfer systems*.

3.6 Phonetic alphabet

International Phonetic Association: *International Phonetic Alphabet* (revised 1993, updated 1996).

4 Definitions

All the Eiffel-specific terms used in the Standard are defined in paragraphs labeled “**Definition**”.

5 Notational conventions

5.1 Standard elements

Every clause of the Standard belongs to one of the following categories:

- **Syntax**: rule affecting the structure of Eiffel texts, including lexical properties as well as syntax proper. The conventions for describing syntax appear below.
- **Validity**: restrictions on syntactically legal texts.
- **Semantics**: properties of the execution of valid texts.
- **Definition**: introduction of a term defining a certain concept, which may relate to syntax, validity or semantic properties; the rest of the Standard may use the term as an abbreviation for the concept.
- **Principle**: a general language design rule, or a property of the software text resulting from other properties covered by definition and syntax, validity and semantic rules.

The clauses of the Standard are labeled with the corresponding category. A clause with no label shall be considered as “Definition”.

5.2 Normative elements

The rules of syntax, validity and semantics constitute the necessary and sufficient conditions for an implementation to be conformant.

The entries labeled **principle**, addressing validity in most cases and semantics in a few others, express properties that follow from the official rules. For example the Feature Identifier principle (8.5.18) states

“Given a class *C* and an identifier *f*, *C* contains at most one feature of identifier *f*.”

This property (expressing that there must be no overloading of feature identifiers within a class) follows from a variety of validity rules.

Such principles are conceptually redundant and do not add any constraint on an implementation; They provide an interesting angle on the language definition, both for:

- Programmers, by capturing a general property in a concise way.
- Implementers, who when reporting a rule violation in a candidate Eiffel text may choose to refer to the principle rather than the (possibly less immediately clear) rules that underlie it.

5.3 Rules on definitions

In a definition, the first occurrence of each term being defined appears in **bold**.

Some definitions are *recursive*, meaning that the definition of a certain property of certain language elements distinguishes between two or more cases, and in some (but not all) of these cases uses the same property applied to smaller language elements. (The definition then makes sense as all cases eventually reduce to basic, non-recursive cases.) For clarity, the recursive branches are always marked by the word “*recursively*” in parentheses.

In the text of the Standard, uses of a defined term may appear before the definition. The underlining convention described next helps avoid any confusion in such cases.

5.4 Use of defined terms

In any formal element of the language specification (definition, syntax, validity, semantics), appearance of a word as underlined means that this is a technical term introduced in one of the definitions of the Standard. For example, condition 1 in the Name Clash rule (8.16.16) reads:

“It is invalid for **C** to introduce two different features with the same name.”

The underlining indicates that words like “introduce” are not used in their plain English sense but as Eiffel-related concepts defined precisely elsewhere in the Standard. Indeed, 8.5.1 defines the notion that a class “introduces” a feature. Similarly, the notion of features having the “same name” is precisely defined (8.5.19) as meaning that their identifiers, excluding any **Operator** aliases, are identical *except* possibly for letter case.

This use of underlining is subject to the following restrictions:

- Underlining applies only to the first occurrence of such a defined term in a particular definition or rule.
- If a single clause defines a group of terms, occurrences of one of these terms in the definition of another are not underlined.
- As a consequence, uses of a term in its own definition are not underlined (they do bear, as noted, the mark “*recursively*”).
- In addition, a number of basic concepts, used throughout, are generally not underlined; they include:

<i>Assertion</i>	<i>Attribute</i>	<i>Call</i>	<i>Character</i>	<i>Class</i>
<i>Cluster</i>	<i>Entity</i>	<i>Expression</i>	<i>Feature</i>	<i>Identifier</i>
<i>Instruction</i>	<i>Semantics</i>	<i>Type</i>	<i>Valid</i>	

5.5 Unfolded forms

The definition of Eiffel in the Standard frequently relies on *unfolded forms* defining certain advanced constructs in terms of simpler ones. For example an “anchored type” of the form **like a** has a “deanchored form”, which is just the type of **a**. Validity and semantic rules can then rely on the unfolded form rather than the original.

This technique, applied to about twenty constructs of the language, makes the description significantly simpler by identifying a basic set of constructs for which it provides direct validity and semantic specifications, and defining the rest of the language in terms of this core.

5.6 Language description

The Standard follows precise rules and conventions for language description, described in 8.2.

5.7 Validity: “if and only if” rules

A distinctive property of the language description is that the validity constraints, for example type rules, do not just, as in many language descriptions, list properties that a software text *must* satisfy to be valid. They give the rules in an “if and only if” style. For example (8.8.3, Formal Argument rule):

“Let *fa* be the **Formal_arguments** part of a routine *r* in a class *C*. Let *formals* be the concatenation of every **Identifier_list** of every **Entity_declaration_group** in *fa*. Then *fa* is valid if and only if no **Identifier e** appearing in *formals* is the final_name of a feature of *C*.”

This does not just impose requirements on programmers, but tells them that if they satisfy these requirements they are *entitled* to having the construct (here a “**Formal_arguments** part”) accepted by an Eiffel language processing tool, for example a compiler. The rules, then, are not only necessary but sufficient. This contributes (except of course for the presence of any error or omission in the Standard) to the completeness of the language definition, and reinforces the programmers’ trust in the language.

6 Acronyms and abbreviations

6.1 Name of the language

The word “Eiffel” is not an acronym. It is written with a capital initial E, followed by letters in lower case.

6.2 Pronunciation

Verbally, with reference to the International Phonetic Alphabet, the name is pronounced:

- In English: **aɪfəl** with the stress on the first syllable.
- In French: **ɛfɛl** with the stress on the second syllable.
- In other languages: the closest approximation to the English or French variant as desired.

7 General description

The following is an informal introduction to the Eiffel language. It is informative only.

7.1 Design principles

The aim of Eiffel is to help specify, design, implement and modify quality software. This goal of quality in software is a combination of many factors; the language design concentrated on the three factors which, in the current state of the industry, are in direct need of improvements: reusability, extendibility and reliability. Also important were other factors such as efficiency, openness and portability.

Reusability is the ability to produce components that may be used in many different applications. Central to the Eiffel approach is the presence of widely used libraries complementing the language, and the language’s support for the production of new libraries.

Extendibility is the ability to produce easily modifiable software. “Soft” as software is supposed to be, it is notoriously hard to modify software systems, especially large ones.

Among quality factors, reusability and extendibility play a special role: satisfying them means having *less* software to write — and hence more time to devote to other important goals such as efficiency, ease of use or integrity.

The third fundamental factor is **reliability**, the ability to produce software that is correct and robust — that is to say, bug-free. Eiffel techniques such as static typing, assertions, disciplined exception handling and automatic garbage collection are essential here.

Four other factors are also part of Eiffel's principal goals:

- The language enables implementors to produce high **efficiency** compilers, so that systems developed with Eiffel may run under speed and space conditions similar to those of programs written in lower-level languages traditionally focused on efficient implementation.
- Ensuring **openness**, so that Eiffel software may cooperate with programs written in other languages.
- Guaranteeing **portability** by a platform-independent language definition, so that the same semantics may be supported on many different platforms.
- **High-precision language definition**, allowing independent implementers to provide compilers and other tools, and providing Eiffel users the guarantee that their software will work on all the Standard-compliant implementations. The language definition does not favor or refer to any particular implementation.

7.2 Object-oriented design

To achieve reusability, extensibility and reliability, the principles of object-oriented design provide the best known technical answer.

An in-depth discussion of these principles falls beyond the scope of this introduction but here is a short definition:

Object-oriented design is the construction of software systems as structured collections of abstract data type implementations, or "classes".

The following points are worth noting in this definition:

- The emphasis is on structuring a system around the types of objects it manipulates (not the functions it performs on them) and on reusing whole data structures together with the associated operations (not isolated routines).
- Objects are described as instances of abstract data types — that is to say, data structures known from an official interface rather than through their representation.
- The basic modular unit, called the class, describes one implementation of an abstract data type (or, in the case of "deferred" classes, as studied below, a set of possible implementations of the same abstract data type).
- The word *collection* reflects how classes should be designed: as units interesting and useful on their own, independently of the systems to which they belong, and ready to be reused by many different systems. Software construction is viewed as the assembly of existing classes, not as a top-down process starting from scratch.
- Finally, the word *structured* reflects the existence of two important relations between classes: the client and inheritance relations.

Eiffel makes these techniques available to software developers in a simple and practical way.

As a language, Eiffel includes more than presented in this introduction, but not *much* more; it is a small language, not much bigger (by such a measure as the number of keywords) than Pascal. The description as given in the Standard text — excluding "informative text" and retaining only the rules — takes up about 80 pages. Eiffel was indeed meant to be a member of the class of languages which programmers can master entirely — as opposed to languages of which most programmers know only a subset. Yet it is appropriate for the development of industrial software systems, as has by now been shown by many full-scale projects, some in the thousands of classes and millions of lines, in companies around the world, for mission-critical systems in many different areas from finance to health care and aerospace.

7.3 Classes

A class, it was said above, is an implementation of an abstract data type. This means that it describes a set of run-time objects, characterized by the **features** (operations) applicable to them, and by the formal properties of these features.

Such objects are called the **direct instances** of the class. Classes and objects should not be confused: "class" is a compile-time notion, whereas objects only exist at run time. This is similar to the difference that exists in pre-object-oriented programming between a program and one execution of that program, or between a type and a run-time value of that type.

("Object-Oriented" is a misnomer; "Class-Oriented Analysis, Design and Programming" would be a more accurate description of the method.)

To see what a class looks like, let us look at a simple example, *ACCOUNT*, which describes bank accounts. But before exploring the class itself it is useful to study how it may be used by other classes, called its **clients**.

A class *X* may become a client of *ACCOUNT* by declaring one or more **entities** of type *ACCOUNT*. Such a declaration is of the form:

```
acc: ACCOUNT
```

An entity such as *acc* that programs can directly modify is called a **variable**. An entity declared of a reference type, such as *acc*, may at any time during execution become "attached to" an object (see figure 1); the type rules imply that this object must be a direct instance of *ACCOUNT* — or, as seen later, of a "descendant" of that class.

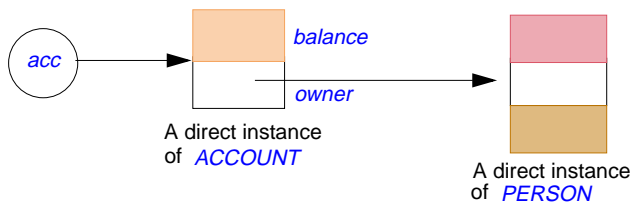


Figure 1: References and objects

An entity is said to be void if it is not attached to any object. As seen below, the type system achieves strict control over void entities, for run-time safety. To obtain objects at run-time, a routine *r* appearing in the client class *X* may use a **creation instruction** of the form

```
create acc
```

which creates a new direct instance of *ACCOUNT*, sets up all its fields for initialization to default values, and attaches *acc* to that instance. A variant of this notation, studied below, makes it possible to override the default initializations.

Once the client has attached *acc* to an object, it may call on this object the features defined in class *ACCOUNT*. Here is an extract with some feature calls using *acc* as their target:

```
acc.open ("Jill")
acc.deposit (5000)
if acc.may_withdraw (3000) then
    acc.withdraw (3000); print (acc.balance)
end
```

These feature calls use dot notation, of the form *target.feature_name*, possibly followed by a list of arguments in parentheses. Features are of two kinds:

- **Routines** (such as *open*, *deposit*, *may_withdraw*, *withdraw*) represent computations applicable to instances of the class.
- **Attributes** represent data items associated with these instances.

Routines are further divided into **procedures** (commands, which do not return a value) and **functions** (queries, returning a value). Here *may_withdraw* is a function returning a boolean; the other three-routines called are procedures.

The above extract of class *X* does not show whether, in class *ACCOUNT*, *balance* is an attribute or a function without argument. This ambiguity is intentional. A client of *ACCOUNT*, such as *X*, does not need to know how a balance is obtained: the balance could be stored as an attribute of every account object, or computed by a function from other attributes. Choosing between these techniques is the business of class *ACCOUNT*, not anybody else's. Because such implementation choices are often changed over the lifetime of a project, it is essential to protect clients against their effects.

This **Principle of Uniform Access** permeates the use of attributes and functions throughout the language design: whenever one of these feature variants can be used, and the other would make sense too, both are permitted. Descendant classes can for example *redefine* a function into an attribute, and attributes can have *assertions* just like functions. The term **query** covers both attributes and functions.

The above example illustrates the Eiffel syntax style: focusing on readability, not overwhelming the reader with symbols, and using simple keywords, each based on a single English word (in its simplest form: **feature** in the singular, **require** and not "requires"). The syntax is free-format — spaces, new lines, tabs have the same effect — and yet does not require a separator between successive instructions. You may use semicolons as separators if you wish, but they are optional in *all* possible contexts, and most Eiffel programmers omit them (as an obstacle to readability) except in the rare case of several instructions written on one line.

So much for how client classes will typically use *ACCOUNT*. Next is a first sketch of how class *ACCOUNT* itself might look. Line segments beginning with *--* are comments. The class includes two **feature** clauses, introducing its features. The first begins with just the keyword **feature**, without further qualification; this means that the features declared in this clause are available (or "exported") to all clients of the class. The second clause is introduced by **feature {NONE}** to indicate that the feature that follows, called *add*, is available to no client. What appears between the braces is a list of client classes to which the listed features are available; *NONE* is a special class of the Kernel Library, which has no instances, so that *add* is in effect a secret feature, available only locally to the other routines of class *ACCOUNT*. In a client class such as *X*, the call *acc.add(-3000)* would be invalid and hence rejected by any Standard-compliant language processing tool.

```
class ACCOUNT feature
  balance: INTEGER
    -- Amount on the account

  owner: PERSON
    -- Account holder

  minimum_balance: INTEGER = 1000
    -- Lowest permitted balance

  open (who: PERSON)
    -- Assign the account to owner who.
  do
    owner := who
  end
```

```

deposit (sum: INTEGER)
    -- Add an amount of sum to the account.
do
    add (sum)
end

withdraw (sum: INTEGER)
    -- Remove an amount of sum from the account.
do
    add (-sum)
end

may_withdraw (sum: INTEGER): BOOLEAN
    -- Is there enough money to withdraw sum?
do
    Result := (balance >= sum + minimum_balance)
end

feature {NONE}
    add (sum: INTEGER)
        -- Add sum to the balance.
do
    balance := balance + sum
end
end

```

Let us examine the features in sequence. The **do ... end** distinguishes routines from attributes. So here the class has implemented *balance* as an attribute, although, as noted, a function would also have been acceptable. Feature *owner* is also an attribute.

The language definition guarantees automatic initialization, so that the balance of an account object will be zero the first time it is accessed. Each type has a default initialization value, such as false for *BOOLEAN*, zero for integer and real types, null character for character types and, for a reference type, either a void value or a reference to a new object of that type. The class designer may also provide clients with different initialization options, as will be shown in a revised version of this example.

The other public features, *open*, *deposit*, *withdraw* and *may_withdraw* are straightforward routines. The special variable *Result*, used in *may_withdraw*, denotes the function result; it is initialized on function entry to the default value of the function's result type.

The secret procedure *add* serves for the implementation of the public procedures *deposit* and *withdraw*; the designer of *ACCOUNT* judged it too general to be exported by itself. The clause *= 1000* introduces *minimum_balance* as a constant attribute, which need not occupy any space in instances of the class; in contrast, every instance has a field for every non-constant attribute such as *balance*.

In Eiffel's object-oriented programming style, every operation is relative to a certain object. In a client invoking the operation, this object is specified by writing the corresponding entity on the left of the dot, as *acc* in *acc.open ("Jill")*. Within the class, however, the "current" instance to which operations apply usually remains implicit, so that unqualified feature names, such as *owner* in procedure *open* or *add* in *deposit*, mean "the *owner* attribute or *add* routine relative to the current instance".

If you need to denote the current object explicitly, you may use the special entity *Current*. For example the unqualified occurrences of *add* appearing in the above class are essentially equivalent to *Current.add*.

In some cases, infix or prefix notation will be more convenient than dot notation. For example, if a class *VECTOR* offers an addition routine, most people will feel more comfortable with calls of the form *v + w* than with the dot-notation call *v.plus (w)*. To make this possible it suffices to give the routine a name of the form *plus alias "+"*; internally, however, the operation is still a normal routine call. You can also use unary operators for prefix notation. It is also possible to define a *bracket*

alias, as in the feature *item alias* "[*i*]" of the Kernel Library class *ARRAY*, which returns an array element of given index *i*: in dot notation, you will write this as *your_array.item(i)*, but the bracket alias allows *your_array [i]* as an exact synonym. At most one feature per class may have a bracket alias. These techniques make it possible to use well-established conventions of mathematical notation and traditional programming languages in full application of object-oriented principles.

The above simple example has shown the basic structuring mechanism of the language: the class. A class describes a data structure, accessible to clients through an official interface comprising some of the class features. Features are implemented as attributes or routines; the implementation of exported features may rely on other, secret ones.

7.4 Types

Eiffel is strongly typed for readability and reliability. Every entity is declared of a certain type, which may be either a reference type or an expanded type.

Any type *T* is based on a class, which defines the operations that will be applicable to instances of *T*. The difference between the two categories of type affects the semantics of using an instance of *T* as source of an *attachment*: assignment or argument passing. An attachment from an object of a reference type will attach a new reference to that object; with an expanded type, the attachment will *copy* the contents of the object. Similarly, comparison operations such as *a = b* will compare references in one case and objects contents in the other. (To get object comparison in all cases, use *a ~ b*.) We talk of objects with *reference semantics* and objects with *copy semantics*. Syntactically, the difference is simple: a class declared without any particular marker, like *ACCOUNT*, yields a reference type. To obtain an expanded type, just start with **expanded class** instead of just **class**.

It may be useful to think of expanded and reference types in terms of figure 2, where we assume that *ACCOUNT* has an extra attribute *exp* of type *EXP*, using a class declared as **expanded class** *EXP*. Figure 2 shows the entity *acc* denoting at run time a reference to an instance of *ACCOUNT* and, in contrast, *exp* in that instance denoting a **subobject**, not a reference to another object:

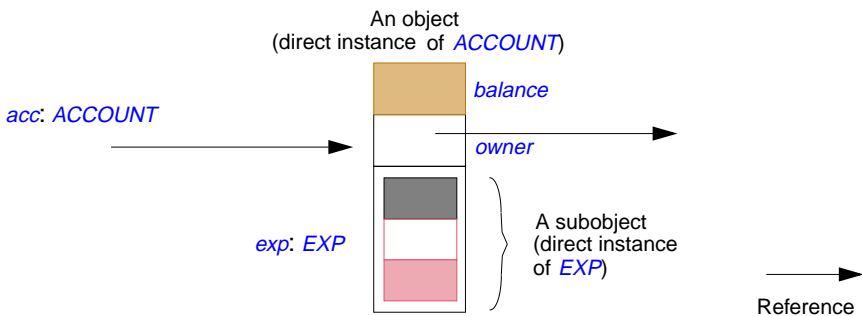


Figure 2: Object and subobject

This is only an illustration, however, and implementations are not required to implement expanded values as subobjects. What matters is the distinction between copy and reference semantics.

An important group of expanded types, based on library classes, includes the basic types *INTEGER*, *REAL*, *CHARACTER* and *BOOLEAN*. Clearly, the value of an entity declared of type *INTEGER* should be an integer, not a reference to an object containing an integer value. Operations on these types are defined by prefix and infix operators such as "+" and "<".

As a result of these conventions, the type system is uniform and consistent: all types, reference or expanded — including the basic types —, are defined from classes.

In the case of basic types, for obvious reasons of efficiency, compilers can and usually do implement the usual arithmetic and boolean operations directly through the corresponding

machine operations, not through routine calls. So the performance is the same as if basic types were “magic”, outside of the object-oriented type system. But this is only a compiler optimization, which does not hamper the conceptual homogeneity of the type edifice.

7.5 Assertions

If classes are to deserve their definition as abstract data type implementations, they must be known not just by the available operations, but also by the formal properties of these operations, which did not appear in the above example.

Eiffel encourages software developers to express formal properties of classes by writing **assertions**, which may in particular appear in the following roles:

- Routine **preconditions** express the requirements that clients must satisfy whenever they call a routine. For example the designer of *ACCOUNT* may wish to permit a withdrawal operation only if it keeps the account’s balance at or above the minimum. Preconditions are introduced by the keyword **require**.
- Routine **postconditions**, introduced by the keyword **ensure**, express conditions that the routine (the supplier) guarantees on return, if the precondition was satisfied on entry.
- A class **invariant** must be satisfied by every instance of the class whenever the instance is externally accessible: after creation, and after any call to an exported routine of the class. The invariant appears in a clause introduced by the keyword **invariant**, and represents a general consistency constraint imposed on all routines of the class.

The above class *ACCOUNT* typifies why classes often need such assertions. Implicitly, as suggested by the feature names, the *balance* of an account should always remain at least equal to its *minimum_balance*. But nothing in the class text expresses this fundamental property, or the constraints it implies on the arguments to such features as *deposit* and *withdraw*. By adding assertions we make these properties an integral part of the class. We also use this opportunity to make the class more flexible by making *minimum_balance* no longer a constant (1000 above) but a variable attribute to be set by each client when it creates an instance of the class. The revised version is:

```
class ACCOUNT create
  make
feature
  ... Some features as before: balance, owner, open ...

  minimum_balance: INTEGER
    -- Minimum permitted value for balance

  deposit (sum: INTEGER)
    -- Deposit sum into the account.
    require
      positive: sum > 0
    do
      add (sum)
    ensure
      deposited: balance = old balance + sum
    end
```

```

withdraw (sum: INTEGER)
  -- Withdraw sum from the account.
  require
    positive: sum > 0
    sufficient_funds: sum <= balance - minimum_balance
  do
    add (-sum)
  ensure
    withdrawn: balance = old balance - sum
  end

may_withdraw ... -- As before

feature {NONE}
  add ... -- As before
  make (initial, min: INTEGER)
    -- Initialize account with balance initial and minimum balance min.
    require
      not_under_minimum: initial >= min
    do
      minimum_balance := min
      balance := initial
    ensure
      balance_initialized: balance = initial
      minimum_initialized: minimum_balance = min
    end

  invariant
    sufficient_balance: balance >= minimum_balance
end

```

The notation **old** *expression* may only be used in a routine postcondition. It denotes the value the *expression* had on routine entry.

In its last version above, the class now includes a creation procedure, *make*. With the first version of *ACCOUNT*, clients used creation instructions such as **create** *acc1* to create accounts; but then the default initialization, setting *balance* to zero, violated the invariant. By having one or more creation procedures, listed in the **create** clause at the beginning of the class text, a class offers a way to override the default initializations. The effect of

```
create acc1.make (5500, 1000)
```

is to allocate the object (as with default creation) and to call procedure *make* on it, with the arguments given. This call is correct since it satisfies the precondition; it will ensure that the created object satisfies the invariant.

Note that the same keyword, **create**, serves both to introduce creation instructions and the creation clause listing creation procedures at the beginning of the class.

A procedure listed in the creation clause, such as *make*, otherwise enjoys the same properties as other routines, especially for calls. Here the procedure *make* is secret since it appears in a clause starting with **feature** {NONE}; so it would be invalid for a client to include a call such as

```
acc.make (5500, 1000)
```

To make such a call valid, it would suffice to move the declaration of *make* to the first **feature** clause of class *ACCOUNT*, which carries no export restriction. Such a call does not create any new object, but simply resets the balance of a previously created account.

Syntactically, assertions are boolean expressions, with a few extensions such as the **old** notation. Writing a succession of assertion clauses, as in the precondition to *withdraw*, is

equivalent to “and-ing” them, but permits individual identification of the components. (As with instructions you could use a semicolon between assertion clauses, although it is optional and generally omitted.)

Assertions play a central part in the Eiffel method for building reliable object-oriented software. They serve to make explicit the assumptions on which programmers rely when they write software elements that they believe are correct. Writing assertions, in particular preconditions and postconditions, amounts to spelling out the terms of the **contract** which governs the relationship between a routine and its callers. The precondition binds the callers; the postcondition binds the routine.

The underlying theory of *Design by Contract*, the centerpiece of the Eiffel method, views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by the assertions.

Assertions are also an indispensable tool for the documentation of reusable software components: one cannot expect large-scale reuse without a precise documentation of what every component expects (precondition), what it guarantees in return (postcondition) and what general conditions it maintains (invariant).

Documentation tools in Eiffel implementations use assertions to produce information for client programmers, describing classes in terms of observable behavior, not implementation. In particular the **contract view** of a class, which serves as its basic documentation, is obtained from the full text by removing all non-exported features and all implementation information such as **do** clauses of routines, but keeping interface information and in particular assertions. Here is the interface of the above class (with some features omitted):

```
class interface ACCOUNT create
  make
feature
  balance: INTEGER
    -- Amount on the account

  minimum_balance: INTEGER
    -- Lowest permitted balance

  deposit (sum: INTEGER)
    -- Deposit sum into the account.
  require
    positive: sum > 0
  ensure
    deposited: balance = old balance + sum
  withdraw (sum: INTEGER)
    -- Withdraw sum from the account.
  require
    positive: sum > 0
    sufficient_funds: sum <= balance - minimum_balance
  ensure
    withdrawn: balance = old balance - sum
invariant
  sufficient_balance: balance >= minimum_balance
end
```

This is not an Eiffel text, only documentation of Eiffel classes, hence the use of slightly different syntax to avoid any confusion (**class interface** rather than **class**). In accordance with observations made above, the output for *balance* would be the same if this feature were a function rather than an attribute.

Such an interface can be produced by automatic tools from the text of the software. It serves as the primary form of class documentation. A variant of the contract view includes inherited features along with those introduced in the class itself.

It is also possible to evaluate assertions at run time, to uncover potential errors (“bugs”). The implementation provides several levels of assertion monitoring: preconditions only, postconditions etc. With monitoring on, an assertion that evaluates to true has no further effect on the execution. An assertion that evaluates to false will trigger an exception, as described next; in the absence of a specific exception handler the exception will cause an error message and termination.

This ability to check assertions provides a powerful testing and debugging mechanism, in particular because the classes of widely used libraries are equipped with extensive assertions.

Run-time checking, however, is only one application of assertions, whose role as design and documentation aids, as part of the theory of Design by Contract, exerts a pervasive influence on the Eiffel style of software development.

7.6 Exceptions

Whenever there is a contract, the risk exists that someone will break it. This is where exceptions come in.

Exceptions — contract violations — may arise from several causes. One is assertion violations, if assertions are monitored. Another is the occurrence of a signal triggered by the hardware or operating system to indicate an abnormal condition such as arithmetic overflow or lack of memory to create a new object.

Unless a routine has made specific provision to handle exceptions, it will **fail** if an exception arises during its execution. Failure of a routine is a third cause of exception: a routine that fails triggers an exception in its caller.

A routine may, however, handle an exception through a **rescue** clause. This optional clause attempts to “patch things up” by bringing the current object to a stable state (one satisfying the class invariant). Then it can terminate in either of two ways:

- The **rescue** clause may execute a **retry** instruction, which causes the routine to restart its execution from the beginning, attempting again to fulfil its contract, usually through another strategy. This assumes that the instructions of the **rescue** clause, before the **retry**, have attempted to correct the cause of the exception.
- If the **rescue** clause does not end with **retry**, then the routine fails: it returns to its caller, immediately signaling an exception. (The caller’s **rescue** clause will be executed according to the same rules.)

The principle is that **a routine must either succeed or fail**: either it fulfils its contract, or it does not; in the latter case it must notify its caller by triggering an exception.

Usually, only a few routines of a system will include explicit **rescue** clauses. An exception occurring during the execution of a routine with no **rescue** clause will trigger a predefined rescue procedure, which does nothing, and so will cause the routine to fail immediately, propagating the exception to the routine’s caller.

An example using the exception mechanism is a routine *attempt_transmission* which tries to transmit a message over a phone line. The actual transmission is performed by an external, low-level routine *transmit*; once started, however, *transmit* may abruptly fail, triggering an exception, if the line is disconnected. Routine *attempt_transmission* tries the transmission at most

50 times; before returning to its caller, it sets a boolean attribute *successful* to **true** or **false** depending on the outcome. Here is the text of the routine:

```
attempt_transmission (message: STRING)
  -- Try to transmit message, at most 50 times.
  -- Set successful accordingly.
  local failures: INTEGER
  do
    if failures < 50 then
      transmit (message); successful := true
    else
      successful := false
    end
  rescue
    failures := failures + 1; retry
  end
```

Initialization rules ensure that *failures*, a local variable, is initially zero.

This example illustrates the simplicity of the mechanism: the **rescue** clause never attempts to achieve the routine's original intent; this is the sole responsibility of the body (the **do** clause). The only role of the **rescue** clause is to clean up the objects involved, and then either to fail or to retry.

The Kernel Library provides a class *EXCEPTION* and a number of descendants describing specific kinds of exception. Triggering of an exception produces an instance of one of these types, making it possible, in the **rescue** clause, to perform more specific exception processing.

This disciplined exception mechanism is essential for software developers, who need protection against unexpected events, but cannot be expected to sacrifice safety and simplicity to pay for this protection.

7.7 Genericity

Building software components (classes) as implementations of abstract data types yields systems with a solid architecture but does not in itself suffice to ensure reusability and extendibility. Two key techniques address the problem: genericity (unconstrained or constrained) and inheritance. Let us look first at the unconstrained form.

To make a class generic is to give it **formal generic parameters** representing arbitrary types, as in these examples from typical libraries:

```
ARRAY[G]
LIST[G]
LINKED_LIST[G]
```

These classes describe data structures — arrays, lists without commitment to a specific representation, lists in linked representation — containing objects of a certain type. The formal generic parameter *G* represents this type.

Such a class describes a type template. To derive a directly usable type, you must provide a type corresponding to *G*, called an **actual generic parameter**; this might be a basic type (such as *INTEGER*) or a reference type. Here are some possible generic derivations:

```
it: LIST[INTEGER]
aa: ARRAY[ACCOUNT]
aal: LIST[ARRAY[ACCOUNT]]
```

As the last example indicates, an actual generic parameter may itself be generically derived.

Without genericity, it would be impossible to obtain static type checking in a realistic object-oriented language.

A variant of this mechanism, *constrained* genericity, introduced below after inheritance, enables a class to place specific requirements on possible actual generic parameters.

7.8 Inheritance

Inheritance, the other fundamental generalization mechanism, makes it possible to define a new class by combination and specialization of existing classes rather than from scratch.

The following simple example describes lists implemented by arrays, combining *LIST* and *ARRAY* through inheritance:

```
class ARRAYED_LIST [G] inherit
  LIST [G]
inherit {NONE}
  ARRAY [G]
  export ... See below ...end
feature
  ... Specific features of lists implemented by arrays ...
end
```

The *inherit...* clauses list all the “parents” of the new class, which is said to be their “heir”. (The “ancestors” of a class include the class itself, its parents, grandparents etc.; the reverse term is “descendant”.) Declaring *ARRAYED_LIST* as shown ensures that all the features and properties of lists and arrays are applicable to arrayed lists as well. Since the class has more than one parent, this is a case of *multiple* inheritance.

In this case one of the parents is introduced by a different clause, reading *inherit {NONE}*; this specifies **non-conforming inheritance**, where it will not be possible to assign values of the new types to variables of the parent type. The other branch, with just *inherit*, is conforming, so we can assign an *ARRAYED_LIST [T]* to a *LIST [T]*. This reflects the distinction between the “subtyping” and “pure reuse” forms of inheritance.

Standard graphical conventions (figure 3) serve to illustrate such inheritance structures:

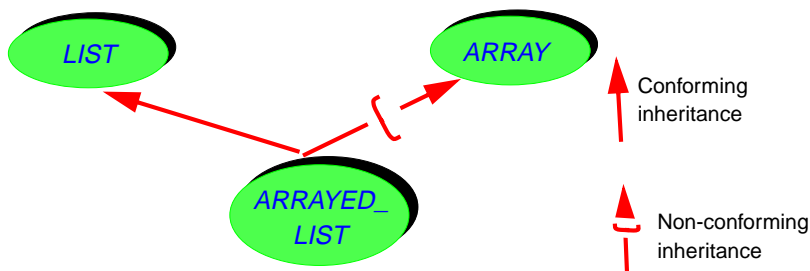


Figure 3: Inheritance

A non-conforming heir class such as *ARRAYED_LIST* needs the ability to define its own export policy. By default, inherited features keep their export status (publicly available, secret, available to selected classes only); but this may be changed in the heir. Here, for example, *ARRAYED_LIST* will export only the exported features of *LIST*, making those of

ARRAY unavailable directly to *ARRAYED_LIST*'s clients. The syntax to achieve this is straightforward:

```
class ARRAYED_LIST[G] inherit
    LIST[G]
inherit {NONE}
    ARRAY[G]
export {NONE} all end
... The rest as above ...
```

Another example of multiple inheritance comes from a windowing system based on a class *WINDOW*, as in the EiffelVision graphics library. Windows have **graphical** features: a height, a width, a position, routines to scale windows, move them, and other graphical operations. The system permits windows to be nested, so that a window also has **hierarchical** features: access to subwindows and the parent window, adding a subwindow, deleting a subwindow, attaching to another parent and so on. Rather than writing a complex class that would contain specific implementations for all of these features, it is preferable to inherit all hierarchical features from *TREE*, and all graphical features from a class *RECTANGLE*. In this case both branches are conforming, so a single **inherit** clause listing two parents suffices:

```
class WINDOW inherit
    RECTANGLE
    TREE [WINDOW]
...
```

Inheritance yields remarkable economies of effort — whether for analysis, design, implementation or evolution — and has a profound effect on the entire development process.

The very power of inheritance demands adequate means to keep it under control. Multiple inheritance, in particular, raises the question of name conflicts between features inherited from different parents. You may simply remove such a name conflict through **renaming**, as in

```
class C inherit
    A rename x as x1, y as y1 end
    B rename x as x2, y as y2 end
feature...
```

Here, if both *A* and *B* have features named *x* and *y*, class *C* would be invalid without the renaming.

Renaming also serves to provide more appropriate feature names in descendants. For example, class *WINDOW* may inherit a routine *insert_subtree* from *TREE*. For clients of *WINDOW*, however, such a routine name is no longer appropriate. An application using this class for window manipulation needs coherent window terminology, and should not be concerned with the inheritance structure that led to the implementation of the class. So you may wish to rename *insert_subtree* as *add_subwindow* in the inheritance clause of *WINDOW*.

As a further facility to protect against misusing the multiple inheritance mechanism, the invariants of all parent classes automatically apply to a newly defined class.

7.9 Polymorphism and dynamic binding

Inheritance is not just a module combination and enrichment mechanism. It also enables the definition of flexible entities that may become attached to objects of various forms at run time, a property known as polymorphism.

Complementary mechanisms make this possibility particularly powerful: **feature redefinition** and **dynamic binding**. The first enables a class to redefine some or all of the features which it inherits

from its parents. For an attribute or function, the redefinition may affect the type, replacing the original by a descendant; for a routine it may also affect the implementation, replacing the original's routine body by a new one.

Assume for example a class *POLYGON*, describing polygons, whose features include an array of points representing the vertices and a function *perimeter* which computes a polygon's perimeter by summing the successive distances between adjacent vertices. An heir of *POLYGON* may begin:

```

class RECTANGLE inherit
    POLYGON redefine perimeter end
feature -- Specific features of rectangles, such as:
    side1: REAL; side2: REAL
    perimeter: REAL
        -- Rectangle-specific version
    do
        Result := 2 * (side1 + side2)
    end
    ... Other RECTANGLE features ...
end

```

Here it is appropriate to redefine *perimeter* for rectangles as there is a simpler and more efficient algorithm. Note the explicit **redefine** subclause (which would come before the **rename** if present).

Other descendants of *POLYGON* may also have their own redefinitions of *perimeter*. The version to use in any call is determined by the run-time form of the target. Consider the following class fragment:

```

p: POLYGON; r: RECTANGLE
... create p; create r, ...
if some_condition then
    p := r
end
print (p.perimeter)

```

The polymorphic assignment *p := r* is valid because the type of the source, *RECTANGLE*, conforms, through inheritance, to the type of the target, *POLYGON*. If *some_condition* is false, *p* will be attached to an object of type *POLYGON* for the computation of *p.perimeter*, which will thus use the polygony algorithm. In the opposite case, however, *p* will be attached to a rectangle; then the computation will use the version redefined for *RECTANGLE*. This is known as dynamic binding.

Dynamic binding provides high flexibility. The advantage for clients is the ability to request an operation (such as perimeter computation) without explicitly selecting one of its variants; the choice only occurs at run-time. This is essential in large systems, where many variants may be available; each component must be protected against changes in other components.

This technique is particularly attractive when compared to its closest equivalent in non-object-oriented approaches where you would need records with variant components, and **case** instructions to discriminate between variants. This means that every client must know about every possible case, and that any extension may invalidate a large body of existing software.

Redefinition, polymorphism and dynamic binding support a development mode in which every module is open and incremental. When you want to reuse an existing class but need to adapt it to a new context, you can always define a new descendant of that class (with new features, redefined ones, or both) without any change to the original. This facility is of great importance in software

development, an activity which — whether by design or by circumstance — is invariably incremental.

The power of polymorphism and dynamic binding demands adequate controls. First, feature redefinition is explicit. Second, because the language is typed, a compiler can check statically whether a feature application *a.f* is valid, as discussed in more detail below. In other words, the language reconciles dynamic *binding* with static *typing*. Dynamic binding guarantees that whenever more than one version of a routine is applicable the *right* version (the one most directly adapted to the target object) will be selected. Static typing means that the compiler makes sure there is *at least one* such version.

This policy also yields an important performance benefit: the design of the inheritance mechanism makes it possible for an implementation to find the appropriate routine, for a dynamically bound call, in constant time.

Assertions provide a further mechanism for controlling the power of redefinition. In the absence of specific precautions, redefinition may be dangerous: how can a client be sure that evaluation of *p.perimeter* will not in some cases return, say, the area? Preconditions and postconditions provide the answer by limiting the amount of freedom granted to eventual redefiners. The rule is that any redefined version must satisfy a weaker or equal precondition and ensure a stronger or equal postcondition than in the original. In other words, it must stay within the semantic boundaries set by the original assertions.

The rules on redefinition and assertions are part of the Design by Contract theory, where redefinition and dynamic binding introduce subcontracting. *POLYGON*, for example, subcontracts the implementation of *perimeter* to *RECTANGLE* when applied to any entity that is attached at run-time to a rectangle object. An honest subcontractor is bound to honor the contract accepted by the prime contractor. This means that it may not impose stronger requirements on the clients, but may accept more general requests, so that the precondition may be weaker; and that it must achieve at least as much as promised by the prime contractor, but may achieve more, so that the postcondition may be stronger.

7.10 Combining genericity and inheritance

Genericity and inheritance, the two fundamental mechanisms for generalizing classes, may be combined in two fruitful ways.

The first technique yields **polymorphic data structures**. Assume that in the generic class *LIST[G]* the insertion procedure *put* has a formal argument of type *G*, representing the element to be inserted. Then with a declaration such as

```
pl: LIST[POLYGON]
```

the type rules imply that in a call *pl.put (...)* the argument may be not just of type *POLYGON*, but also of type *RECTANGLE* (an heir of *POLYGON*) or any other type conforming to *POLYGON* through inheritance.

The conformance requirement used here is the inheritance-based type compatibility rule: *V* can only conform to *T* if it is a descendant of *T*.

Structures such as *pl* may contain objects of different types, hence the name “polymorphic data structure”. Such polymorphism is, again, made safe by the type rules: by choosing an actual generic parameter (*POLYGON* in the example) based higher or lower in the inheritance graph, you extend or restrict the permissible types of objects in *pl*. A fully general list would be declared as

```
LIST[ANY]
```

where *ANY*, a Kernel Library class, is automatically an ancestor of any class that you may write.

The other mechanism for combining genericity and inheritance is **constrained genericity**. By indicating a class name after a formal generic parameter, as in

VECTOR [*G* → *ADDABLE*]

you express that only descendants of that class (here *ADDABLE*) may be used as the corresponding actual generic parameters. This makes it possible to use the corresponding operations. Here, for example, class *VECTOR* may define a routine *plus alias "+"* for adding vectors, based on the corresponding routine from *ADDABLE* for adding vector elements. Then by making *VECTOR* itself inherit from *ADDABLE*, you ensure that it satisfies its own generic constraint and enable the definition of types such as *VECTOR* [*VECTOR* [*T*]].

Unconstrained genericity, as in *LIST* [*G*], may be viewed as an abbreviation for genericity constrained by *ANY*, as in

LIST [*G* → *ANY*]

With these basic forms of genericity, it is not possible to *create* an instance of a formal generic type, for example an object of type *G* in *VECTOR* [*G*]. Indeed without further information we don't know whether any creation procedures are available. To request specific ones for an actual generic parameter, list them in the class declaration, just after the constraint:

VECTOR [*G* → *ADDABLE* *create make end*]

Then for *x* of type *G* you can use the instruction *create x.make (a)*, with the appropriate argument type for *a* as specified for *make* in *ADDABLE*, and rely on the guarantee that when this gets applied to a *VECTOR* [*T*] for a permissible *T* this type will have its own appropriate version of *make*.

7.11 Deferred classes

The inheritance mechanism includes one more major component: deferred routines and classes.

Declaring a routine *r* as deferred in a class *C* expresses that there is no default implementation of *r* in *C*; such implementations will appear in eventual descendants of *C*. A class having one or more deferred routines is itself said to be deferred. A non-deferred routine or class is called **effective**.

For example, a system used by a Department of Motor Vehicles to register vehicles could include a class of the form

```

deferred class VEHICLE feature
  dues_paid (year: INTEGER): BOOLEAN
    do... end
  valid_plate (year: INTEGER): BOOLEAN
    do... end
  register (year: INTEGER)
    -- Register vehicle for year.
  require
    dues_paid: dues_paid (year)
  deferred
  ensure
    valid_plate: valid_plate (year)
  end
  ... Other features, deferred or effective...
end

```

This example assumes that no single registration algorithm applies to all kinds of vehicle; passenger cars, motorcycles, trucks etc. are all registered differently. But the same precondition and postcondition apply in all cases. The solution is to treat `register` as a deferred routine, making `VEHICLE` a deferred class. Descendants of class `VEHICLE`, such as `CAR` or `TRUCK`, effect this routine, that is to say, give effective versions (figure 4). An effecting is similar to a redefinition; only here there is no effective definition in the original class, just a specification in the form of a deferred routine. There is no need here for a `redefine` clause; the effective versions simply take over any inherited deferred version. The term **redeclaration** covers both redefinition and effecting.

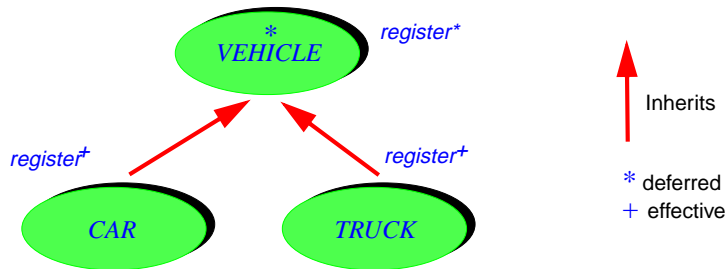


Figure 4: Abstracting variants into a deferred parent

Deferred classes describe a group of implementations of an abstract data type rather than just a single implementation. You may not instantiate a deferred class: `create v` is invalid if `v` is a variable declared of type `VEHICLE`. But you may assign to `v` a reference to an instance of a non-deferred descendant of `VEHICLE`. For example, assuming `CAR` and `TRUCK` provide effective definitions for all deferred routines of `VEHICLE`, the following will be valid:

```

v: VEHICLE; c: CAR; t: TRUCK
...
create c ...; create t ...;...
if some_test then v := c else v := t end
v.register (1996)
  
```

This example fully exploits polymorphism: depending on the outcome of `some_test`, `v` will be treated as a car or a truck, and the appropriate registration algorithm will be applied. Also, “Some test” may depend on some event whose outcome is impossible to predict until run-time, for example a user clicking with the mouse to select one among several vehicle icons displayed on the screen.

Deferred classes are particularly useful at the **design** stage. The first version of a module may be a deferred class, which will later be refined into one or more effective (non-deferred) classes. Particularly important for this application is the possibility of associating a precondition and a postcondition with a routine even though it is a deferred routine (as with `register` above), and an invariant with a class even though it is a deferred class. This enables the designer to attach precise semantics to a module at the design stage, long before making any implementation choices.

These possibilities make Eiffel an attractive alternative to specialized notations, graphical or textual, for design and also for **analysis**. The combination of deferred classes to capture partially understood concepts, assertions to express what is known about their semantics, and the language's other structuring facilities (information hiding, inheritance, genericity) to obtain clear, convincing architectures, yields a higher-level design method. A further benefit, of course, is that

the notation is also a programming language, making the development cycle smoother by reducing the gap between design and implementation.

At the analysis stage, deferred classes describe not software objects, but objects from the external reality's model — documents, airplanes, investments. The Eiffel mechanisms are just as attractive for such modeling.

An important property of deferred classes supporting all these lifecycle tasks, as part of a seamless software development cycle, is that they do not have to be *fully* deferred, like pure “interfaces”. A class may have a mix of effective features capturing what is known at a certain stage of the development, and deferred ones describing what remains to be refined. This supports a continuous refinement-based process, proceeding smoothly from the abstract to the concrete.

7.12 Tuples and agents

A simple extension to the notion of class is the tuple. The type `TUPLE [A, B, C]` has, as its instances, sequences (“tuples”) whose first three elements are of types `A`, `B` and `C` respectively. A tuple expression appears as simply `[a1, b1, c1]` with elements of the given type. It is also possible in the tuple type declaration to label the components, as in `TUPLE [x: A; y: B; z: C]`, making it simpler to access the elements, as in `your_tuple.y`, with the proper type, here `B`, rather than `your_tuple.item (2)` of type `ANY` by default. Tuples provide a simpler alternative to classes when you don't need specific features, just a sequence of values of given types.

Tuples also help in the definition of **agents**. An agent is a way to define an object that represents a certain routine, ready to be called, possibly with some of its arguments set at the time of the agent definition (**closed operands**) and others to be provided at the time of each actual call (**open operands**). For example with a routine `r (x: A; y: B)` in a class `C`, as well as `a1` of type `A` and `c1` of type `C`, the agent

```
agent c1.r(a1, ?)
```

represents the routine `r` ready to be called on the target `c1`, with the argument `a1` and another argument (corresponding to `y`, of type `B`) to be provided at the time of the call. The question mark `?` represents an open operand. The routine using this agent, for example having received it as an actual argument to a routine, for the formal argument `operation`, can then call the associated routine through

```
operation.call [b1]
```

for some `b1` of type `B`, the only open operand. This will have the same effect as an original call `c1.r (a1, b1)`, but the routine executing it does not know that `operation` represents `r` rather than any other routine with the same expected open operands. The argument to `call` is not directly `b1` but a **tuple** with `b1` as its sole item; this is because `call` — a routine of the corresponding general-purpose agent class in the Kernel Library — must be able to accept argument sequences of any length, while ensuring type safety.

Agents add a further level of expressiveness to the mechanisms discussed earlier. They are particularly useful for numerical applications — for example to pass **agent** `f`, where `f` is a mathematical function, to an integration routine — and Graphical User Interface (GUI) applications, where they provide an attractive alternative to techniques such as “function pointers” and the Observer Pattern. For example an application may pass the above **agent** `c1.r (a1, ?)` to a GUI routine, telling it to “subscribe” the associated operation to certain event types such as mouse click. When such an event occurs, the GUI will automatically trigger the operation through a `call`.

7.13 Type- and void-safety

As noted, Eiffel puts a great emphasis on reliability through static typing. Polymorphism and dynamic binding in particular are controlled by type rules. The basic requirement is simple: an assignment of the form `a := b` is permitted only if `a` and `b` are of reference types `A` and `B`, based on classes `A` and `B` such that `B` is a descendant of `A`. The same applies to argument passing.

This corresponds to the intuitive idea that a value of a more specialized type may be assigned to an entity of a less specialized type — but not the reverse. (As an analogy, consider that if you request vegetables, getting green vegetables is fine, but if you ask for green vegetables, receiving a dish labeled just “vegetables” is not acceptable, as it could include, say, carrots.)

This inheritance-based type system rules out numerous errors, some potentially catastrophic if their detection was left to run time.

Another, just as dangerous type of error has generally eluded static detection: *void calls*. In a framework offering references or pointers, the risk exists that in the execution of a call *x.f(...)* the value of *x* will be void rather than attached to an object, leading to a crash. The design of Eiffel treats this case through the type system. Specifically:

- Types are by default **attached**, meaning that they do not permit void values. To support void, a type must be declared as **detachable**: *? T* rather than just *T*.
- You may use a qualified call *x.f(...)* only if the type of *x* is attached.
- As part of the conformance rules, you may assign a *T* source to a *? T* target but (for risk of losing the characteristic property of attached types) not the other way around.
- All entities, as noted, are initialized to default values. For detachable types the default value is void, but for an attached type it must always be attached to an object. This means that either the type must provide a default creation procedure (a procedure *default_create* from class *ANY* is available for that purpose, which any class can use as creation procedure, after possibly redefining it to suit its needs), or every variable must be explicitly initialized before use.

These simple rules, compiler-enforceable, remove a whole category of tricky and dangerous failures.

7.14 Putting a system together

This discussion has focused so far on individual classes. This is consistent with the Eiffel method, which emphasizes reusability and, as a consequence, the construction of autonomous modules.

To execute software, you will need to group classes into executable compounds. Such a compound is called a **system** — the Eiffel concept closest to the traditional notion of program — and is defined by the following elements:

- A set of classes, called a **universe**.
- The designation of the system’s **root type**, based on one of these classes (the **root class**).
- The designation of one of the creation procedures of the root class as the **root procedure**.

To execute such a system is to create one direct instance of the root type (the **root object** for the current execution), and to apply to it the root procedure — which will usually create other objects, call other routines and so on.

The method suggests grouping related classes — typically 5 to 40 classes — into collections called **clusters**. A common convention, for the practical implementation on the file system, is to store each class in a file, and associate each cluster with a directory. Then the universe is simply the set of classes stored across a set of directories.

The classes of a system will include its root class and all the classes that it **needs** directly or indirectly, where a class is said to need another if it is one of its heirs or clients.

To specify a system you will need to state, in addition to the list of directories, the root type and the root procedure (which must be one of the creation procedures of the root class). This is achieved through either a graphical interface or a control file.

8 Language specification

8.1 General organization

Informative text

The remainder of the text provides the precise specification of Eiffel.

The overall order of the description is top-down, global structure to specific details:

- Conventions for the language description and basic conventions of the language itself.
- Architecture of Eiffel software, including the fundamental structuring mechanisms: cluster, class, feature, inheritance, client.
- Key elements of a class: routines and assertions.
- Type and type adaptation mechanisms, including redeclaration, genericity, tuples, conformance, convertibility and repeated inheritance.
- Control structures.
- Dynamic model: objects, attributes, entities, creation, copying.
- The calling mechanism and its consequences: expressions, type checking, barring void calls.
- Advanced mechanisms: exceptions, agents.
- Elementary mechanisms: constants, basic types, lexical elements.

End

8.2 Syntax, validity and semantics

8.2.1 Definition: Syntax, BNF-E

Syntax is the set of rules describing the structure of software texts.

The notation used to define Eiffel's syntax is called **BNF-E**.

Informative text

"BNF" is *Backus-Naur Form*, a traditional technique for describing the syntax of a certain category of formalisms ("*context-free languages*"), originally introduced for the description of Algol 60. BNF-E adds a few conventions — one production per construct, a simple notation for repetitions with separators — to make descriptions clearer. The range of formalisms that can be described by BNF-E is the same as for traditional BNF.

End

8.2.2 Definition: Component, construct, specimen

Any class text, or syntactically meaningful part of it, such as an instruction, an expression or an identifier, is called a **component**.

The structure of any kind of components is described by a **construct**. A component of a kind described by a certain construct is called a **specimen** of that construct.

Informative text

For example, any particular class text, built according to the rules given in this language description, is a *component*. The *construct* **Class** describes the structure of class texts; any class text is a *specimen* of that construct. At the other end of the complexity spectrum, an identifier such as *your_variable* is a specimen of the construct **Identifier**.

Although we could use the term "instance" in lieu of "specimen", it could cause confusion with the instances of an Eiffel class — the run-time objects built according to the class specification.

End

8.2.3 Construct Specimen convention

The phrase “an *X*”, where *X* is the name of a construct, serves as a shorthand for “a specimen of *X*”.

Informative text

For example, “a *Class*” means “a specimen of construct *Class*”: a text built according to the syntactical specification of the construct *Class*.

End

8.2.4 Construct Name convention

Every construct has a name starting with an upper-case letter and continuing with lower-case letters, possibly with underscores (to separate parts of the name if it uses several English words).

Informative text

Typesetting conventions complement the Construct Name convention: construct names, such as *Class*, always appear in Roman and in *Green* — distinguishing them from the blue of Eiffel text, as in *Result := x*.

End

8.2.5 Definition: Terminal, non-terminal, token

Specimens of a **terminal construct** have no further syntactical structure. Examples include:

- Reserved words such as *if* and *Result*.
- Manifest constants such as the integer *234*; symbols such as *;* (semicolon) and *+* (plus sign).
- Identifiers (used to denote classes, features, entities) such as *LINKED_LIST* and *put*.

The specimens of terminal constructs are called **tokens**.

In contrast, the specimens of a **non-terminal** construct are defined in terms of other constructs.

Informative text

Tokens (also called **lexical components**) form the basic vocabulary of Eiffel texts. By starting with tokens and applying the rules of syntax you may build more complex components — specimens of non-terminals.

End

8.2.6 Definition: Production

A **production** is a formal description of the structure of all specimens of a non-terminal construct. It has the form

Construct \triangleq *right-side*

where *right-side* describes how to obtain specimens of the *Construct*.

Informative text

The symbol \triangleq may be read aloud as “is defined as”.

BNF-E uses exactly one production for each non-terminal. The reason for this convention is explained below.

End

8.2.7 Kinds of production

A production is of one of the following three kinds, distinguished by the form of the *right-side*:

- **Aggregate**, describing a construct whose specimens are made of a fixed sequence of parts, some of which may be optional.
- **Choice**, describing a construct having a set of given variants.

- **Repetition**, describing a construct whose specimens are made of a variable number of parts, all specimens of a given construct.

8.2.8 Definition: Aggregate production

An **aggregate** right side is of the form $C_1 C_2 \dots C_n$ ($n > 0$), where every one of the C_i is a construct and any contiguous subsequence may appear in square brackets as $[C_i \dots C_j]$ for $1 \leq i \leq j \leq n$.

Every specimen of the corresponding construct consists of a specimen of C_1 , followed by a specimen of C_2 , ..., followed by a specimen of C_n , with the provision that for any subsequence in brackets the corresponding specimens may be absent.

8.2.9 Definition: Choice production

A **choice** right side is of the form $C_1 | C_2 | \dots | C_n$ ($n > 1$), where every one of the C_i is a construct.

Every specimen of the corresponding construct consists of exactly one specimen of one of the C_i .

8.2.10 Definition: Repetition production, separator

A **repetition** right side is of one of the two forms

$\{C \ S \ \dots\}^*$

$\{C \ S \ \dots\}^+$

where C and S (the **separator**) are constructs.

Every specimen of the corresponding construct consists of zero or more (one or more in the second form) specimens of C , each separated from the next, if any, by a specimen of S .

The following abbreviations may be used if the separator is empty:

C^*

C^+

Informative text

The language definition makes only moderate use of recursion thanks to the availability of Repetition productions: when the purpose is simply to describe a construct whose specimens may contain successive specimens of another construct, a Repetition generally gives a clearer picture; see for example the definition of **Compound** as a repetition of **Instruction**. Recursion remains necessary to describe constructs with unbounded nesting possibilities, such as **Conditional** and **Loop**.

End

8.2.11 Basic syntax description rule

Every non-terminal construct is defined by exactly one production.

Informative text

Unlike in most BNF variants, every BNF-E production always uses exactly one of Aggregate, Choice and Repetition, *never* mixing them in the right sides. This convention yields a considerably clearer grammar, even if it has a few more productions (which in the end is good since they give a more accurate image of the language's complexity).

End

8.2.12 Definition: Non-production syntax rule

A **non-production syntax rule**, marked “(non-production)”, is a syntax property expressed outside of the BNF-E formalism.

Informative text

Unlike validity rules, non-production syntax rules belong to the syntax, that is to say the description of the structure of Eiffel texts, but they capture properties that are not expressible, or not conveniently expressible, through a context-free grammar.

For example the BNF-E Aggregate productions allow successive right-side components to be separated by an arbitrary break — any sequence of spaces, tabs and “new line” characters. In a few cases, for example in an *Alias* declaration such as *alias "+"*, it is convenient to use BNF-E — with a right-side listing the keyword *alias*, a double quote, an *Operator* and again a double quote — but we need to *prohibit* breaks between either double quote and the operator. We still use BNF-E to specify such constructs, but add a non-production syntax rule stating the supplementary constraints.

End

8.2.13 Textual conventions

The syntax (BNF-E) productions and other rules of the Standard apply the following conventions:

- 1 Symbols of BNF-E itself, such as the vertical bars | signaling a choice production, appear in black (non-bold, non-italic).
- 2 Any construct name appears in **dark green** (non-bold, non-italic), with a first letter in upper case, as **Class**.
- 3 Any component (Eiffel text element) appears in **blue**.
- 4 The double quote, one of Eiffel's special symbols, appears in productions as `""`: a double quote character (blue like other Eiffel text) enclosed in two single quote characters (black since they belong to BNF-E, not Eiffel).
- 5 All other special symbols appear in double quotes, for example a comma as `","`, an assignment symbol as `":="`, a single quote as `"""` (double quotes black, single quote blue).
- 6 Keywords and other reserved words, such as **class** and **Result**, appear in **bold** (blue like other Eiffel text). They do not require quotes since the conventions avoid ambiguity with construct names: **Class** is the name of a construct, **class** a keyword.
- 7 Examples of Eiffel comment text appear in non-bold, non-italic (and in blue), as `-- A comment`.
- 8 Other elements of Eiffel text, such as entities and feature names (including in comments) appear in non-bold *italic* (blue).

The color-related parts of these conventions do not affect the language definition, which remains unambiguous under black-and-white printing (thanks to the letter-case and font parts of the conventions). Color printing is recommended for readability.

Informative text

Because of the difference between cases 1 and 3, `{` denotes the opening brace as it might appear in an Eiffel class text, whereas `{` is a symbol of the syntax description, used in repetition productions.

In case 2 the use of an upper-case first letter is a consequence of the “Construct Name convention”.

Special symbols are normally enclosed in double quotes (case 5), except for the double quote itself which, to avoid any confusion, appears enclosed in single quotes (case 4). In either variant, the enclosing quotes — double or single respectively — are not part of the symbol.

In some contexts, such as the table of all such symbols, special symbols (cases 4 and 5) appear in bold for emphasis.

In application of cases 7 and 8, occurrences of Eiffel entities or feature names in comments appear in italics, to avoid confusion with other comment text, as in a comment

`-- Update the value of value.`

where the last word denotes a query of name *value* in the enclosing class.

End

8.2.14 Definition: Validity constraint

A **validity constraint** on a construct is a requirement that every syntactically well-formed specimen of the construct must satisfy to be acceptable as part of a software text.

8.2.15 Definition: Valid

A construct specimen, built according to the syntax structure defined by the construct's production, is said to be **valid**, and will be accepted by the language processing tools of any Eiffel environment, if and only if it satisfies the validity constraints, if any, applying to the construct.

8.2.16 Validity: General Validity rule

Validity code: *VBGV*

Every validity constraint relative to a construct is considered to include an implicit supplementary condition stating that every component of the construct satisfies every validity constraint applicable to the component.

8.2.17 Definition: Semantics

The **semantics** of a construct specimen that is syntactically legal and valid is the construct's effect on the execution of a system that includes the specimen.

8.2.18 Definition: Execution terminology

- **Run time** is the period during which a system is executed.
- The **machine** is the combination of hardware (one or more computers) and operating system through which you can execute systems.
- The machine type, that is to say a certain combination of computer type and operating system, is called a **platform**.
- **Language processing tools** serve to build, manipulate, explore and execute the text of an Eiffel system on a machine.

Informative text

The most obvious example of a language processing tool is an Eiffel compiler or interpreter, which you can use to execute a system. But many other tools can manipulate Eiffel texts: Eiffel-aware editors, browsers to explore systems and their properties, documentation tools, debuggers, configuration management systems. Hence the generality of the term "*language processing tool*".

End

8.2.19 Semantics: Case Insensitivity principle

In writing the letters of an **Identifier** serving as name for a class, feature or entity, or a reserved word, using the upper-case or lower-case versions has no effect on the semantics.

Informative text

So you can write a class or feature name as *DOCUMENT*, *document* and even *dOcUmEnT* with exactly the same meaning.

End

8.2.20 Definition: Upper name, lower name

The **upper name** of an **Identifier** or **Operator** *i* is *i* written with all letters in upper case; its **lower name**, *i* with all letters in lower case.

Informative text

In the example the lower name is *document* and the upper name *DOCUMENT*.

The definition is mostly useful for identifiers, but the names of some operators, such as **and** and other boolean operators, also contain letters.

The reason for not letting letter case stand in the way of semantic interpretation is that it is simply too risky to let the meaning of a software text hang on fine nuances of writing, such as changing a letter into its upper-case variant; this can only cause confusion and errors. Different things should, in reliable and maintainable software, have clearly different names.

Letter case is of course significant in “manifest strings”, denoting texts to be taken verbatim, such as error messages or file names.

This letter case policy goes with strong rules on **style**:

- Classes and types should always use the upper name, as with a class *DOCUMENT*.
- Non-constant features and entities should always use the lower name, as with an attribute *document*.
- Constants and “once” functions should use the lower name with the first letter changed to upper, as with a constant attribute *Document*.

End

8.2.21 Syntax (non-production): Semicolon Optionality rule

In writing specimens of **any** construct defined by a Repetition production specifying the semicolon “;” as separator, it is permitted, without any effect on the syntax structure, validity and semantics of the software, to omit any of the semicolons, or to add a semicolon after the last element.

Informative text

This rule applies to instructions, declarations, successive groups of formal arguments, and many other Repetition constructs. It does not rely on the *layout* of the software: Eiffel’s syntax is free-format, so that a return to the next line has the same effect as one or more spaces or any other “break”. Rather than relying on line returns, the Semicolon Optionality rule is ensured by the syntax design of the language, which guarantees that omitting a semicolon never creates an ambiguity.

The rule also guarantees that an extra semicolon at the end, as in *a; b;* instead of just *a; b* is harmless.

The style guidelines suggest omitting semicolons (which would only obscure reading) for successive elements appearing on separate lines, as is usually the case for instructions and declarations, and including them to separate elements on a given line.

Because the semicolon is still formally in the grammar, programmers used to languages where the semicolon is an instruction *terminator*, who may then out of habit add a semicolon after every instruction, will not suffer any adverse effect, and will get the expected meaning.

End

8.3 The architecture of Eiffel software

Informative text

The constituents of Eiffel software are called **classes**. To keep your classes and your development organized, it is convenient to group classes into **clusters**. By combining classes from one or more clusters, you may build executable **systems**.

These three concepts provide the basis for structuring Eiffel software:

- A *class* is a modular unit.
- A *cluster* is a logical grouping of classes.
- A *system* results from the assembly of one or more classes to produce an executable unit.

Of these, only “class”, describing the basic building blocks, corresponds directly to a construct of the language. To build clusters and systems out of classes, you will use not a language mechanism, but tools of the supporting environment.

Clusters provide an intermediate level between classes and systems, indispensable as soon as your systems grow beyond the trivial:

- At one extreme, a cluster may be a simple group of a few classes.
- At the other end, a system as a whole is simply a cluster that you have made executable (by selecting a *root class* and a *root procedure*).
- In-between, a cluster may be a library consisting of several subclusters, or an existing system that you wish to integrate as a subcluster into a larger system.

Clusters also serve to store and group classes using the facilities of the underlying operating system, such as files, folders and directories.

After the basic definitions, the language description will concentrate on classes, indeed the most important concept in the Eiffel method, which views software construction as an industrial production activity: combining components, not writing one-of-a-kind applications.

End

8.3.1 Definition: Cluster, subcluster, contains directly, contains

A **cluster** is a collection of classes, (recursively) other clusters called its **subclusters**, or both. The cluster is said to **contain directly** these classes and subclusters.

A cluster **contains** a class *C* if it contains directly either *C* or a cluster that (recursively) contains *C*.

Informative text

In the presence of subclusters, several clusters may contain a class, but exactly one contains it directly.

End

8.3.2 Definition: Terminal cluster, internal cluster

A cluster is **terminal** if it contains directly at least one class.

A cluster is **internal** if it contains at least one subcluster.

Informative text

From these definitions, it is possible for a cluster to be both terminal and internal.

End

8.3.3 Definition: Universe

A **universe** is a set of classes.

Informative text

The universe provides a reference from which to draw classes of interest for a particular system. Any Eiffel environment will provide a way to specify a universe.

End

8.3.4 Syntax: Class names

Class_name $\hat{=}$ *Identifier*

8.3.5 Validity: Class Name rule

Validity code: *VSCN*

It is valid for a universe to include a class if and only if no other class of the universe has the same upper name.

Informative text

Eiffel expressly does not include a notion of “namespace” as present in some other languages. Experience with these mechanisms shows that they suffer from two limitations:

- They only push forward the problem of class name clashes, turning it into a problem of namespace clashes.

- Even more seriously, they tie a class to a particular context, making it impossible to reorganize (“*refactor*”) the software later without breaking existing code, and hence defeating some of the principal benefits of object technology and modern software engineering.

Name clashes, in the current Eiffel view, should be handled by *tools* of the development environment, enabling application writers to combine classes from many different sources, some possibly with clashing names, and resolving these clashes automatically (with the possibility of registering user preferences and remembering them from one release of an acquired external set of classes to the next) while maintaining clarity, reusability and extensibility.

End

8.3.6 Semantics: Class name semantics

A **Class_name** *C* appearing in the text of a class *D* denotes the class called *C* in the enclosing universe.

8.3.7 Definition: System, root type name, root procedure name

A **system** is defined by the combination of:

- 1 A universe.
- 2 A type name, called the **root type name**.
- 3 A feature name, called the **root procedure name**.

8.3.8 Definition: Type dependency

A type *T* **depends** on a type *R* if any of the following holds:

- 1 *R* is a parent of the base class *C* of *T*.
- 2 *T* is a client of *R*.
- 3 (Recursively) there is a type *S* such that *T* depends on *S* and *S* depends on *R*.

Informative text

This states that *C* depends on *A* if it is connected to *A* directly or indirectly through some combination of the basic relations between types and classes — inheritance and client — studied later. Case 1 relies on the property that every type derives from a class, called its “base class”; for example a generically derived type such as *LIST [INTEGER]* has base class *LIST*. Case 3 gives us indirect forms of dependency, derived from the other cases.

End

8.3.9 Validity: Root Type rule

Validity code: **VSRT**

It is valid to designate a type *TN* as root type of a system of universe *U* if and only if it satisfies the following conditions:

- 1 *TN* is the name of a stand-alone type *T*.
- 2 *T* only involves classes in *U*.
- 3 *T*'s base class is not deferred.
- 4 The base class of any type on which *T* depends is in *U*.

Informative text

These conditions make it possible to create the root object:

- A type is “*stand-alone*” if it only involves class names; this excludes “anchored” types (**like *some_entity***) and formal generic parameters, which only mean something in the context of a particular class text. Clearly, if we want to use a type as root for a system, it must have an absolute meaning, independent of any specific context. “Stand-alone type” is defined at the end of the discussion of types.

- A deferred class is not fully implemented, and so cannot have any direct instances. It wouldn't work as base class here, since the very purpose of a root type is to be instantiated, as the first event of system execution.
- To be able to assemble the system, we must ensure that any class to which the root refers directly or indirectly is also part of the universe.

In condition 2, a type *T* “involves” a class *C* if it is defined in terms of *C*, meaning that *C* is the base class of *T* or of any of its generic parameters: *U*[*V*, *X* [*Y*, *Z*]] involves *U*, *V*, *X*, *Y* and *Z*. If *T* is a non-generic class used as a type, *T* “involves” only itself.

End

8.3.10 Validity: Root Procedure rule

Validity code: *VSRP*

It is valid to specify a name *pn* as root procedure name for a system *S* if and only if it satisfies the following conditions:

- 1 *pn* is the name of a creation procedure *p* of *S*'s root type.
- 2 *p* has no formal argument.
- 3 *p* is precondition-free.

Informative text

A routine is *precondition-free* (condition 3) if it has no precondition, or a precondition that evaluates to true. A routine can impose preconditions on its callers if these callers are other routines; but it makes no sense to impose a precondition on the external agent (person, hardware device, other program...) that triggers an entire system execution, since there is no way to ascertain that such an agent, beyond the system's control, will observe the precondition. Hence the last condition of the rule.

Regarding condition 1, note that a non-deferred class that doesn't explicitly list any creation procedures is understood to have a single one, procedure default_create, which does nothing by default but may be redefined in any class to carry out specific initializations.

End

8.3.11 Definition: Root type, root procedure, root class

In a system *S* of root type name *TN* and root procedure name *pn*, the **root type** is the type of name *TN*, the **root class** is the base class of that root type, and the **root procedure** is the procedure of name *pn* in that class.

8.3.12 Semantics: System execution

To **execute** a system on a machine means to cause the machine to apply a creation instruction to the system's root type.

Informative text

If a routine is a creation procedure of a type used as root of a system, its execution will usually create other objects and call other features on these objects. In other words, the execution of any system is a chain of explosions — creations and calls — each one firing off the next, and the root procedure is the spark that detonates the first step.

End

8.4 Classes

Informative text

Classes are the components used to build Eiffel software.

Classes serve two complementary purposes: they are the modular units of software decomposition; they also provide the basis for the type system of Eiffel.

End

8.4.1 Definition: Current class

The **current class** of a construct specimen is the class in which it appears.

Informative text

Every Eiffel software element — feature, expression, instruction, ... — indeed appears in a class, justifying this definition. Most language properties refer directly or indirectly, through this notion, to the class in which an element belongs.

End

8.4.2 Syntax: Class declarations

Class_declaration \triangleq [Notes]

Class_header

[Formal_generics]

[Obsolete]

[Inheritance]

[Creators]

[Converters]

[Features]

[Invariant]

[Notes]

end

8.4.3 Syntax: Notes

Notes \triangleq **note** Note_list

Note_list \triangleq {Note_entry ";" ...}*

Note_entry \triangleq Note_name Note_values

Note_name \triangleq Identifier ":"

Note_values \triangleq {Note_item ";" ...}+

Note_item \triangleq Identifier | Manifest_constant

Informative text

Notes parts (there may be up to two, one at the beginning and one at the end) have no effect on the execution semantics of the class. They serve to associate information with the class, for use in particular by tools for configuration management, documentation, cataloging, archival, and for retrieving classes based on their properties.

End

8.4.4 Semantics: Notes semantics

A **Notes** part has no effect on system execution.

8.4.5 Syntax: Class headers

Class_header \triangleq [Header_mark] **class** Class_name

Header_mark \triangleq **deferred** | **expanded** | **frozen**

Informative text

The **Class_name** part gives the name of the class. The recommended convention (here and in any context where a class text refers to a class name) is the upper name.

The keyword **class** may optionally be preceded by one of the keywords **deferred**, **expanded** and **frozen**, corresponding to variants of the basic notion of class:

- A **deferred** class describes an incompletely implemented abstraction, which descendants will use as a basis for further refinement.
- Declaring a class as **expanded** indicates that entities declared of the corresponding type will denote objects rather than references to objects.
- Making a class **frozen** prohibits it from serving as “conforming parent” to other classes.

End

8.4.6 Validity: Class Header rule

Validity code: **VCCH**

A **Class_header** appearing in the text of a class **C** is valid if and only if has either no **deferred** feature or a **Header_mark** of the **deferred** form.

Informative text

If a class has at least one deferred feature, either introduced as deferred in the class itself, or inherited as deferred and not “effected” (redeclared in non-deferred form), then its declaration must start not just with **class** but with **deferred class**.

There is no particular rule on the other possible markers, **expanded** and **frozen**, for a **Class_header**. Expanded classes often make the procedure *default_create* available for creation, but this is not a requirement since the corresponding entities may be initialized in other ways; they follow the same rules as other “attached” entities.

End

8.4.7 Definition: Expanded, frozen, deferred, effective class

A class is:

- **Expanded** if its **Class_header** is of the **expanded** form.
- **Frozen** if its **Class_header** is of the **frozen** or **expanded** form.
- **Deferred** if its **Class_header** is of the **deferred** form.
- **Effective** if it is not deferred.

Informative text

Making **C frozen** prohibits it from serving as “conforming parent” to other classes. The second case indicates the two ways of ensuring this:

- Inheritance from expanded classes, as explained in the discussion of inheritance, is non-conforming. As a consequence, any expanded class is also frozen.
- You can explicitly mark a non-expanded class as **frozen**.

The third case defines what makes a class *deferred*:

- If it has at least one deferred feature, the class itself is deferred. The Class Header rule below requires it to be marked **deferred** for clarity.
- If it only has effective features, the class is effective unless you decide otherwise: you can still explicitly mark it **deferred**. This ensures that it will have no direct instances, since one may not apply creation instructions or expressions to a variable whose type is based on a deferred class.

End

8.4.8 Syntax: Obsolete marks

Obsolete \triangleq **obsolete** Message

Message \triangleq Manifest_string

8.4.9 Semantics: Obsolete semantics

Specifying an **Obsolete** mark for a class or feature has no run-time effect.

When encountering such a mark, [language processing tools](#) may issue a report, citing the obsolescence [Message](#) and advising software authors to replace the class or feature by a newer version.

8.5 Features

Informative text

A class is characterized by its features. Every feature describes an operation for accessing or modifying instances of the class.

A feature is either an *attribute*, describing information stored with each instance, or a *routine*, describing an algorithm. Clients of a class *C* may apply *C*'s features to instances of *C* through **call** instructions or expressions.

Every feature has an identifier, which identifies it uniquely in its class. In addition, a feature may have an *alias* to permit calls using operator or bracket syntax.

The following discussion introduces the various categories of feature, explains how to write feature declarations, and describes the form of feature names.

End

8.5.1 Definition: Inherited, immediate; origin; redeclaration; introduce

Any feature *f* of a class *C* is of one of the following two kinds:

- 1 If *C* obtains *f* from one of its [parents](#), *f* is an **inherited** feature of *C*. In this case any declaration of *f* in *C* (adapting the original properties of *f* for *C*) is a **redeclaration**.
- 2 If a declaration appearing in *C* applies to a feature that is not inherited, the feature is said to be **immediate** in *C*. Then *C* is the **origin** (short for "class of origin") of *f*, and is said to **introduce** *f*.

Informative text

A feature redeclaration is a declaration that locally changes an inherited feature. The details of redeclaration appear in the study of inheritance; what is important here is that a declaration in the [Features](#) part only introduces a new feature (called "immediate" in *C*, or "introduced" by *C*) if it is not a redeclaration of some feature obtained from a parent.

Every feature of a class is immediate either in the class or in one of its proper ancestors (parents, grandparents and so on).

End

8.5.2 Syntax: Feature parts

Features \triangleq Feature_clause⁺

Feature_clause \triangleq **feature** [Clients] [Header_comment] Feature_declaration_list

Feature_declaration_list \triangleq {Feature_declaration ";" ...}^{*}

Header_comment \triangleq Comment

Informative text

As part of a general syntactical convention, semicolons are **optional** between a [Feature_declaration](#) and the next. The recommended style rule suggests omitting them except in the infrequent case of two successive declarations on a single line.

End

8.5.3 Feature categories: overview

Every feature of a class is either an *attribute* or a *routine*.

An attribute is either *constant* or *variable*.

A routine is either a *procedure* or a *function*.

Informative text

A set of definitions in the discussion that follows introduces each of these notions precisely, making it possible to recognize, from a valid feature declaration, which kind of feature it introduces.

End

8.5.4 Syntax: Feature declarations

$\text{Feature_declaration} \triangleq \text{New_feature_list Declaration_body}$
 $\text{Declaration_body} \triangleq [\text{Formal_arguments}] [\text{Query_mark}] [\text{Feature_value}]$
 $\text{Query_mark} \triangleq \text{Type_mark} [\text{Assigner_mark}]$
 $\text{Type_mark} \triangleq \text{":"} \text{Type}$
 $\text{Feature_value} \triangleq [\text{Explicit_value}]$
 $[\text{Obsolete}]$
 $[\text{Header_comment}]$
 $[\text{Attribute_or_routine}]$
 $\text{Explicit_value} \triangleq \text{"="} \text{Manifest_constant}$

Informative text

Not all combinations of **Formal_arguments**, **Query_mark** and **Feature_value** are possible; the Feature Body rule and Feature Declaration rule will give the exact constraints. For example it appears from the above syntax that both a **Declaration_body** and a **Feature_value** can be empty, since their right-side components are all optional, but the validity constraints rule this out.

End

8.5.5 Syntax: New feature lists

$\text{New_feature_list} \triangleq \{\text{New_feature} \text{" , " } \dots\}^+$
 $\text{New_feature} \triangleq [\text{frozen}] \text{Extended_feature_name}$

Informative text

Having a list of features, rather than just one, makes it possible for example to declare together several attributes of the same type or, in the case of routines, to introduce several “synonym” routines, with the same body.

End

8.5.6 Syntax: Feature bodies

$\text{Attribute_or_routine} \triangleq [\text{Precondition}]$
 $[\text{Local_declarations}]$
 Feature_body
 $[\text{Postcondition}]$
 $[\text{Rescue}]$
 end
 $\text{Feature_body} \triangleq \text{Deferred} \mid \text{Effective_routine} \mid \text{Attribute}$

8.5.7 Validity: Feature Body rule

Validity code: *VFFB*

A **Feature_value** is valid if and only if it satisfies one of the following conditions:

- 1 It has an **Explicit_value** and no **Attribute_or_routine**.
- 2 It has an **Attribute_or_routine** with a **Feature_body** of the **Attribute** kind.
- 3 It has no **Explicit_value** and has an **Attribute_or_routine** with a **Feature_body** of the **Effective_routine** kind, itself of the **Internal** kind (beginning with **do** or **once**).

- 4 It has no **Explicit_value** and has an **Attribute_or_routine** with neither a **Local_declarations** nor a **Rescue** part, and a **Feature_body** that is either **Deferred** or an **Effective_routine** of the **External** kind.

Informative text

The **Explicit_value** only makes sense for an attribute — either declared explicitly with **Attribute** or simply given a type and a value — so cases 3 and 4 exclude this possibility.

The **Local_declarations** and **Rescue** parts only make sense (case 4) for a feature with an associated algorithm present in the class text itself; this means a routine that is neither deferred nor external, or an attribute with explicit initialization.

In both cases 1 and 2 the feature will be an attribute. Case 1 is a constant attribute declaration such as *n: INTEGER = 100*, with no further details. Case 2 is the long form, explicitly using the keyword **attribute** and making it possible, as with routines, to have a **Precondition**, a **Postcondition**, and even an implementation (including a **Rescue** clause if desired) which will be used, for “self-initializing” types, on first use of an uninitialized field.

The Feature Body rule is the basic validity condition on feature declarations. But for a full view of the constraints we must take into account a set of definitions appearing next, which say what it takes for a feature declaration — already satisfying the Feature Body rule — to belong to one of the relevant categories: *variable attribute*, *constant attribute*, *function*, *procedure*. Another fundamental constraint, the Feature Declaration rule (VFFD), will then require that the feature described by any declaration match one of these categories. So the definitions below are a little more than definitions: they collectively yield a validity requirement complementing the Feature Body rule.

End

8.5.8 Definition: Variable attribute

A **Feature_declaration** is a **variable attribute** declaration if and only if it satisfies the following conditions:

- 1 There is no **Formal_arguments** part.
- 2 There is a **Query_mark** part.
- 3 There is no **Explicit_value** part.
- 4 If there is a **Feature_value** part, it has an **Attribute_or_routine** with a **Feature_body** of the **Attribute** kind.

8.5.9 Definition: Constant attribute

A **Feature_declaration** is a **constant attribute** declaration if and only if it satisfies the following conditions:

- 1 It has no **Formal_arguments** part.
- 2 It has a **Query_mark** part.
- 3 There is a **Feature_value** part including an **Explicit_value**.

8.5.10 Definition: Routine, function, procedure

A **Feature_declaration** is a **routine** declaration if and only if it satisfies the following condition:

- There is a **Feature_value** including an **Attribute_or_routine**, whose **Feature_body** is of the **Deferred** or **Effective_routine** kind.

If a **Query_mark** is present, the routine is a **function**; otherwise it is a **procedure**.

Informative text

For a routine the **Formal_arguments** (like the **Query_mark**) may or may not be present.

By convention this definition treats a deferred feature as a routine, even though its effectings in proper descendants may be, in the case of a query, attributes as well as functions.

End

8.5.11 Definition: Command, query

A **command** is a procedure. A **query** is an attribute or function.

Informative text

These notions underlie two important principles of the Eiffel method:

- The Command-Query separation principle, which suggests that queries should not change objects.
- The Uniform Access principle, which enjoins, whenever possible, to make no distinction between attributes and argumentless functions.

End

8.5.12 Definition: Signature, argument signature of a feature

The **signature** of a feature *f* is a pair *argument_types*, *result_type* where *argument_types* and *result_type* are the following sequences of types:

- For *argument_types*: if *f* is a routine, the possibly empty sequence of its formal argument types, in the order of the arguments; if *f* is an attribute, an empty sequence.
- For *result_type*: if *f* is a query, a one-element sequence, whose element is the type of *f*; if *f* is a procedure, an empty sequence.

The *argument_types* part is the feature's **argument signature**.

Informative text

The argument signature is an empty sequence for attributes and for routines without arguments.

End

8.5.13 Feature principle

Every feature has an associated identifier.

Any valid call (qualified or unqualified) to the feature can be expressed through this identifier.

Informative text

The syntactic variants, available through **alias** clauses, offer other ways to express calls, reconciling object-oriented structure with earlier notations:

- You may qualify the name with **alias "§"** where § is some operator. For example if a feature is named *plus*, clients must call it as *a.plus(b)*; by naming it *plus alias "+"* you still allow this form of calls — per the Feature principle — but you also permit *a + b* in accordance with traditional syntax for arithmetic expressions. The details of alias operators, as well as the associated conversion mechanism, appear next.
- You may also use a "bracket alias", written simply **alias "[]"** (with an opening bracket immediately followed by a closing bracket). This allows access through bracket syntax *x[index]*. For example if a class describing some table structure has a feature *item alias "[]" (index: H): G* where *H* is some index type, items can be accessed through *your_table.item(i)* but also through the more concise *your_table [i]*. Again this is just a syntactic facility: the second form is a synonym for the first, which remains available.

End

8.5.14 Syntax: Feature names

Extended_feature_name \triangleq Feature_name [Alias]

Feature_name \triangleq Identifier

Alias \triangleq `alias` `'` Alias_name `'` [`convert`]

Alias_name \triangleq Operator | Bracket

Bracket \triangleq `"[]"`

Informative text

The optional `convert` mark, for an operator feature, supports mixed-type expressions causing a conversion of the target, as in the expression `your_integer + your_real`, which should use the `+` operation from `REAL`, not `INTEGER`, for compatibility with ordinary arithmetic practice. See the presentation of conversions.

End

8.5.15 Syntax (non-production): Alias Syntax rule

The `Alias_name` of an `Alias` must immediately follow and precede the enclosing double quote symbols, with no intervening characters (in particular no `breaks`).

When appearing in such an `Alias_name`, the two-word operators `and then` and `or else` must be written with exactly one space (but no other characters) between the two words.

Informative text

In general, `breaks` or comment lines may appear between components prescribed by a BNF-E production, making this rule necessary to complement the grammar: you must write `alias " + "`, not `alias " + "`.

End

8.5.16 Definition: Operator feature, bracket feature, identifier-only

A feature is an **operator feature** if its `Extended_feature_name` `fn` includes an `Operator` alias, a **bracket feature** if `fn` includes a `Bracket` alias. It is **identifier-only** if neither of these cases applies.

Informative text

The most common case is identifier-only. The other two kinds provide convenient modes of expression ("*syntactic sugar*") for some cases where a shorter form, compatible with traditional mathematical conventions, is desirable for calling the feature.

When referring to feature names, some syntax rules use the `Extended_feature_name`, and some use the `Feature_name`, which is just the identifier, dropping the `Alias` if any. The criterion is simple: when a class text needs to refer to one of its own features, the `Feature_name` is sufficient since (from the Feature Identifier principle below) it uniquely identifies the feature. So the `Extended_feature_name` is used in only two cases: when you first introduce a feature, in a `Feature_declaration` as discussed above, and when you change its name for a descendant, in a `Rename` clause (for both inheritance and constrained genericity).

This also means that in descendants of its original class a feature will retain its `Alias`, if any, unless a descendant explicitly renames it to a name that may drop the `Alias`, or provide a new one. In particular, redeclaring a feature does not affect its `Alias`.

End

8.5.17 Definition: Identifier of a feature name

The `Identifier` that starts a `Extended_feature_name` is called the **identifier of** that `Extended_feature_name` and, by extension, of the associated feature.

8.5.18 Validity: Feature Identifier principle

Given a class `C` and an identifier `f`, `C` contains at most one feature of identifier `f`.

Informative text

This principle reflects a critical property of object-oriented programming in general and Eiffel in particular: no “*overloading*” of feature names within a class. It is marked as “*validity*” but has no code of its own since it is just a consequence of other validity rules.

End

8.5.19 Definition: Same feature name, same operator, same alias

Two feature names are considered to be “**the same feature name**” if and only if their identifiers have identical lower names.

Two operators are “**the same operator**” if they have identical lower names.

An Alias in an Extended_feature_name is “**the same alias**” as another if and only if they satisfy the following conditions:

- They are either the same Operator or both Bracket.
- If either has a **convert** mark, so does the other.

Informative text

So *my_name*, *MY_NAME* and *mY_nAMe* are considered to be the same feature name. The recommended style uses a name with an initial capital and the rest in lower case (as in *My_name*) for constant attributes, and the lower name, all in lower case (as in *my_name*) for all other features. If letters appear in operator feature names, letter case is also irrelevant when it comes to deciding which feature names are the same and which different.

This notion is useful in particular to enforce the rule that, in any class, there can be only one feature of a given name (no “*overloading*”), and to determine what constitutes a “*name clash*” under multiple inheritance. In such cases the language rules simply ignore letter case.

End

8.5.20 Syntax: Operators

Operator \triangleq Unary | Binary

Unary \triangleq **not** | “+” | “-” | Free_unary

Binary \triangleq “+” | “-” | “*” | “/” | “//” | “\” | “^” | “..” |

“<” | “>” | “<=” | “>=” |

and | **or** | **xor** | **and then** | **or else** | **implies** |

Free_binary

Informative text

Free operators enable developers to define their own operators with considerable latitude. This is particularly useful in scientific applications where it is common to define special notations, which Eiffel will render as prefix or infix operators. You may for example define operators such as ******, **|** (maybe as an infix alias for a *distance* function), or various forms of arrow such as **<->**, **+->**, **=>**.

End

8.5.21 Syntax: Assigner marks

Assigner_mark \triangleq **assign** Feature_name

Informative text

In an assignment *x := v* the target *x* must be a variable. If *item* is an attribute of the type *T* of *a*, programmers used to other languages may be tempted to write an assignment such as *a.item := v* to assign directly to the corresponding object field, but this is not permitted as it goes against all the rules of data abstraction and object technology. The normal mechanism is for the author of the base class of *T* to provide a “*setter*” command (procedure), say *put*, enabling the clients to use *a.put (v)*.

The class author may, for convenience, permit `a.item := v` as a shorthand for this call `a.put(v)`, by specifying `put` as an **assigner command** associated with `item`. An instruction such as `a.item := v` is not an assignment, but simply a different notation for a procedure call; it is known as an **assigner call**. This scheme, a notational simplification only, is also convenient for features that have a **Bracket** alias, allowing for example, with `a` an array, an assigner call `a[i] := v` as shorthand for a call `a.put(v, i)`.

The mechanism is applicable not just to attributes but (in line with the Uniform Access principle) to all queries, including functions with arguments.

The following rule defines under what conditions you may, as author of a class, permit such assigner calls from your clients by associating an assigner command with a query.

End

8.5.22 Validity: Assigner Command rule

Validity code: *VFAC*

An **Assigner_mark** appearing in the declaration of a query `q` with n arguments ($n \geq 0$) and listing a **Feature_name** `fn`, called the **assigner command** for `q`, is valid if and only if it satisfies the following conditions:

- 1 `fn` is the identifier of a command `c` of the class.
- 2 `c` has $n + 1$ arguments.
- 3 The type of `c`'s first argument and the result type of `q` have the same deanchored form.
- 4 For every i in $1..n$, the type of the $i+1$ -st argument of `c` and the type of the i -th argument of `q` have the same deanchored form.

Informative text

The feature `q` can only be a query since, from the syntax of **Declaration_body**, an **Assigner_mark** can only appear as part of a **Query_mark**, whose presence makes the feature a query.

In cases 3 and 4, we require the types (more precisely their deanchored forms, obtained by replacing any anchored type such as **like** `x` by the type of the anchor `x`) to be identical, not just compatible (converting or conforming). To understand why, recall that the assignment `a.item := y` is only a shorthand for a call `a.put(x)` with, as a typical implementation:

```
item: T assign put do ... end
put (b: U) do ... item := b ... end
```

Now assume that `U` is not identical to `T` but only compatible with it, and consider the procedure call

```
a.put (a.item)
```

or the equivalent assignment form

```
a.item := a.item
```

which are in principle useless — they reassign to a field its own value — but should certainly be permitted. They become invalid, however, because the source `a.item` (actual argument of the call or right side of the assignment) is of type `T`, the target (the formal argument) of type `U`, and it's generally impossible for two different types to be each compatible with the other.

This explains clause 3: the first argument of the assigner procedure must have *exactly* the same type as the result of the query (once both have been deanchored). Similar reasoning applied to other arguments, if any, leads to clause 4.

End

8.5.23 Definition: Synonym

A **synonym** of a feature of a class `C` is a feature with a different **Extended_feature_name** such that both names appear in the same **New_feature_list** of a **Feature_declaration** of `C`.

8.5.24 Definition: Unfolded form of a possibly multiple declaration

The **unfolded form** of a **Feature_declaration** listing one or more feature names, as in:

```
f1, f2, ..., fn declaration_body (n ≥ 1)
```

where each f_i is a **New_feature**, is the corresponding sequence of declarations naming only one feature each, and with identical declaration bodies, as in:

```

 $f_1$  declaration_body
 $f_2$  declaration_body
...
 $f_n$  declaration_body

```

Informative text

Thanks to the unfolded form, we may always assume, when studying the validity and semantics of feature declarations, that each declaration applies to only one feature name. This convention is used throughout the language description; to define both the validity and the semantics, it simply refers to the unfolded form, which may give several declarations even if they are all grouped in the class text.

A multiple declaration introduces the feature names as synonyms. But the synonymy only applies to the enclosing class; there is no permanent binding between the corresponding features. Their only relationship is to have the same **Declaration_body** at the point of introduction.

This means in particular that a proper descendant of the class may rename or redeclare one without affecting the other.

Each f_i , being a **New_feature**, may include a **frozen** mark. In the unfolded form this mark only applies to the i -th declaration.

End

8.5.25 Validity: Feature Declaration rule

Validity code: *VFFD*

A **Feature_declaration** appearing in a class C is valid if and only if it satisfies all of the following conditions for every declaration of a feature f in its **unfolded form**:

- 1 The **Declaration_body** describes a feature which, according to the rules given earlier, is one of: **variable attribute**, **constant attribute**, **procedure**, **function**.
- 2 f does not have the **same feature name** as any other feature **introduced** in C (in particular, any other feature of the unfolded form).
- 3 If f has the same feature name as the **final name** of any inherited feature, the **Declaration_body** satisfies the Redeclaration rule.
- 4 If the **Declaration_body** describes a **deferred feature**, then the **Extended_feature_name** of f is not preceded by **frozen**.
- 5 If the **Declaration_body** describes a **once function**, the result type is **stand-alone**.
- 6 Any **anchored type** for an argument is **detachable**.
- 7 The **Alias** clause, if present, is **alias-valid** for f .

Informative text

As stated at the beginning of the rule, the conditions apply to the **unfolded form** of the declaration; this means that the rule treats a multiple declaration f_1, f_2, \dots, f_n **declaration_body** as a succession of n separate declarations with different feature names but the same **declaration_body**.

Conditions 1 and 2 are straightforward: the **Declaration_body** must make sense, and the name or names of the feature being introduced must not conflict with those of any other feature introduced in the class.

In applying conditions 2 and 3, remember that two feature names are “the same” not just if they are written identically, but also if they only differ by letter case. Only the identifiers (**Feature_name**) of the features play a role in this notion, not any **Alias** they may have.

The Feature Name rule will state a consequence of conditions 2 and 3 that may be more appropriate for error messages in some cases of violation.

Condition 4 prohibits a frozen feature from being declared as deferred. The two properties are conceptually incompatible since frozen features, by definition, may not be redeclared, whereas the purpose of deferred features is precisely to force redeclaration in proper descendants.

Condition 5 applies to once functions. A once routine only executes its body on its first call. Further calls have no effect; for a function, they yield the result computed by the first call. This puts a special requirement on the result type T of such a function: if the class is generic, T should not depend on any formal generic parameter, since successive calls could then apply to instances obtained from different generic derivations; and T must not be anchored, as in the context of dynamic binding it could yield incompatible types depending on the type of the target of each particular call. The notion of *stand-alone type* captures these constraints on T .

Condition 6 addresses delicate cases of polymorphism and dynamic binding, where anchored arguments and their implicit form of “covariance” may cause run-time errors known as “catcalls”. It follows from the general rule for signature conformance and is discussed with it.

The last condition, 7, is the consistency requirement on features with an operator or bracket alias. It relies on the following definition (which has a validity code enabling compilers to give more precise error messages).

End

8.5.26 Validity: Alias Validity rule

Validity code: **VFAV**

An *Alias* clause is **alias-valid** for a feature f of a class C if and only if it satisfies the following conditions:

- 1 If it lists an **Operator** op : f is a **query**; no other query of C has an **Operator** alias using the same operator and the same number of arguments; and either: op is a **Unary** and f has no argument, or op is a **Binary** and f has one argument.
- 2 If it lists a **Bracket** alias: f is a query with at least one argument, and no other feature of C has a **Bracket** alias.
- 3 If it includes a **convert** mark: it lists an **Operator** and f has one argument.

Informative text

The first two conditions express the uniqueness and signature requirements on operator and bracket aliases:

- An operator feature **plus alias** “ $\$$ ” can be either unary (called as $\$ a$) or binary (called as $a \$ b$), and so must take either zero or one argument. Two features may indeed share the same alias—like **identity alias** “+” and **plus alias** “+”, respectively unary and binary addition in class **INTEGER** and others from the Kernel Library — as long as they have different identifiers (here **identity** and **plus**) and different signatures, one unary and the other binary.
- A bracket feature, of which there may be at most one in a class, will be called under the form $x [a_1, \dots, a_n]$ with $n \geq 1$, and so must be a query with at least one argument (and hence a function). Condition 2 tells us that there may be at most one bracket feature per class.

Condition 3 indicates that a **convert** mark, specifying “target conversion” as in **your_integer + your_real**, makes sense only for features with one argument, with an **Operator** which (from condition 1) must be a **Binary**.

End

8.6 The inheritance relation

Informative text

Inheritance is one of the most powerful facilities available to software developers. It addresses two key issues of software development, corresponding to the two roles of classes:

- As a **module extension** mechanism, inheritance makes it possible to define new classes from existing ones by adding or adapting features.

- As a **type refinement** mechanism, inheritance supports the definition of new types as specializations of existing ones, and plays a key role in defining the type system.

End

8.6.1 Syntax: Inheritance parts

Inheritance \triangleq **Inherit_clause**⁺
Inherit_clause \triangleq **inherit** [**Non_conformance**] **Parent_list**
Non_conformance \triangleq "{ *NONE* }"
Parent_list \triangleq {**Parent** ";" ...}⁺
Parent \triangleq **Class_type** [**Feature_adaptation**]
Feature_adaptation \triangleq [**Undefine**]
 [**Redefine**]
 [**Rename**]
 [**New_exports**]
 [**Select**]
 end

Informative text

As with all other uses of semicolons, the semicolon separating successive **Parent** parts is optional. The style guidelines suggest omitting it between clauses that appear (as they should) on successive lines.

End

8.6.2 Syntax (non-production): Feature adaptation

A **Feature_adaptation** part must include at least one of the optional components.

Informative text

This rule removes a potential syntax ambiguity by implying that the **end** in **class B inherit A end** closes the class; otherwise it could be understood as closing just the **Parent** part.

End

8.6.3 Definition: Parent part for a type, for a class

If a **Parent** part *p* of an **Inheritance** part lists a **Class_type** *T*, *p* is said to be a **Parent part for T**, and also for the base class of *T*.

Informative text

So in **inherit TREE [T]** there is a **Parent** part for the type **TREE [T]** and for its base class **TREE**. For convenience this definition, like those for “parent” and “heir” below, applies to both types and classes.

End

Informative text

An important property of the inheritance structure is that every class inherits, directly or indirectly, from a class called **ANY**, of which a version is provided in the Kernel Library as required by the next rule. The semantics of the language depends on the presence of such a class, whether the library version or one that a programmer has provided as a replacement.

End

8.6.4 Validity: Class **ANY** rule

Validity code: *VHCA*

Every system must include a non-generic class called **ANY**.

Informative text

The key property of **ANY** is that it is not only an ancestor of all classes and hence types, but that all types **conform** to it, according to the following principle, which is not a separate validity rule (although for reference it has a code of its own) but a consequence of the definitions and rules below.

End

8.6.5 Validity: Universal Conformance principle

Validity code: *VHUC*

Every type conforms to **ANY**.

Informative text

To achieve the Universal Conformance principle, the semantics of the language guarantees that a class that doesn't list any explicit **Parent** is considered to have **ANY** as its parent. This is captured by the following notion: **Unfolded Inheritance Part**. The above definition of "parent", and through it the definition of "ancestor", refer to the **Unfolded Inheritance Part** of a class rather than its actual **Inheritance** part.

End

8.6.6 Definition: Unfolded Inheritance Part of a class

Any class **C** has an **Unfolded Inheritance Part** defined as follows:

- 1 If **C** has an **Inheritance** part: that part.
- 2 Otherwise: an **Inheritance** part of the form **inherit ANY**.

8.6.7 Definition: Multiple, single inheritance

A class has **multiple inheritance** if it has an **Unfolded Inheritance Part** with two or more **Parent** parts. It has **single inheritance** otherwise.

Informative text

What counts for this definition is the number not of parent classes but of **Parent** parts. If two clauses refer to the same parent class, this is still a case of multiple inheritance, known as **repeated inheritance** and studied later on its own. If there is no **Parent** part, the class (as will be seen below) has a de facto parent anyway, the Kernel Library class **ANY**.

The definition refers to the "Unfolded" inheritance part which is usually just the **Inheritance** part but may take into account implicit inheritance from **ANY**, as detailed in the corresponding definition below.

Multiple inheritance is a frequent occurrence in Eiffel development; most of the effective classes in the widely used EiffelBase library of data structures and algorithms, for example, have two or more parents. The widespread view that multiple inheritance is "bad" or "dangerous" is not justified; most of the time, it results from experience with imperfect multiple inheritance mechanisms, or improper uses of inheritance. Well-applied multiple and repeated inheritance is a powerful way to combine abstractions, and a key technique of object-oriented software development.

End

8.6.8 Definition: Inherits, heir, parent

A class **C inherits** from a type or class **B** if and only if **C**'s **Unfolded Inheritance Part** contains a **Parent part** for **B**.

B is then a **parent** of **C** ("parent type" or "parent class" if there is any ambiguity), and **C** an **heir** (or "heir class") of **B**. Any type of **base class C** is also an heir of **B** ("heir type" in case of ambiguity).

8.6.9 Definition: Conforming, non-conforming parent

A parent *B* in an *Inheritance* part is **non-conforming** if and only if every *Parent part for B* in the clause appears in an *Inherit_clause* with a *Non_conformance* marker. It is **conforming** otherwise.

8.6.10 Definition: Ancestor types of a type, of a class

The **ancestor types** of a type *CT* of base class *C* include:

- 1 *CT* itself.
- 2 (Recursively) The result of applying *CT*'s *generic substitution* to the ancestor types of every *parent type for C*.

The ancestor types of a *class* are the ancestor types of its *current type*.

Informative text

The basic definition covers ancestor types of a *type*; the second part of the definition extends this notion to classes.

Case 1 indicates that a type is its own ancestor.

Case 2, the recursive case, applies the notion of *generic substitution* introduced in the discussion of genericity. The idea that if we consider the type *C [INTEGER]*, with the class declaration **class C [G] inherit D [G] ...**, the type to include in the ancestors of *C [INTEGER]* as a result of this *Inheritance* part is not *D [G]*, which makes no sense outside of the text of *C*, but *D [INTEGER]*, the result of applying to *D [G]* the substitution $G \rightarrow \text{INTEGER}$; this is the substitution that yields the type *C [INTEGER]* from the class *C [G]* and is known as the generic substitution of that type.

End

8.6.11 Definition: Ancestor, descendant

Class *A* is an **ancestor** of class *B* if and only if *A* is the *base class* of an *ancestor type* of *B*.

Class *B* is a **descendant** of class *A* if and only if *A* is an ancestor of *B*.

Informative text

Any class, then, is both one of its own descendants and one of its own ancestors. *Proper* descendants and ancestors exclude these cases.

End

8.6.12 Definition: Proper ancestor, proper descendant

The **proper ancestors** of a class *C* are its *ancestors* other than *C* itself. The **proper descendants** of a class *B* are its *descendants* other than *B* itself.

8.6.13 Validity: Parent rule

Validity code: *VHPR*

The *Unfolded Inheritance Part* of a class *D* is valid if and only if it satisfies the following conditions:

- 1 In every *Parent part* for a class *B*, *B* is not a *descendant* of *D*.
- 2 No *conforming parent* is a *frozen class*.
- 3 If two or more *Parent* parts are for classes which have a common ancestor *A*, *D* meets the conditions of the *Repeated Inheritance Consistency constraint* for *A*.
- 4 At least one of the *Parent* parts is *conforming*.
- 5 No two ancestor types of *D* are different *generic derivations* of the same class.
- 6 Every *Parent* is *generic-creation-ready*.

Informative text

Condition 1 ensures that there are no cycles in the inheritance relation.

The purpose of declaring a class as **frozen** (case 2) is to prohibit subtyping. We still permit the *non-conforming* form of inheritance, which permits reuse but not subtyping.

Condition 3 corresponds to the case of repeated inheritance; the Repeated Inheritance Consistency constraint will guarantee that there is no ambiguity on features that *D* inherits repeatedly from *A*.

Condition 4 ensures a central property of the type system: the Universal Conformance principle, stating that all types conform to *ANY*. Without this condition, it would be possible for all *Parent* parts of a class to be non-conforming and hence to cause violation of the principle. Note that in the Unfolded Inheritance Part there is always at least one *Parent* part, since the absence of an *Inheritance* part is a shorthand for *inherit ANY*, ensuring that condition 4 holds.

Condition 5 avoids various cases of ambiguity which could arise if we allowed a class *C* to inherit from both *A*[*T*] and *A*[*U*] for different *T* and *U*. For example, if *C* redefines a feature *f* from *A*, the notation *Precursor* {*A*} in the redefinition could refer to either of the parents' generic derivations.

Condition 6 also concerns the case of a generically derived *Parent* *A* [*T*]; requiring it to be "generic-creation-ready" guarantees that creation operations on *D* or its descendants will function properly if they need to create objects of type *T*

End

8.6.14 Syntax: Rename clauses

Rename \triangleq *rename* *Rename_list*

Rename_list \triangleq {*Rename_pair* ", " ...}⁺

Rename_pair \triangleq *Feature_name* **as** *Extended_feature_name*

Informative text

The first component of a *Rename_pair* is just a *Feature_name*, the identifier for the feature; the second part is a full *Extended_feature_name*, which may include an *alias* clause. Indeed:

- To identify the feature you are renaming, its *Feature_name* suffices.
- At the same time you are renaming the feature, you may give it a new operator or bracket alias, or remove the alias if it had one.

Forms of feature adaptation other than renaming, in particular effecting and redefinition, do not affect the *Alias*, if any, associated with a *Feature_name*.

End

8.6.15 Validity: Rename Clause rule

Validity code: *VHRC*

A *Rename_pair* of the form *old_name as new_name*, appearing in the *Rename* subclause of the *Parent* part for *B* in a class *C*, is valid if and only if it satisfies the following conditions:

- 1 *old_name* is the final_name of a feature *f* of *B*.
- 2 *old_name* does not appear as the first element of any other *Rename_pair* in the same *Rename* subclause.
- 3 *new_name* satisfies the Feature Name rule for *C*.
- 4 The *Alias* of *new_name*, if present, is alias-valid for the version of *f* in *C*.

Informative text

In condition 4, the "alias-valid" condition captures the signature properties allowing a query to have an operator or bracket aliases. It was enforced when we wanted to give a feature an alias in the first place and, naturally, we encounter it again when we give it an alias through renaming.

End

8.6.16 Semantics: Renaming principle

Renaming does not affect the semantics of an inherited feature.

Informative text

The “positive” semantics of renaming (as opposed to the negative observation captured by this principle) follows from the definition of *final name* and *extended final name* of a feature below.

End

8.6.17 Definition: Final name, extended final name, final name set

Every feature *f* of a class *C* has an **extended final name** in *C*, an *Extended_feature_name*, and a **final name**, a *Feature_name*, defined as follows:

- 1 The final name is the identifier of the extended final name.
- 2 If *f* is immediate in *C*, its extended final name is the *Extended_feature_name* under which *C* declares it.
- 3 If *f* is inherited, *f* is obtained from a feature of a parent *B* of *C*. Let *extended_parent_name* be (recursively) the extended final name of that feature in *B*, and *parent_name* its final name of *f* in *B*. Then the extended final name of *f* in *C* is:
 - If the *Parent* part for *B* in *C* contains a *Rename_pair* of the form **rename *parent_name* as *new_name*: *new_name***.
 - Otherwise: *extended_parent_name*.

The final names of all the features of a class constitute the **final name set** of a class.

Informative text

Since an inherited feature may be obtained from two or more parent features, case 3 only makes sense if they are all inherited under the same name. This will follow from the final definition of “inherited feature” in the discussion of repeated inheritance.

The extended final name is an *Extended_feature_name*, possibly including an *Alias* part; the final name is its identifier only, a *Feature_name*, without the alias. The recursive definition defines the two together.

End

8.6.18 Definition: Inherited name

The **inherited name** of a feature obtained from a feature *f* of a parent *B* is the final name of *f* in *B*.

Informative text

In the rest of the language description, references to the “name” of a feature, if not further qualified, always denote the final name.

End

8.6.19 Definition: Declaration for a feature

A *Feature_declaration* in a class *C*, listing a *Feature_name* *fn*, is a **declaration for** a feature *f* if and only if *fn* is the final name of *f* in *C*.

Informative text

Although it may seem almost tautological, we need this definition so that we can talk about a declaration “for” a feature *f* whether *f* is immediate — in which case *fn* is just the name given in its declaration — or inherited, with possible renaming. This will be useful in particular when we look at a *redeclaration*, which overrides a version inherited from a parent.

End

8.7 Clients and exports

Informative text

Along with inheritance, the client relation is one of the basic mechanisms for structuring software. In broad terms, a class *C* is a client of a type *S* — which is then a *supplier* of *C* — when it can manipulate objects of type *S* and apply *S*'s features to them.

The simplest and most common way is for *C* to contain the declaration of an entity of type *S*.

Variants of the relation introduce similar dependencies through other mechanisms, in particular generic parameters.

Although the original definitions introduce “client” in its various forms as a relation between a class and a type, we’ll immediately extend it, by considering *S*'s base class, to a relation between classes.

It is useful to distinguish between several variants of the client relation: simple client, expanded client and generic client relations. Each is studied below. The more general notion of client is the union of these cases, according to the following definition.

End

8.7.1 Definition: Client relation between classes and types

A class *C* is a **client** of a type *S* if some ancestor of *C* is a simple client, an expanded client or a generic client of *S*.

Informative text

Recall that the ancestors of *C* include *C* itself. The definition involves all of *C*'s ancestors to include dependencies caused by inherited features along with those due to the immediate features of *C*. Assume that an inherited routine *r* of *C* uses a local variable *x* of type *S*; this means that *C* may depend on *S* even if the text of *C* does not mention *S*. (If *C* redefines *r*, the definition may then needlessly make *C* a client of *S*, but this has no harmful consequences.)

8.7.2 Definition: Client relation between classes

A class *C* is a **client of a class** *B* if and only if *C* is a client of a type whose base class is *B*.

The same convention applies to the simple client, expanded client and generic client relations.

8.7.3 Definition: Supplier

A type or class *S* is a **supplier** of a class *C* if *C* is a client of *S*, with corresponding variants: simple, expanded, generic, indirect.

8.7.4 Definition: Simple client

A class *C* is a **simple client** of a type *S* if, in *C*, *S* is the type of some entity or expression or the Explicit_creation_type of a Creation_instruction, or is one of the Constraining_types of a formal generic parameter of *C*, or is involved in the Type of a Non_object_call or of a Manifest_type.

Informative text

The constructs listed reflect the various ways in which a class may, by listing a type *S* in its text, enable itself to use features of *S* on targets of type *S*.

No constraint restricts how the classes of a system may be simple clients of one another. In particular, cycles are permitted: a class may be its own simple client, both directly according to this definition and indirectly.

End

8.7.5 Definition: Expanded client

A class *C* is an **expanded client** of a type *S* if *S* is an expanded type and some attribute of *C* is of type *S*.

8.7.6 Definition: Generic client, generic supplier

A class C is a **generic client** of a type S if for some generically derived type T of the form $B[... , S, ...]$ one of the following holds:

- 1 C is a client of T .
- 2 T is a parent type of an ancestor of C .

Informative text

Case 1 captures for example the use in C of an entity of type $B[S]$ (with B having just one generic parameter). Case 2 covers C inheriting directly or indirectly (remember that C is one of its own ancestors) from $B[S]$.

End

8.7.7 Definition: Indirect client

A class A is an **indirect client** of a type S of base class B if there is a sequence of classes $C_1 = A, C_2, \dots, C_n = B$ such that $n > 2$ and every C_i is a client of C_{i+1} for $1 \leq i < n$.

The indirect forms of the simple client, expanded client and generic client relations are defined similarly.

8.7.8 Definition: Client set of a Clients part

The **client set** of a **Clients** part is the set of descendants of every class of the universe whose name it lists.

By convention, the client set of an absent **Clients** part includes all classes of the system.

Informative text

The descendants of a class include the class itself. The “convention” of this definition simplifies the following definitions in the case of no **Clients** part, which should be treated as if there were a **Clients** part listing just *ANY*, ancestor of all classes.

No validity rule prevents listing in a **Clients** part a name n that does not denote a class of the universe. In this case — explicitly permitted by the phrasing of the definition — n does not denote any class and hence has no descendants; it does not contribute to the client set.

This important convention is in line with the reuse focus of Eiffel and its application to component-based development. You may develop a class C in a certain system, where it lists some class S in a **Clients** part, to give S access to some of its features; then you reuse C in another system that does not include S . You should not have to change C since no bad consequence can result from listing a class not present in the system, as long as C does not itself use S as its supplier or ancestor.

Even in a single system, this policy means that you can remove S — if you find it is no longer needed — without causing compilation errors in the classes that list it in their **Clients** parts. With a stricter rule, you would have to remove S from every such **Clients** part. But then if you later change your mind — as part of the normal hesitations of an incremental design process — you would have to put it back in each of these places. This process is tedious, and it wouldn’t take many iterations until programmers start making many features public just in case — hardly an improvement for information hiding, the purpose of all this.

End

8.7.9 Syntax: Clients

Clients \triangleq "{" Class_list "}"

Class_list \triangleq {Class_name " , " ...}+

Informative text

There is **no validity constraint** on **Clients** part. In particular, it is valid for a **Clients** part both:

- To list a class that does not belong to the universe.

- To list a class twice.

End

End

8.7.10 Syntax: Export adaptation

$\text{New_exports} \triangleq \text{export New_export_list}$

$\text{New_export_list} \triangleq \{\text{New_export_item } ";" \dots\}^+$

$\text{New_export_item} \triangleq \text{Clients} [\text{Header_comment}] \text{Feature_set}$

$\text{Feature_set} \triangleq \text{Feature_list} \mid \text{all}$

$\text{Feature_list} \triangleq \{\text{Feature_name } ";" \dots\}^+$

8.7.11 Validity: Export List rule

Validity code: *VLEL*

A *New_exports* clause appearing in class *C* in a Parent part for a parent *B*, of the form

export

{class_list₁} feature_set₁

...

{class_list_n} feature_set_n

is valid if and only if for every *feature_set_i* (for *i* in the interval *1..n*) that is a *Feature_list* (rather than **all**):

- 1 Every element of the list is the final name of a feature of *C* inherited from *B*.
- 2 No feature name appears more than once in any such list.

Informative text

To obtain the export status of a feature, we need to look at the *Feature_clause* that introduces it if it is immediate, at the applicable *New_exports* clause, if any, if it is inherited, and at the *Feature_clause* containing its redeclaration if it is inherited and redeclared. In a *New_exports*, the keyword **all** means that the chosen status will apply to all the features inherited from the given parent.

The following definitions and rules express these properties. They start by extending the notion of “client set” from entire *Clients* parts to individual features.

End

8.7.12 Definition: Client set of a feature

The **client set** of a feature *f* of a class *C*, of final name *fname*, includes the following classes (for all cases that match):

- 1 If *f* is introduced or redeclared in *C*: the client set of the *Feature_clause* of the declaration for *f* in *C*.
- 2 If *f* is inherited: the union of the client sets (recursively) of all its precursors from conforming parents.
- 3 If the *Feature_set* of one or more *New_exports* clauses of *C* includes *fname* or **all**, the union of the client sets of their *Clients* parts.

Informative text

This definition is the principal rule for determining the export status of a feature. It has two important properties:

- The different cases are cumulative rather than exclusive. For example a “redeclared” feature (case 1) is also “inherited” (case 2) and the applicable *Parent part* may have a *New_exports* (case 3).

- As a result of case 2, **the client set can never diminish under conforming inheritance**: features can win new clients, but never lose one. This is necessary under polymorphism and dynamic binding to avoid certain type of “catcalls” leading to run-time crashes.

End

8.7.13 Definition: Available for call, available

A feature *f* is **available for call**, or just **available** for short, to a class *C* or to a type based on *C*, if and only if *C* belongs to the client set of *f*.

Informative text

In line with others in the present discussion, the definition of “available for call” introduces a notion about *classes* and immediately generalizes it to *types* based on those classes.

The key validity constraint on calls, export validity, will express that a call *a.f(...)* can only be valid if *f* is available to the type of *a*.

There is also a notion of “available for creation”, governing whether a *Creation_call* **create** *a.f(...)* is valid. “Available” without further qualification means “available for call”.

There are three degrees of availability, as given by the following definition.

End

8.7.14 Definition: Exported, selectively available, secret

The export status of a feature of a class is one of the following:

- 1 The feature may be available to all classes. It is said to be **exported**, or **generally available**.
- 2 The feature may be available to specific classes (other than *NONE* and *ANY*) only. In that case it is also available to the descendants of all these classes. Such a feature is said to be **selectively available** to the given classes and their descendants.
- 3 Otherwise the feature is available only to *NONE*. It is then said to be **secret**.

Informative text

This is the fundamental terminology for information hiding, which determines when it is possible to call a feature through a *qualified call* *x.f*. As special cases:

- A feature introduced by **feature** *{NONE}* (case 3) is available to no useful classes.
- A feature introduced by **feature** *{ANY}*, or just **feature**, is available to all classes and so will be considered to fall under case 1.
- A feature introduced by **feature** *{A, B, C}*, where none of *{A, B, C}* is *ANY*, falls under case 2.

A feature available to a class is also available to all the proper descendants of that class. As a consequence, selective export does not restrict reuse as much as it may seem at first: while the features will only be available to certain classes, these may be classes written much later, as long as they are descendants of one of the listed *Clients*.

End

8.7.15 Definition: Secret, public

A property of a class text is **secret** if and only if it involves any of the following, describing information on which client classes cannot rely to establish their correctness:

- 1 Any feature that is not available to the given client, unless this is overridden by the next case.
- 2 Any feature that is not available for creation to the given client, unless this is overridden by the previous case.
- 3 The body and rescue clause of any feature, except for the information that the feature is external or *Once* and, in the last case, its *once* keys if any.

- 4 For a query without formal arguments, whether it is implemented as an attribute or a function, except for the information that it is a constant attribute.
- 5 Any Assertion_clause that (recursively) includes secret information.
- 6 Any parent part for a non-conforming parent (and as a consequence the very presence of that parent).
- 7 The information that a feature is frozen.

Any property of a class text that is not secret is **public**.

Informative text

Software developers must be able to use a class as supplier on the basis of public information only.

A feature may be available for call, or for creation, or both (cases 1 and 2). If either of these properties applies, the affected clients must know about the feature, even if they can use it in only one of these two ways.

Whether a feature is external (case 3) or constant (case 4) determines whether it is possible to use it in a Non_object_call and hence is public information.

End

8.7.16 Definition: Incremental contract view, short form

The **incremental contract view** of a class, also called its **short form**, is a text with the same structure as the class but retaining only public properties.

Informative text

Eiffel environments usually provide tools that automatically produce the incremental contract view of a class from the class text. This provides the principal form of software documentation: abstract yet precise, and extracted from the program text rather than written and maintained separately.

The definition specifies the information that the incremental contract view must retain, but not its exact display format, which typically will be close to Eiffel syntax.

End

8.7.17 Definition: Contract view, flat-short form

The **contract view** of a class, also called its **flat-short form**, is a text following the same conventions as the incremental contract view form but extended to include information about inherited as well as immediate features, the resulting combined preconditions and postconditions and the unfolded form of the class invariant including inherited clauses.

Informative text

The contract view is the full interface information about a class, including everything that clients need to know (but no more) to use it properly. The “combined forms” of preconditions and postconditions take into account parents’ versions as possibly modified by **require else** and **ensure then** clauses, and hence describing features’ contracts as they must appear to the clients. The “unfolded form” of the class invariant includes clauses from parents. In all these, of course, we still eliminate any clause that includes secret information, as with the incremental contract view.

The contract view is the principal means of documenting Eiffel software, in particular libraries of reusable components. It provides the right mix of abstraction, clarity and precision, and excludes implementation-dependent properties. Being produced automatically by software tools from the actual text, it does not require extra effort on the part of software developers, and stands a much better chance to remain accurate when the software changes.

End

8.8 Routines

Informative text

Routines describe computations.

Syntactically, routines are one of the two kinds of feature of a class; the other kind is attributes, which describe data fields associated with instances of the class. Since every Eiffel operation applies to a specific object, a routine of a class describes a computation applicable to instances of that class. When applied to an instance, a routine may query or update some or all fields of the instance, corresponding to attributes of the class.

A routine is either a procedure, which does not return a result, or a function, which does. A routine may further be declared as **deferred**, meaning that the class introducing it only gives its specification, leaving it for descendants to provide implementations. A routine that is not deferred is said to be **effective**.

An effective routine has a **body**, which describes the computation to be performed by the routine. A body is a **Compound**, or sequence of instructions; each instruction is a step of the computation. The present discussion explores the structure of routine declarations, ending with the list of possible various forms of instructions.

End

8.8.1 Definition: Formal argument, actual argument

Entities declared in a routine to represent information passed by callers are the routine's **formal arguments**.

The corresponding expressions in a particular call to the routine are the call's **actual arguments**.

Informative text

Rules on **Call** require the number of actual arguments to be the same as the number of formal arguments, and the type of each actual argument to be compatible with (conform or convert to) the type of the formal argument at the same position in the list.

A note on terminology: Eiffel always uses the term **argument** to refer to the arguments of a routine. The word "parameter" is never used in this context, because it could create confusion with the types that can parameterize *classes*, called **generic parameters**.

End

8.8.2 Syntax: Formal argument and entity declarations

Formal_arguments \triangleq "(" Entity_declaration_list ")"

Entity_declaration_list \triangleq {Entity_declaration_group ";" ...}⁺

Entity_declaration_group \triangleq Identifier_list Type_mark

Identifier_list \triangleq {Identifier ";" ...}⁺

Informative text

As with other semicolons, those separating an **Entity_declaration_group** from the next are optional. The style guidelines suggest including them for successive declarations on a line, as with short formal argument lists, but omitting them between successive lines, as with local variable declarations (also covered by **Entity_declaration_group**).

End

8.8.3 Validity: Formal Argument rule

Validity code: *VRFA*

Let *fa* be the **Formal_arguments** part of a routine *r* in a class *C*. Let *formals* be the concatenation of every **Identifier_list** of every **Entity_declaration_group** in *fa*. Then *fa* is valid if and only if no **Identifier** *e* appearing in *formals* is the final name of a feature of *C*.

Informative text

Another rule, given later, applies the same conditions to names of local variables. Permitting a formal argument or local variable to bear the same name as a feature could only cause confusion (even if we had a scoping rule removing any ambiguity by specifying that the local name overrides the feature name) and serves no useful purpose.

End

8.8.4 Validity: Entity Declaration rule

Validity code: *VRED*

Let *el* be an *Entity_declaration_list*. Let *identifiers* be the concatenation of every *Identifier_list* of every *Entity_declaration_group* in *el*. Then *el* is valid if and only if no *Identifier* appears more than once in the list *identifiers*.

8.8.5 Syntax: Routine bodies

Deferred \triangleq *deferred*

Effective_routine \triangleq *Internal* | *External*

Internal \triangleq *Routine_mark* *Compound*

Routine_mark \triangleq *do* | *Once*

Once \triangleq *once* [("*Key_list*")]

Key_list \triangleq {*Manifest_string* ", ..."}⁺

8.8.6 Definition: Once routine, once procedure, once function

A **once routine** is an *Internal* routine *r* with a *Routine_mark* of the *Once* form.

If *r* is a *procedure* it is also a **once procedure**; if *r* is a *function*, it is also a **once function**.

8.8.7 Syntax: Local variable declarations

Local_declarations \triangleq *local* [*Entity_declaration_list*]

8.8.8 Validity: Local Variable rule

Validity code: *VRLV*

Let *ld* be the *Local_declarations* part of a routine *r* in a class *C*. Let *locals* be the concatenation of every *Identifier_list* of every *Entity_declaration_group* in *ld*. Then *ld* is valid if and only if every *Identifier* *e* in *locals* satisfies the following conditions:

- 1 No feature of *C* has *e* as its *final name*.
- 2 No formal argument of *r* has *e* as its *Identifier*.

Informative text

Most of the rules governing the validity and semantics of declared local variables also apply to a special predefined entity: **Result**, which may only appear in a function or attribute, and denotes the value to be returned by the function. The following definition of "local variable" reflects this similarity.

End

8.8.9 Definition: Local variable

The local variables of a routine include all *entities* declared in its *Local_declarations* part, if any, and, if it is a query, the predefined entity **Result**.

Informative text

Result can appear not only in the **Compound**, **Postcondition** or **Rescue** of a function or variable attribute but also in the optional **Postcondition** of a constant attribute, where it denotes the value of the attribute and allows stating abstract properties of that value, for example after a redefinition. In this case execution cannot change that value, but for simplicity we continue to call **Result** a local “variable” anyway.

End

8.8.10 Syntax: Instructions

Compound \triangleq {**Instruction** ";" ...}*

Instruction \triangleq **Creation_instruction** | **Call** | **Assignment** | **Assigner_call** | **Conditional** | **Multi_branch** | **Loop** | **Debug** | **Precursor** | **Check** | **Retry**

Informative text

A **Compound** is a possibly empty list of instructions, to be executed in the order given. In the various parts of control structures, such as the branches of a **Conditional** or the body of a **Loop**, the syntax never specifies **Instruction** but always **Compound**, so that you can include zero, one or more instructions.

A **Creation_instruction** creates a new object, initializes its fields to default values, calls on it one of the creation procedures of the class (if any), and attaches the object to an entity.

Call executes a routine. For the **Call** to yield an instruction, the routine must be a procedure.

Assignment changes the value attached to a variable.

An **Assigner_call** is a procedure call written with an assignment-like syntax, as in $x.a := b$, but with the semantics of a call, as just a notational abbreviation for $x.set_a(b)$ where the declaration of a specifies an assigner command set_a .

Conditional, **Multi_branch**, **Loop** and **Compound** describe control structures, made out of instructions; to execute a control structure is to execute some or all of its constituent instructions, according to a schedule specified by the control structure.

Debug, which may also be considered a control structure, is used for instructions that should only be part of the system when you enable the *debug* compilation option.

Precursor enables you, in redefining a routine, to rely on its original implementation.

Check is used to express that certain assertions must hold at certain moments during run time.

Retry is used in conjunction with the exception handling mechanism.

End

8.9 Correctness and contracts

Informative text

Eiffel software texts — classes and their routines — may be equipped with elements of formal specification, called **assertions**, expressing correctness conditions.

Assertions play several roles: they help in the production of correct and robust software, yield high-level documentation, provide debugging support, allow effective software testing, and serve as a basis for exception handling. With advances in formal methods technology, they open the way to proofs of software correctness.

Assertions are at the basis of the **Design by Contract** method of Eiffel software construction.

End

8.9.1 Syntax: Assertions

Precondition \triangleq **require** [**else**] **Assertion**

Postcondition \triangleq **ensure** [**then**] **Assertion** [**Only**]

Invariant \triangleq **invariant** Assertion
 Assertion \triangleq {Assertion_clause ";" ...}^{*}
 Assertion_clause \triangleq [Tag_mark] Unlabeled_assertion_clause
 Unlabeled_assertion_clause \triangleq Boolean_expression | Comment
 Tag_mark \triangleq Tag ":"
 Tag \triangleq Identifier

8.9.2 Syntax (non-production): Assertion Syntax rule

An **Assertion** without a **Tag_mark** may not begin with any of the following:

- 1 An opening parenthesis "(".
- 2 An opening bracket "[".
- 3 A non-keyword **Unary** operator that is also **Binary**.

Informative text

This rule participates in the achievement of the general Semicolon Optionality rule. Without it, after an **Assertion_clause** starting for example with the **Identifier** *a*, and continuing (case 2) with [*x*] it is not immediately obvious whether this is the continuation of the same clause, using *a* [*x*] as the application of a bracket feature to *a*, or a new clause that starts by mentioning the **Manifest_tuple** [*x*]. From the context, the validity rules will exclude one of these possibilities, but a language processing tool should be able to parse an Eiffel text without recourse to non-syntactic information. A similar issue arises with an opening parenthesis (case 1) and also (case 3) if what follows *a* is *-b*, which could express a subtraction from *a* in the same clause, or start a new clause about the negated value of *b*. The Assertion Syntax rule avoids this.

The rule does significantly restrict expressiveness, since violations are rare and will be flagged clearly in reference to the rule, and it is recommended practice anyway to use a **Tag_mark**, which removes any ambiguity.

End

8.9.3 Definition: Precondition, postcondition, invariant

The **precondition** and **postcondition** of a feature, or the **invariant** of a class, is the **Assertion** of, respectively, the corresponding **Precondition**, **Postcondition** or **Invariant** clause if present and non-empty, and otherwise the assertion **True**.

Informative text

So in these three contexts we consider any absent or empty assertion clause as the assertion **True**, satisfied by every state of the computation. Then we can talk, under any circumstance, of "the precondition of a feature" and "the invariant of a class" even if the clauses do not appear explicitly.

End

8.9.4 Definition: Contract, subcontract

Let *pre* and *post* be the precondition and postcondition of a feature *f*. The **contract** of *f* is the pair of assertions [*pre*, *post*].

A contract [*pre'*, *post'*] is said to be a **subcontract** of [*pre*, *post*] if and only if *pre* implies *pre'* and *post'* implies *post*.

8.9.5 Validity: Precondition Export rule

Validity code: *VAPE*

A **Precondition** of a feature *r* of a class *S* is valid if and only if every feature *f* appearing in every **Assertion_clause** of its **unfolded form** *u* satisfies the following two conditions for every class *C* to which *r* is available:

- 1 If *f* appears as feature of a call in *u* or any of its subexpressions, *f* is available to *C*.

- 2 If u or any of its subexpressions uses f as creation procedure of a **Creation_expression**, f is available for creation to C .

Informative text

If (condition 1) r were available to a class B but its precondition involved a feature f not available to B , r would be imposing to B a condition that B would not be able to check for itself; this would amount to a secret clause in the contract, preventing the designer of B from guaranteeing the correctness of calls.

The rule applies to the *unfolded form* of a precondition, which will be defined as the fully reconstructed assertion, including conditions defined by ancestor versions of a feature in addition to those explicitly mentioned in a redeclared version.

The unfolded form (by relying on the “Equivalent Dot Form” of the expressions involved) treats all operators as denoting features; for example an occurrence of $a > b$ in an assertion yields $a.greater(b)$ in the unfolded form, where *greater* is the name of a feature of alias “>”. The Precondition Export rule then requires, if the occurrence is in a **Precondition**, that this feature be available to any classes to which the enclosing feature is available.

Condition 2 places the same obligation on any feature f used in a creation expression **create a.f (...)** appearing in the precondition (a rare but possible case). The requirement in this case is “available for creation”.

End

8.9.6 Definition: Availability of an assertion clause

An **Assertion_clause** a of a routine **Precondition** or **Postcondition** is **available** to a class B if and only if all the features involved in the Equivalent Dot Form of a are available to B .

Informative text

This notion is necessary to define interface forms of a class adapted to individual clients, such as the incremental contract view (“short form”).

End

8.9.7 Syntax: “Old” postcondition expressions

Old \triangleq **old Expression**

8.9.8 Validity: Old Expression rule

Validity code: **VAOX**

An **Old** expression oe of the form **old e** is valid if and only if it satisfies the following conditions:

- 1 It appears in a **Postcondition** part *post* of a feature.
- 2 It does not involve **Result**.
- 3 Replacing oe by e in *post* yields a valid **Postcondition**.

Informative text

Result is otherwise permitted in postconditions, but condition 2 rules it out since its value is meaningless on entry to the routine. Condition 3 simply states that **old e** is valid in a postcondition if e itself is. The expression e may not, for example, involve any local variables (although it might include **Result** were it not for condition 2), but may refer to features of the class and formal arguments of the routine.

End

8.9.9 Semantics: Old Expression Semantics, associated variable, associated exception marker

The effect of including an **Old** expression oe in a **Postcondition** of an effective feature f is equivalent to replacing the semantics of its **Feature_body** by the effect of a call to a fictitious routine possessing a local variable av , called the **associated variable** of oe , and semantics defined by the following succession of steps:

- 1 Evaluate *oe*.
- 2 If this evaluation triggers an exception, record this event in an **associated exception marker** for *oe*.
- 3 Otherwise, assign the value of *oe* to *av*.
- 4 Proceed with the original semantics.

Informative text

The recourse to a fictitious variable and fictitious operations is in the style of “unfolded forms” used throughout the language description. The reason for these techniques is the somewhat peculiar nature of the **Old** expression, used at postcondition evaluation time, but pre-computed (if assertion monitoring is on for postconditions) on entry to the feature.

The matter of exceptions is particularly delicate and justifies the use of “associated exception markers”. If an **Old** expression’s evaluation triggers an exception, the time of that exception — feature entry — is not the right moment to start handling the exception, because the postcondition might not need the value. For example, a postcondition clause could read

$((x \neq 0) \text{ and } (\text{old } x \neq 0)) \text{ implies } (((1 / x) + (1 / (\text{old } x)))) = y$

If *x* is 0 on entry, **old** *x* \neq 0 will be false on exit and hence the postcondition will hold. But there is no way to know this when evaluating the various **Old** expressions, such as **1 / old** *x* on entry. We must evaluate this expression anyway, to be prepared for all possible cases. If *x* is zero, this may cause an arithmetic overflow and trigger an exception. This exception should not be processed immediately; instead it should be remembered — hence the associated exception marker — and triggered only if the evaluation of the *postcondition*, on routine exit, attempts to evaluate the associated variable; hence the following rule.

The “associated variable” is defined only for effective features, since a deferred feature has no **Feature_body**. If an **Old** expression appears in the postcondition of a deferred feature, the rule will apply to effectings in descendants through the “unfolded form” of the postconditions, which includes inherited clauses.

Like any variable, the associated variable *av* of an **Old** expression raises a potential initialization problem; but we need not require its type to be self-initializing since the above rule implies that *av* appears in a Certified Attachment Pattern that assigns it a value (the value of *oe*) prior to use.

End

8.9.10 Semantics: Associated Variable Semantics

As part of the evaluation of a postcondition clause, the evaluation of the associated variable of an **Old** expression:

- 1 Triggers an exception of type **OLD_EXCEPTION** if an associated exception marker has been recorded.
- 2 Otherwise, yields the value to which the variable has been set.

8.9.11 Syntax: “Only” postcondition clauses

Only \triangleq **only** [**Feature_list**]

Informative text

The syntax of assertions indicates that an **Only** clause may only appear in a **Postcondition** of a feature, as its last clause.

Those other postcondition clauses let you specify how a feature *may* change specific properties of the target object, as expressed by queries. You may also want — this is called the **frame problem** — to restrict the scope of features by specifying which properties it *may not* change. You can always do this through postcondition clauses **q = old** *q*, one for each applicable query *q*. This is inconvenient, not only because there may be many such *q* to list but also, worse, because it forces you to list them all even though evolution of the software may bring in some new queries, which will not be listed. Inheritance makes matters even more delicate since such “frame” requirements of parents should be passed on to heirs.

An **Only** clause addresses the issue by enabling you to list which queries a feature may affect, with the implication that:

- Any query *not* listed is left unchanged by the feature.
- The constraints apply not only to the given version of the feature but also, as enforced by the following rules, to any redeclarations in descendants (specifically, to their effect on the queries of the original class).

The syntax allows omitting the **Feature_list**; this is how you can specify that the routine must leave *all* queries unchanged (it is then known as a “*pure*” routine).

End

8.9.12 Validity: Only Clause rule

Validity code: *VAON*

An **Only** clause appearing in a **Postcondition** of a feature of a class **C** is valid if and only if every **Feature_name** *qn* appearing its **Feature_list** if any satisfies the following conditions:

- 1 There is no other occurrence of *qn* in that **Feature_list**.
- 2 *qn* is the final name of a query *q* of **C**, with no arguments.
- 3 If **C** redeclares *f* from a parent **B**, *q* is not a feature of **B**.

Informative text

Another condition, following from the syntax, is that an **Only** clause appears at the last element of a **Postcondition**; in particular, you may not include more than one **Only** clause in a **postcondition**.

End

8.9.13 Definition: Unfolded feature list of an Only clause

The **unfolded feature list** of an **Only** clause appearing in a **Postcondition** of a feature *f* in a class **C** is the **Feature_list** containing:

- 1 All the feature names appearing in its **Feature_list** if any.
- 2 If *f* is the redeclaration of one or more features, the final names in **C** of all the features whose names appear (recursively) in their unfolded Only clauses.

Informative text

For an immediate feature (a feature introduced in **C**, not a redeclaration), the purpose of an **Only** clause of the form

only *q, r, s*

is to state that *f* may only change the values of queries *q, r, s*.

In the case of a redeclaration, previous versions may have had their own **Only** clauses. Then:

- If there was already an **Only** clause in an ancestor **A**, the features listed, here *q, r* and *s*, must be new features, not present in **A**. Otherwise specifying **only** *q, r, s* would either contradict the **Only** clause of **A** if it did not include these features (thus ruling out any modification to them in any descendant), or be redundant with it if it listed any one of them.
- The meaning of the **Only** clause is that *f* may only change *q, r* and *s* *in addition* to inherited queries that earlier **Only** clauses allowed it to change.

Note that this definition is mutually recursive with the next one.

End

8.9.14 Definition: Unfolded Only clause

The **unfolded Only clause** of a feature *f* of a class **C** is a sequence of **Assertion_clause** components of the following form, one for every argument-less query *q* of **C** that does not appear in the unfolded feature list of the **Only** clause of its **Postcondition** if any:

q = (old q)

Informative text

This will make it possible to express the semantics of an **Only** clause through a sequence of assertion clauses stating that the feature may change the value of no queries except those explicitly listed.

Note the use of the equal sign: for a query q returning a reference, the **Only** clause states (by *not* including q) that after the feature's execution the reference will be attached to the same object as before. That object might, internally, have changed. You can still rule out such changes by listing in the **Only** clause other queries reflecting properties of the object's *contents*.

End

8.9.15 Definition: Hoare triple notation (total correctness)

In definitions of correctness notions for Eiffel constructs, the notation $\{P\} A \{Q\}$ (a mathematical convention, not a part of Eiffel) expresses that any execution of the **Instruction** or **Compound A** started in a state of the computation satisfying the assertion P will terminate in a state satisfying the assertion Q .

8.9.16 Semantics: Class consistency

A class C is **consistent** if and only if it satisfies the following conditions:

- 1 For every creation procedure p of C :

$\{pre_p\} do_p \{INV_C \text{ and then } post_p\}$

- 2 For every feature f of C exported generally or selectively:

$\{INV_C \text{ and then } pre_f\} do_f \{INV_C \text{ and then } post_f\}$

where INV_C is the invariant of C and, for any feature f , pre_f is the unfolded form of the precondition of f , $post_f$ the unfolded form of its postcondition, and do_f its body.

Informative text

Class consistency is one of the most important aspects of the *correctness* of a class: adequation of routine implementations to the specification. The other aspects of correctness, studied below, involve **Check** instructions, **Loop** instructions and **Rescue** clauses.

End

8.9.17 Syntax: Check instructions

Check \triangleq **check** Assertion [Notes] **end**

Informative text

The **Notes** part is intended for expressing a formal or informal justification of the assumption behind the property being asserted.

End

8.9.18 Definition: Check-correct

An *effective* routine r is **check-correct** if, for every **Check** instruction c in r , any execution of c (as part of an execution of r) satisfies its **Assertion**.

8.9.19 Syntax: Variants

Variant \triangleq **variant** [Tag_mark] Expression

8.9.20 Validity: Variant Expression rule

Validity code: **VAVE**

A **Variant** is valid if and only if its variant expression is of type **INTEGER** or one of its sized variants.

8.9.21 Definition: Loop invariant and variant

The **Assertion** introduced by the **Invariant** clause of a loop is called its **loop invariant**. The **Expression** introduced by the **Variant** clause is called its **loop variant**.

8.9.22 Definition: Loop-correct

A routine is **loop-correct** if every loop it contains, with loop invariant *INV*, loop variant *VAR*, Initialization *INIT*, Exit condition *EXIT* and body (Compound part of the Loop_body) *BODY*, satisfies the following conditions:

- 1 {true} *INIT* {*INV*}
- 2 {true} *INIT* {*VAR* ≥ 0}
- 3 {*INV* and then not *EXIT*} *BODY* {*INV*}
- 4 {*INV* and then not *EXIT* and then (*VAR* = *v*)} *BODY* {0 ≤ *VAR* < *v*}

Informative text

Conditions 1 and 2 express that the initialization yields a state in which the invariant is satisfied and the variant is non-negative. Conditions 3 and 4 express that the body, when executed in a state where the invariant is satisfied but not the exit condition, will preserve the invariant and decrease the variant, while keeping it non-negative. (*v* is an auxiliary variable used to refer to the value of *VAR* before *BODY*'s execution.)

End

8.9.23 Definition: Correctness (class)

A class is **correct** if and only if it is consistent and every routine of the class is check-correct, loop-correct and exception-correct.

8.9.24 Definition: Local unfolded form of an assertion

The **local unfolded form** of an assertion *a* — a Boolean expression — is the Equivalent Dot Form of the expression that would be obtained by applying the following transformations to *a* in order:

- 1 Replace any Only clause by the corresponding unfolded Only clause.
- 2 Replace any Old expression by its associated variable.
- 3 Replace any clause of the Comment form by True.

Informative text

The unfolded form enables you to understand an assertion, possibly with many clauses, as a single boolean expression. The use of **and then** to separate the clauses indicates that you may, in a later clause, use an expression that is defined only if an earlier clause holds (has value true). This unfolded form is "local" because it does not take into account any inherited assertion clauses. This is the business of the full (non-local) notion of unfolded form of an assertion, introduced in the discussion of redeclaration.

The Equivalent Dot Form of an expression removes all operators and replaces them by explicit call, turning for example *a + b* into *a.plus (b)*. This puts the result in a simpler form used by later rules.

If an Only clause is present, we replace it by its own unfolded form, a sequence of Assertion_clause components of the form *q = old q*, so that we can treat it like other clauses for the assertion's local unfolded form. Note that this unfolding only takes into account queries explicitly listed in the Only clause, but not in any Only clause from an ancestor version; inheritance aspects are handled by the normal unfolding of postconditions, applicable after this one according (as noted above) to the general notion of unfolded form of an assertion

The syntax permits a Comment as Unlabeled_assertion_clause. Such clauses are useful for clarity and documentation but, as reflected by condition 3, cannot have any effect on run-time monitoring.

End

8.9.25 Semantics: Evaluation of an assertion

To **evaluate** an assertion consists of computing the value of its unfolded form.

Informative text

This defines the value of an assertion in terms of the value of a boolean expression, as given by the discussion of expressions.

End

8.9.26 Semantics: Assertion monitoring

The execution of an Eiffel system may evaluate, or **monitor**, specific kinds of assertion, and loop variants, at specific stages:

- 1 Precondition of a routine *r*: on starting a call to *r*, after argument evaluation and prior to executing any of the instructions in *r*'s body.
- 2 Postcondition of a routine *r*: on successful (not interrupted by an exception) completion of a call to *r*, after executing any applicable instructions of *r*.
- 3 Invariant of a class *C*: on both start and termination of a *qualified* call to a routine of *C*.
- 4 Invariant of a loop: after execution of the **Initialization**, and after every execution (if any) of the **Loop_body**.
- 5 Assertion in a **Check** instruction: on any execution of that instruction.
- 6 Variant of a loop: as with the loop invariant.

8.9.27 Semantics: Assertion violation

An **assertion violation** is the occurrence at run time, as a result of assertion monitoring, of any of the following:

- An assertion (in the strict sense of the term) evaluating to false.
- A loop variant found to be negative.
- A loop variant found, after the execution of a **Loop_body**, to be no less than in its previous evaluation.

Informative text

To simplify the discussion these cases are all called "*assertion violations*" even though a variant is not technically an assertion.

End

8.9.28 Semantics: Assertion semantics

In the absence of assertion violations, assertions have no effect on system execution other than through their evaluation as a result of assertion monitoring.

An assertion violation causes an exception of type **ASSERTION_VIOLATION** or one of its descendants.

8.9.29 Semantics: Assertion monitoring levels

An Eiffel implementation must provide facilities to enable or disable assertion monitoring according to some combinations of the following criteria:

- Statically (at compile time) or dynamically (at run time).
- Through control information specified within the Eiffel text or through outside elements such as a user interface or configuration files.
- For specific kinds as listed in the definition of assertion monitoring: routine preconditions, routine postconditions, class invariants, loop invariants, **Check** instructions, loop variants.
- For specific classes, specific clusters, or the entire system.

The following combinations must be supported:

- 1 Statically disable all monitoring for the entire system.
- 2 Statically enable precondition monitoring for an entire system.

- 3 Statically enable precondition monitoring for specified classes.
- 4 Statically enable all assertion monitoring for an entire system.

8.10 Feature adaptation

Informative text

A key attraction of the inheritance mechanism is that it lets you tune inherited features to the context of the new class. This is known as feature adaptation. The present discussion covers the principal mechanisms, leaving to a later one some important complements related to repeated inheritance.

End

8.10.1 Definition: Redeclare, redeclaration

A class **redeclares** an inherited feature if it redefines or effects it.

A declaration for a feature *f* is a **redeclaration** of *f* if it is either a redefinition or an effecting of *f*.

Informative text

This definition relies on two others, appearing below, for the two cases: *redefinition* and *effecting*. Be sure to distinguish *redeclaration* from *redefinition*, the first of these cases. Redeclaration is the more general notion, redefinition one of its two cases; the other is *effecting*, which provides an implementation for a feature that was deferred in the parent. In both cases, a redeclaration does not introduce a new feature, but simply overrides the parent's version of an inherited feature.

End

8.10.2 Definition: Unfolded form of an assertion

The **unfolded form** of an assertion *a* of local unfolded form *ua* in a class *C* is the following Boolean_expression:

- 1 If *a* is the invariant of *C* and *C* has *n* parents for some $n \geq 1$: *up₁* **and ... and** *up_n* **and then** *ua*, where *up₁*, ..., *up_n* are (recursively) the unfolded forms of the invariants of these parents, after application of any feature renaming specified by *C*'s corresponding Parent clauses.
- 2 If *a* is the precondition of a redeclared feature *f*: the combined precondition for *a*.
- 3 If *a* is the postcondition of a redeclared feature *f*: the combined postcondition for *a*.
- 4 In all other cases: *ua*.

Informative text

The unfolded form of an assertion is the form that will define its semantics. It takes into account not only the assertion as written in the class, but also any applicable property inherited from the parent. The "local unfolded form" is the expression deduced from the assertion in the class itself; for an invariant we "and then" it with the "and" of the parents, and for preconditions and postconditions we use "combined forms", defined next, to integrate the effect of **require else** and **ensure then** clauses, to ensure that things will still work as expected in the context of polymorphism and dynamic binding.

The earlier definitions enable us to talk about the "precondition of" and "postcondition of" a feature and the "invariant of" even in the absence of explicit clauses, by using **True** in such cases. This explains in particular why case 1 can mention "the invariants of" the parents of *C*.

End

8.10.3 Definition: Assertion extensions

For a feature *f* of a class *C*:

- If C redeclares f with a non-empty **Precondition** (starting with **require else**), the **precondition extension** of f in C is the corresponding **Assertion**.
- If C redeclares f with a non-empty **Postcondition** (starting with **ensure then**), the **postcondition extension** of f in C is the corresponding **Assertion**.

In all other cases, the precondition extension of f in C is **False** and the postcondition extension of f in C is **True**.

Informative text

These are the forms that routines can use to override inherited specifications while remaining compatible with the original contracts for polymorphism and dynamic binding. **require else** makes it possible to weaken a precondition, **ensure then** to strengthen a postcondition, under the exact interpretation explained next.

End

8.10.4 Definition: Covariance-aware form of an assertion extension

The **covariance-aware form** of an inherited assertion a is:

- 1 If the enclosing routine has one or more arguments x_1, \dots, x_n redefined covariantly to types U_1, \dots, U_n ; the assertion
 $((x_1: U_1) y_1 \text{ and } \dots \text{ and } (x_n: U_n) y_n) \text{ and then } a'$
 where y_1, \dots, y_n are fresh names and a' is the result of substituting y_i for each corresponding x_i in a .
- 2 Otherwise: a .

Informative text

A covariant redefinition may make some of the new clauses inapplicable to actual arguments of the old type (leading to “catcalls”). The covariance-aware form avoids this by ignoring the clauses that are not applicable. The rule on covariant redefinition avoid any bad consequences.

End

8.10.5 Definition: Combined precondition, postcondition

Consider a feature f redeclared in a class C . Let f_1, \dots, f_n ($n \geq 1$) be its versions in parents, pre_1, \dots, pre_n the covariance-aware forms of (recursively) the combined preconditions of these versions, and $post_1, \dots, post_n$ the covariance-aware forms of (recursively) their combined postconditions.

Let pre be the precondition extension of f if defined and not empty, otherwise **False**.

Let $post$ be the postcondition extension of f if defined and not empty, otherwise **True**.

The **combined precondition** of f is the **Assertion**

$(pre_1 \text{ or } \dots \text{ or } pre_n) \text{ or else } pre$

The **combined postcondition** of f is the **Assertion**

$(\text{old } pre_1 \text{ implies } post_1)$

and ... **and**

$(\text{old } pre_n \text{ implies } post_n)$

and then $post$

Informative text

The informal rule is “perform an *or* of the preconditions and an *and* of the postconditions”. This indeed the definition for “combined precondition”. For “combined postconditions” the informal rule is sufficient in most cases, but occasionally it may be too strong because it requires the old postconditions even in cases that do *not* satisfy the old preconditions, and hence only need the new postcondition. The combined postcondition as defined reflects this property.

End

8.10.6 Definition: Inherited as effective, inherited as deferred

An inherited feature is **inherited as effective** if it has at least one precursor that is an effective feature, and the corresponding Parent part does not undefine it.

Otherwise the feature is **inherited as deferred**.

8.10.7 Definition: Effect, effecting

A class **effects** an inherited feature *f* if and only if it inherits *f* as deferred and contains a declaration for *f* that defines an effective feature.

Informative text

Effecting a feature (making it *effective*, hence the terminology) consists of providing an implementation for a feature that was inherited as deferred. No particular clause (such as **redefine**) will appear in the Inheritance part: the new implementation will without ado subsume the deferred form inherited from the parent.

End

8.10.8 Definition: Redefine, redefinition

A class **redefines** an inherited feature *f* if and only if it contains a declaration for *f* that is not an effecting of *f*.

Such a declaration is then known as a **redefinition of *f***

Informative text

Redefining a feature consists of providing a new implementation, specification or both. The applicable Parent clause or clauses must specify **redefine *f*** (with *f*'s original name if the new class renames *f*.)

Redefinition must keep the inherited status, deferred or effective, of *f*.

- It cannot turn a deferred feature into an effective one, as this would fall be an effecting.
- It may not turn an effective feature into a deferred one, as there is another mechanism specifically for this purpose, *undefinition*. The Redeclaration rule enforces this property.

As defined earlier, the two cases, effecting and redefinition, are together called *redeclaration*.

End

8.10.9 Definition: Name clash

A class has a **name clash** if it inherits two or more features from different parents under the same final name.

Informative text

Since final names include the identifier part only, aliases if any play no role in this definition.

Name clashes would usually render the class invalid. Only three cases may — as detailed by the validity rules — make a name clash permissible:

- At most one of the clashing features is effective.
- The class redefines all the clashing features into a common version.

- The clashing features are really the same feature, inherited without redeclaration from a common ancestor.

End

8.10.10 Syntax: Precursor

Precursor \triangleq **Precursor** [Parent_qualification] [Actuals]

Parent_qualification \triangleq "{" Class_name "}"

8.10.11 Definition: Relative unfolded form of a Precursor

In a class *C*, consider a **Precursor** specimen *p* appearing in the redefinition of a routine *r* inherited from a parent class *B*. Its **unfolded form relative to B** is an **Unqualified_call** of the form *r'* if *p* has no **Actuals**, or *r'*(*args*) if *p* has actual arguments *args*, where *r'* is a fictitious feature name added, with a **frozen** mark, as synonym for *r* in *B*.

8.10.12 Validity: Precursor rule

Validity code: *VDPR*

A **Precursor** is valid if and only if it satisfies the following conditions:

- 1 It appears in the **Feature_body** of a **Feature_declaration** of a feature *f*.
- 2 If the **Parent_qualification** part is present, its **Class_name** is the name of a parent class *P* of *C*.
- 3 Among the features of *C*'s parents, limited to features of *P* if condition 2 applies, exactly one is an effective feature redefined by *C* into *f*. (The class to which this feature belongs is called the **applicable parent** of the **Precursor**.)
- 4 The unfolded form relative to the applicable parent is, as an **Unqualified_call**, argument-valid.

In addition:

- 5 It is valid as an **Instruction** if and only if *f* is a command, and as an **Expression** if and only if *f* is a query.

Informative text

This constraint also serves, in condition 3, as a definition of the “applicable parent”: the parent from which we reuse the implementation. Condition 4 relies on this notion.

Condition 1 states that the **Precursor** construct is only valid in a routine redefinition. In general the language definition treats functions and attributes equally (*Uniform Access* principle), but here an attribute would not be permissible, even with an **Attribute** body.

Because of our interpretation of a multiple declaration as a set of separate declarations, this means that if **Precursor** appears in the body of a multiple declaration it applies separately to every feature being redeclared. This is an unlikely case, and this rule makes it unlikely to be valid.

Condition 2 states that if you include a class name, as in **Precursor {B}**, then *B* must be the name of one of the parents of the current class. The following condition makes this qualified form compulsory in case of potential ambiguity, but even in the absence of ambiguity you may use it to state the parent explicitly if you think this improves readability.

Condition 3 specifies when this explicit parent qualification is required. This is whenever an ambiguity could arise because the redefinition applies to more than one effective parent version. The phrasing takes care of all the cases in which this could happen, for example as a result of a join.

Condition 4 simply expresses that we understand the **Precursor** specimen as a call to a frozen version of the original routine; we must make sure that such a call would be valid, more precisely “argument-valid”, the requirement applicable to such an **Unqualified_call**.

A **Precursor** will be used as either an **Instruction** or an **Expression**, in the same way as a call to (respectively) a procedure or a function; indeed **Precursor** appears as one of the syntax variants for both of these constructs. So in addition to being valid on its own, it must be valid in the appropriate role. Condition 5 takes care of this.

End

8.10.13 Definition: Unfolded form of a Precursor

The **unfolded form** (absolute) of a valid **Precursor** is its unfolded form relative to its applicable parent.

8.10.14 Semantics: Precursor semantics

The effect of a **Precursor** is the effect of its unfolded form.

8.10.15 Syntax: Redefinition

Redefine \triangleq **redefine** *Feature_list*

8.10.16 Validity: Redefine Subclause rule

Validity code: *VDRS*

A **Redefine** subclause appearing in a Parent part for a class **B** in a class **C** is valid if and only if every **Feature_name** *fname* that it lists (in its **Feature_list**) satisfies the following conditions:

- 1 *fname* is the final name of a feature *f* of **B**.
- 2 *f* was not frozen in **B**, and was not a constant attribute.
- 3 *fname* appears only once in the **Feature_list**.
- 4 The **Features** part of **C** contains one **Feature_declaration** that is a redeclaration but not an effecting of *f*.
- 5 If that redeclaration specifies a deferred feature, **C** inherits *f* as deferred.

8.10.17 Semantics: Redefinition semantics

The effect in a class **C** of redefining a feature *f* in a Parent part for **A** is that the version of *f* in **C** is, rather than its version in **A**, the feature described by the applicable declaration in **C**.

Informative text

This new version will serve for any use of the feature in the class, its clients, its proper descendants (barring further redeclarations), and even ancestors and their clients under dynamic binding.

End

8.10.18 Syntax: Undefine clauses

Undefine \triangleq **undefine** *Feature_list*

8.10.19 Validity: Undefine Subclause rule

Validity code: *VDUS*

An **Undefine** subclause appearing in a Parent part for a class **B** in a class **C** is valid if and only if every **Feature_name** *fname* that it lists (in its **Feature_list**) satisfies the following conditions:

- 1 *fname* is the final name of a feature *f* of **B**.
- 2 *f* was not frozen in **B**, and was not an attribute.
- 3 *f* was effective in **B**.
- 4 *fname* appears only once in the **Feature_list**.
- 5 Any redeclaration of *f* in **C** specifies a deferred feature.

8.10.20 Semantics: Undefinition semantics

The effect in a class **C** of undefining a feature *f* in a Parent part for **A** is to cause **C** to inherit from **A**, rather than the version of *f* in **A**, a deferred form of that version.

8.10.21 Definition: Effective, deferred feature

A feature *f* of a class *C* is an **effective feature** of *C* if and only if it satisfies either of the following conditions:

- 1 *C* contains a declaration for *f* whose **Feature_body** is not of the **Deferred** form.
- 2 *f* is an **inherited feature**, coming from a **parent** *B* of *C* where it is (recursively) effective, and *C* does not undefine it.

f is **deferred** if and only if it is not effective.

Informative text

As a result of this definition, a feature is deferred in *C* not only if it is introduced or redefined in *C* as deferred, but also if its precursor was deferred and *C* does not redeclare it effectively. In the latter case, the feature is “inherited as deferred”.

The definition captures the semantics of deferred features and of their effecting. In case 1 it's clear that the feature is effective, since *C* itself declares it as either an attribute of a non-deferred routine. In case 2 the feature is inherited; it was already effective in the parent, and *C* doesn't change that status.

End

8.10.22 Definition: Effecting

A redeclaration into an **effective feature** of a feature **inherited as deferred** is said to **effect** that feature.

8.10.23 Deferred class property

A class that has at least one **deferred feature** must have a **Class_header** starting with the keyword **deferred**. The class is then said to be **deferred**.

8.10.24 Effective class property

A class whose features, if any, are all effective, is effective unless its **Class_header** starts with the keyword **deferred**.

Informative text

It is not an error to declare a class **deferred** if it has no deferred features; the effect is simply that clients are not able to create direct instances. It is indeed sometimes useful to introduce a class that cannot be directly instantiated; for example the designer may intend the class to be used only through inheritance. The technique to achieve this is simply to state the abstract nature of the class by declaring it **deferred** even if all its features are effective.

End

8.10.25 Definition: Origin, seed

Every feature *f* of a class *C* has one or more features known as its **seeds** and one or more classes known as its **origins**, as follows:

- 1 If *f* is **immediate** in *C*: *f* itself as seed; *C* as an origin.
- 2 If *f* is **inherited**: (recursively) all the seeds and origins of its **precursors**.

Informative text

The origin, a class, is “where the feature comes from”, and the seed is the version of the feature from that origin. In the vast majority of cases this is all there is to know. With repeated inheritance and “join”, a feature may result from the merging of two or more features, and hence may have more than one seed and more than one origin. That's what case 2 is about.

End

8.10.26 Validity: Redeclaration rule

Validity code: *VDRD*

Let *C* be a class and *g* a feature of *C*. It is valid for *g* to be a redeclaration of a feature *f* inherited from a parent *B* of *C* if and only if the following conditions are satisfied.

- 1 No effective feature of *C* other than *f* and *g* has the same final name.
- 2 The signature of *g* conforms to the signature of *f*.
- 3 The Precondition of *g*, if any, begins with **require else** (not just **require**), and its Postcondition, if any, begins with **ensure then** (not just **ensure**).
- 4 If the redeclaration is a redefinition (rather than an effecting) the Redefine subclause of the Parent part for *B* lists in its Feature_list the final name of *f* in *B*.
- 5 If *f* is inherited as effective, then *g* is also effective.
- 6 If *f* is an attribute, *g* is an attribute, *f* and *g* are both variable, and their types are either both expanded or both non-expanded.
- 7 *f* and *g* have either both no alias or the same alias.
- 8 If both features are queries with associated assigner commands *fp* and *gp*, then *gp* is the version of *fp* in *C*.

Informative text

Condition 1 prohibits name clashes between effective features. For *g* to be a redeclaration of *f*, both features must have the same final name; but no other feature of the class may share that name. This is the fundamental rule of **no overloading**.

No invalidity results, however, if *f* is deferred. Then if *g* is also deferred, the redeclaration is simply a redefinition of a deferred feature by another (to change the signature or specification). If *g* is effective, the redeclaration is an effecting of *f*. If *g* plays this role for more than one inherited *f*, it both joins and effects these features: this is the case in which *C* kills several deferred birds with one effective stone.

Condition 2 is the fundamental type compatibility rule: signature conformance. In the case of a join, *g* may be the redeclaration of more than one *f*; then *g*'s signature must conform to all of the precursors' signatures.

Signature conformance permits *covariant* redefinition of both query results and routine arguments, but for arguments you must make the new type detachable — *?U* rather than just *U* — to prevent “catcalls”.

Condition 3 requires adapting the assertions of a redeclared feature, as governed by rules given earlier.

Condition 4 requires listing *f* in the appropriate Redefine subclause, but only for a redefinition, not for an effecting. (We have a redefinition only if *g* and the inherited form of *f* are both deferred or both effective.) If two or more features inherited as deferred are joined and then redefined together, **every one of them** must appear in the Redefine subclause for the corresponding parent.

Condition 5 bars the use of redeclaration for turning an effective feature into a deferred one. This is because a specific mechanism is available for that purpose: undefinition. It is possible to apply both undefinition and redefinition to the same feature to make it deferred and at the same time change its signature.

Condition 6 prohibits redeclaring a constant attribute, or redeclaring a variable attribute into a function or constant attribute. It also precludes redeclaring a (variable) attribute of an expanded type into one of reference type or conversely. You may, however, redeclare a function into an attribute — variable or constant.

Condition 7 requires the features, if they have aliases, to have the same ones. If you want to introduce an alias for an inherited feature, change an inherited alias, or remove it, redeclaration is not the appropriate technique: you must rename the feature. Of course you can still redeclare it as well.

Condition 8 applies to assigner commands. It is valid for a redeclaration to include an assigner command if the precursor did not include one, or conversely; but if both versions of the query have assigner commands, they must, for obvious reasons of consistency, be the same procedure in *C*.

End

8.10.27 Definition: Precursor (joined features)

A **precursor** of an inherited feature is a version of the feature in the parent from which it is inherited.

8.10.28 Definition: Transposition to a class or type

The **transposition** to a class *C* of a specimen *s* appearing in a ancestor *A* of *C* is the specimen obtained from *s* by replacing every expression by its Equivalent Dot Form, then:

- 1 Replacing the arguments of any Call by (recursively) their transposition to *C*.
- 2 If *s* is part of the declaration of a feature *g* replicated in *C* along a certain repeated inheritance path, replacing any Feature_name used as name of the feature of an unqualified call or as anchor of an anchored type by the name resulting from any renaming of the feature along that path.
- 3 Replacing any Feature_name used as name of the feature of an unqualified call or as anchor of an anchored type, if case 2 does not apply, by the result of any renaming along applicable inheritance paths.
- 4 In every qualified call of target *t*, replacing *t* by (recursively) its transposition *t'* to *C* and the feature of the call by (recursively) its transposition to the type of *t'* in *C*.
- 5 In every Non_object_call of target type *T*, replacing *T* by (recursively) its transposition *T'* to *C* and the feature of the call by (recursively) its transposition to *T'*.
- 6 For every entity *e*, other than an attribute, such that *s* includes a declaration for *e*, replacing every occurrence of *e* by a fresh identifier not used in *C*.
- 7 If an ancestor *B* of *C* has a parent type *P* of base class *A*, replacing every occurrence of any generic parameter *G* of *A* by (recursively) the transposition to *C* of the application to *G* of *P*'s generic substitution.

The transposition to a type *T* of a specimen *s* appearing in a ancestor of the base class *C* of *T* is the result of applying the generic substitution of *T* to the class transposition of *s* to *C*.

8.10.29 Definition: Transposition

The **direct transposition** to a class *B* of a specimen *s* appearing in a parent class *A* of *B* is the specimen obtained from *s* by replacing every expression by its Equivalent Dot Form, then:

- 1 Replacing the arguments of any Call by (recursively) their direct transposition to *B*.
- 2 If *s* is part of the declaration of a feature *g* replicated in *B* along a certain repeated inheritance path, replacing the name of the feature of any unqualified call by the name of the feature as resulting from any renaming along that path.
- 3 In every unqualified call of feature *f* whose feature name *fn* appears in a Rename_pair of the form *fn as gn* in a Parent part for *A*, such that case 2 does not apply, replacing *fn* by the identifier of *gn*.
- 4 In every qualified call of target *t*, replacing *t* by (recursively) its class transposition *t'* to *B* and the feature of the call by (recursively) its transposition to the type of *t'* in *B*.
- 5 In every Non_object_call of target type *T*, replacing *T* by (recursively) its class transposition *T'* to *B* and the feature of the call by (recursively) its transposition to *T'*.
- 6 For every entity *e*, other than an attribute, such that *s* includes a declaration for *e*, replacing every occurrence of *e* by a fresh identifier not used in *B*.
- 7 Replacing every occurrence of a formal generic parameter of *A* by the generic substitution of *B*'s parent type of base class *A*.

The **class transposition** to a class *C* of a specimen *s* appearing in an ancestor *A* of *C* is:

- 1 If *A* and *C* are the same class: *s*.
- 2 If *A* is a parent of an ancestor *B* of *C*: (recursively) the transposition to *C* of the direct transposition of *s* to *B*.

The **transposition** to a type *T* of a specimen *s* appearing in an ancestor of the base class *C* of *T* is the result of applying the generic substitution of *T* to the class transposition of *s* to *C*.

Informative text

The first part (cases 3 and 7) defines transposition to an heir (direct descendant). Cases 8 and 9 generalize this to any descendant. Recall that a descendant *C* of *A* is either *A* itself (case 8) or, recursively, a descendant of an heir *B* of *A* (case 9).

Case takes care of feature renaming. Because

End

8.10.30 Definition: Unfolded redeclaration

Consider a feature *f* of a class *A*. The **unfolded redeclaration** of *f* in an heir *C* of *A* is a **Feature_declaration** defined as follows:

- 1 If *C* **redeclares** *f*, the declaration of *f* in *C*.
- 2 Otherwise, a **Feature_declaration** for a feature with the same extended name, the same signature as *f* and the same **Assigner_mark** if any, both transposed to *C*, and an **Attribute_or_routine** consisting solely of:
 - If *f* is deferred, a **Feature_body** of the **Deferred** kind.
 - If *f* is an effective routine, a **do** clause whose **Compound** reads just **Precursor** (if *f* is a procedure) or **Result := Precursor** (if *f* is a function), followed by the parenthesized list of formal arguments if any.
 - If *f* is an attribute, an **attribute** clause whose **Compound** reads just **Result := Precursor**.

8.10.31 Validity: Join rule

Validity code: *VDJR*

It is valid for a class *C* to inherit two different features under the same final name under and only under the following conditions:

- 1 If both are inherited as effective, *C* redefines both into a common version.
- 2 If both are inherited as deferred, the unfolded redeclaration in *C* of each of them is a valid redeclaration of the other.
- 3 Otherwise, the unfolded redeclaration in *C* of the one inherited as effective is a valid redeclaration of the one inherited as deferred.

Informative text

THE FOLLOWING INFORMATIVE TEXT NEEDS UPDATING. The Join rule indicates that joined features must have exactly the same signature — argument and result types.

What matters is the signature after possible redefinition or effecting. So in practice you may join precursor features with different signatures: it suffices to redeclare them using a feature which (as required by point 2 of the Redeclaration rule) must have a signature conforming to all of the precursors' signatures.

If the redeclaration describes an effective feature, this is the case of both joining and effecting a set of inherited features. If the redeclaration describes a feature that is still deferred, it is a redefinition, used to adapt the signature and possibly the specification. In this case, point 4 of the Redeclaration rule requires every one of the precursors to appear in the **Redefine** subclause for the corresponding parent.

Condition 1 mentions “redeclaration or effecting”. These two cases are not exclusive: an effecting — turning a feature *f*, inherited as deferred from a parent of *C*, into an effective one — can result from a new declaration of *f* in *C*, but also from a “join” of *f* with an effective feature inherited under the same name from another parent.

In any case, nothing requires the precursors' signatures to conform to each other, as long as the signature of the version in *C* conforms to all of them. This means you may write a class inheriting two deferred features of the form

f (*p*: *P*): *T* ...

f (*t*: *Q*): *U* ...

and redeclare them with

f (*x*: ? *R*): *V* ...

provided *R* conforms to both *P* and *Q* and *V* to both *T* and *U*. No conformance is required between the types appearing in the precursors' signatures (*P* and *Q*, *T* and *U*).

The assumption that the features are "different" is important: they could in fact be the same feature, appearing in two parents of *C* that have inherited it from a common ancestor, without any intervening redeclaration. This would be a valid case of repeated inheritance; here the rule that determines validity is the Repeated Inheritance Consistency constraint. The semantic specification (sharing under the Repeated Inheritance rule) indicates that *C* will have just one version of the feature.

Conditions 1 and 2 of the Join rule are consistency requirements on aliases and on assigner commands. The condition on aliases is consistent with condition Z of the Redeclaration rule, which requires a redeclaration to keep the alias if any; it was noted in the comment to that rule that redeclaration is not the appropriate way to add, change or remove an alias (you should use renaming for that purpose); neither is join. The condition on assigner commands ensures that any Assigner_call has the expected effect, even under dynamic binding on a target declared of a parent type.

End

8.10.32 Semantics: Join Semantics rule

Joining in a class *C* two or more inherited features with the same final name under the terms of the Join rule yields a single feature of *C* defined as follows:

- 1 If at least one of these features is effective: its unfolded redeclaration in *C*.
- 2 Otherwise: the unfolded redeclaration in *C* of any of them.

Informative text

***** TO BE REDONE *****The rule covers three cases:

- An explicit redeclaration, which serves as a redeclaration of all the joined precursors, and gives them a new signature (which must conform to all their signatures per the Join rule), body (since it serves as "unfolded redeclaration" in point *****) and assertions (point *****).
- No redeclaration, with precursors all deferred, all having the same signature; they are then merged into a single deferred feature.
- No redeclaration, with one effective feature and the others deferred, all with the same signature; the effective feature then serves as effecting of the others.

In the absence of a redeclaration, point***** states that the new feature has no specific precondition and postcondition. It will still, however, have a **combined precondition** and a **combined postcondition** obtained from the precursors' assertions. In the case of a redeclaration, the combined precondition and postcondition also include the assertions, if any, of the redeclared version.

Point ***** leaves the concatenation order unspecified.

In point ***** , there can be at most one effective precursor because of the Join rule.

In point ***** (corresponding to a rare case) language processing tools should produce an obsolescence message for the class performing the join, but the resulting feature is not itself obsolete.

End

8.11 Types

Informative text

Types describe the form and properties of objects that can be created during the execution of a system. The type system lies at the heart of the object-oriented approach; the use of types to declare all entities leads to more clear software texts and permits compilers to detect many potential errors and inconsistencies before they can cause damage.

End

8.11.1 Syntax: Types

Type \triangleq Class_or_tuple_type | Formal_generic_name | Anchored

Class_or_tuple_type \triangleq Class_type | Tuple_type

Class_type \triangleq [Attachment_mark] Class_name [Actual_generics]

Attachment_mark \triangleq "?" | "!"

Anchored \triangleq [Attachment_mark] like Anchor

Anchor \triangleq Feature_name | Current

Informative text

The most common and versatile kind is **Class_type**, covering types described by a class name, followed by actual generic parameters if the class is generic. The class name gives the type's base class. If the base class is expanded, the **Class_type** itself is an expanded type; if the base class is non-expanded, the **Class_type** is a reference type.

An **Attachment_mark ?** indicates that the type is **detachable**: its values may be void — not attached to an object. The **!** mark indicates the reverse: the type is **attached**, meaning that its values will always denote an object; language rules, in particular constraints on attachment, guarantee this. No **Attachment_mark** means the same as **!**, to ensure that a type, by default, will be attached.

End

8.11.2 Semantics: Direct instances and values of a type

The **direct instances** of a type *T* are the run-time objects resulting from: representing a **manifest constant**, **manifest tuple**, **Manifest_type**, **agent** or **Address** expression of type *T*; applying a **creation operation** to a **target** of type *T*; (recursively) cloning an existing direct instance of *T*.

The **values** of a type *T* are the possible run-time values of an entity or expression of type *T*.

8.11.3 Semantics: Instance of a type

The **instances** of a type *TX* are the **direct instances** of any type **conforming** to *TX*.

Informative text

Since every type conforms to itself, this is equivalent to stating that the instances of *TX* are the direct instances of *TX* and, recursively, the instances of any other type conforming to *TX*.

End

8.11.4 Semantics: Instance principle

Any value of a type *T* is:

- If *T* is **reference**, either a reference to an **instance** of *T* or (unless *T* is **attached**) a void reference.
- If *T* is **expanded**, an instance of *T*.

8.11.5 Definition: Instance, direct instance of a class

An instance of a class *C* is an instance of any type *T* based on *C*.

A direct instance of *C* is a direct instance of any type *T* based on *C*.

Informative text

For non-generic classes the difference between *C* and *T* is irrelevant, but for a generic class you must remember that by itself the class does not fully determine the shape of its direct instances: you need a type, which requires providing a set of actual generic parameters.

End

8.11.6 Base principle

Any type *T* proceeds, directly or indirectly, from a *Class_or_tuple_type* called its **base type**, and an underlying class called its **base class**.

The base class of a type is also the base class of its base type.

Informative text

A *Class_type* is its own base type; an anchored type *like anchor* with *anchor* having base type *U* also has *U* as its base type. For a formal generic parameter *G* in *class C [G → T] ...* the base type is (in simple cases) the constraining type *T*, or *ANY* if the constraint is implicit.

The base class is the class providing the features applicable to instances of the type. If *T* is a *Class_type* the connection to a class is direct: *T* is either the name of a non-generic class, such as *PARAGRAPH*, or the name of a generic class followed by *Actual_generics*, such as *LIST [WINDOW]*. In both cases the base class of *T* is the class whose name is used to obtain *T*, with any *Actual_generics* removed: *PARAGRAPH* and *LIST* in the examples. For a *Tuple_type*, the base class is a fictitious class *TUPLE*, providing the features applicable to all tuples.

For types not immediately obtained from a class we obtain the base class by going through base type: for example *T* is an *Anchored* type of the form *like anchor*, and *anchor* is of type *LIST [WINDOW]*, then the base class of that type, *LIST*, is also the base class of *T*.

End

8.11.7 Base rule

The **base type** of any type is a *Class_or_tuple_type*, with no *Attachment_mark*.

The **base class** of any type other than a *Class_or_tuple_type* is (recursively) the base class of its base type.

The **direct instances** of a type are those of its base type.

Informative text

Why are these notions important? Many of a type's key properties (such as the features applicable to the corresponding entities) are defined by its base class. Furthermore, class texts almost never directly refer to classes: they refer to *types* based on these classes.

End

8.11.8 Validity: Class Type rule

Validity code: *VTCT*

A *Class_type* is valid if and only if it satisfies the following two conditions:

- 1 Its *Class_name* is the name of a class in the surrounding *universe*.
- 2 If it has a *"?" Attachment_mark*, that class is not expanded.

Informative text

The class given by condition 1 will be the type's base class. Regarding condition 2, an expanded type is always attached, so an *Attachment_mark* would not make sense in that case.

End

8.11.9 Semantics: Type Semantics rule

To define the semantics of a type *T* it suffices to specify:

- 1 Whether T is expanded or reference.
- 2 Whether T , if reference, is attached or detachable.
- 3 What is T 's base type.
- 4 If T is a Class_or_tuple_type, what are its base class and its type parameters if any.

8.11.10 Definition: Base class and base type of an expression

Any expression e has a **base type** and a **base class**, defined as the base type and base class of the type of e .

8.11.11 Semantics: Non-generic class type semantics

A non-generic class C used as a type (of the Class_type category) has the same expansion status as C (i.e. it is expanded if C is an expanded class, reference otherwise). It is its own base type (after removal of any Attachment_mark) and base class.

8.11.12 Definition: Expanded type, reference type

A type T is **expanded** if and only if it is not a Formal_generic_name and the base class of its deanchored form is an expanded class.

T is a **reference type** if it is neither a Formal_generic_name nor expanded.

Informative text

This definition characterizes every type as either reference or expanded, except for the case of a Formal_generic_name, which stands for any type to be used as actual generic parameter in a generic derivation: some derivations might use a reference type, others an expanded type.

Tuple types are, as a consequence of the definition, reference types.

End

8.11.13 Definition: Basic type

The basic types are BOOLEAN, CHARACTER and its sized variants, INTEGER and its sized variants, REAL and its sized variants and POINTER.

Informative text

Like most other types, the basic types are defined by classes, found in the Kernel Library. In other words they are not predefined, “magic” types, but fit in the normal class-based type system of Eiffel.

Compilers typically know about them, so that they can generate code that performs arithmetic and relational operations as fast as in lower-level languages where basic types are built-in. This is only for efficient implementation: semantically, the basic types are just like other class types.

End

8.11.14 Definition: Anchor, anchored type, anchored entity

The **anchor** of an anchored type like anchor is the entity anchor. A declaration of an entity with such a type is an **anchored declaration**, and the entity itself is an **anchored entity**.

Informative text

The anchor must be either an entity, or Current. If an entity, anchor must be the final name of a feature of the enclosing class.

End

Informative text

The syntax permits *x* to be declared of type **like anchor** if *anchor* is itself anchored, of type **like other_anchor**. Although most developments do not need such anchor chains, they turn out to be occasionally useful for advanced applications. But then of course we must make sure that an anchor chain is meaningful, by excluding cycles such as *a* declared as **like b**, *b* as **like c**, and *c* as **like a**. The following definition helps.

End

8.11.15 Definition: Anchor set; cyclic anchor

The **anchor set** of a type *T* is the set of entities containing, for every anchored type **like anchor** involved in *T*:

- *anchor*.
- (Recursively) the anchor set of the type of *anchor*.

An entity *a* of type *T* is a **cyclic anchor** if the anchor set of *T* includes *a* itself.

Informative text

The anchor set of *LIST [like a, HASH_TABLE [like b, STRING]]* is, according to this definition, the set *{a, b}*.

Because of genericity, the cycles that make an anchor “cyclic” might occur not directly through the anchors but through the types they involve, as with *a* of type *LIST [like b]* where *b* is of type **like a**. Here we say that a type “involves” all the types appearing in its definition, as captured by the following definition.

End

8.11.16 Definition: Types and classes involved in a type

The types **involved** in a type *T* are the following:

- *T* itself.
- If *T* is of the form *a T'* where *a* is an **Attachment_mark**: (recursively) the types involved in *T'*.
- If *T* is a **generically derived Class_type** or a **Tuple_type**: all the types (recursively) involved in any of its actual parameters.

The *classes* involved in *T* are the base classes of the types involved in *T*.

Informative text

A [B, C, LIST [ARRAY [D]]] involves itself as well as *B, C, D, ARRAY [D]* and *LIST [ARRAY [D]]*. The notion of *cyclic anchor* captures this notion in full generality; the basic rule, stated next, will be that if *a* is a cyclic anchor you may not use it as anchor: the type **like a** will be invalid.

End

8.11.17 Definition: Deanchored form of a type

The **deanchored form** of a type *T* in a class *C* is the type (**Class_or_tuple_type** or **Formal_generic**) defined as follows:

- 1 If *T* is **like Current**: the current type of *C*.
- 2 If *T* is **like anchor** where the type *AT* of *anchor* is not anchored: (recursively) the deanchored form of *AT*.
- 3 If *T* is **like anchor** where the type *AT* of *anchor* is anchored but *anchor* is not a cyclic anchor: (recursively) the deanchored form of *AT* in *C*.
- 4 If *T* is *a AT*, where *a* is an **Attachment_mark**: *a DT*, where *DT* is (recursively) the deanchored form of *AT* deprived of its **Attachment_mark** if any.

- 5 If none of the previous cases applies: *T* after replacement of any actual parameter by (recursively) its deanchored form.

Informative text

Although useful mostly for anchored types, the notion of “deanchored form” is, thanks to the phrasing of the definition, applicable to *any* type. Informally, the deanchored form yields, for an anchored type, what the type “really means”, in terms of its anchor’s type. It reflects the role of anchoring as what programmers might call a macro mechanism, a notational convenience to define types in terms of others.

Case 4 enables us to treat ? *like anchor* as a detachable type whether the type of *anchor* is attached or detachable.

End

8.11.18 Validity: Anchored Type rule

Validity code: VTAT

It is valid to use an anchored type *AT* of the form *like anchor* in a class *C* if and only if it satisfies the following conditions:

- 1 *anchor* is either **Current** or the final name of a query of *C*.
- 2 *anchor* is not a cyclic anchor.
- 3 The deanchored form *UT* of *AT* is valid in *C*.

The base class and base type of *AT* are those of *UT*.

Informative text

An anchored type has no properties of its own; it stands as an abbreviation for its unfolded form. You will not, for example, find special conformance rules for anchored type, but should simply apply the usual conformance rules to its deanchored form.

Anchored declaration is essentially a syntactical device: you may always replace it by explicit redefinition. But it is extremely useful in practice, avoiding much code duplication when you must deal with a set of entities (attributes, function results, routine arguments) which should all follow suit whenever a proper descendant redefines the type of one of them, to take advantage of the descendant’s more specific context.

End

8.11.19 Definition: Attached, detachable

A type is **detachable** if its deanchored form is a **Class_type** declared with the ? **Attachment_mark**.

A type is **attached** if it is not detachable.

Informative text

By taking the “deanchored form”, we can apply the concepts of “attached” and “detachable” to an anchored type *like a*, by just looking at the type of *a* and finding out whether it is attached or not.

As a consequence of this definition, an expanded type is attached.

As the following semantic definition indicates, the idea of declaring a type as attached is to guarantee that its values will never be void.

End

8.11.20 Semantics: Attached type semantics

Every run-time value of an attached type is non-void (attached to an object).

Informative text

In contrast, values of a detachable type may be void.

These definitions rely on the run-time notion of a *value* being attached (to an object) or void. So there is a distinction between the *static* property that an entity is attached (meaning that language rules guarantee that its run-time values will never be void) or detachable, and the *dynamic* property that, at some point during execution, its value will be attached or not. If there's any risk of confusion we may say "statically attached" for the entity, and "dynamically attached" for the run-time property of its value.

The validity and semantic rules, in particular on attachment operations, ensure that attached types indeed deserve this qualification, by initializing all the corresponding entities to attached values, and protecting them in the rest of their lives from attachment to void.

From the above semantics, the **!** mark appears useless since an absent **Attachment_mark** has the same effect. The mark exists to ensure a smooth transition: since earlier versions of Eiffel did not guarantee void-safety, types were detachable by default. To facilitate adaptation to current Eiffel and avoid breaking existing code, compilers may offer a compatibility option (departing from the Standard, of course) that treats the absence of an **Attachment_mark** as equivalent to **?**. You can then use **!** to mark the types that you have moved to the attached world and adapt your software at your own pace, class by class if you wish, to the new, void-safe convention.

End

8.11.21 Definition: Stand-alone type

A **Type** is **stand-alone** if and only if it **involves** neither any **Anchored** type nor any **Formal_generic_name**.

Informative text

In general, the semantics of a type may be relative to the text of class in which the type appears: if the type involves generic parameters or anchors, we can only understand it with respect to some class context. A stand-alone type always makes sense — and always makes the same sense — regardless of the context.

We restrict ourselves to stand-alone types when we want a solidly defined type that we can use anywhere. This is the case in the validity rules enabling creation of a root object for a system, and the definition of a once function.

End

8.12 Genericity

Informative text

The types discussed so far were directly defined by classes. The *genericity* mechanism, still based on classes, gives us a new level of flexibility through **type parameterization**. You may for example define a class as **LIST [G]**, yielding not just one type but many: **LIST [INTEGER]**, **LIST [AIRPLANE]** and so on, parameterized by **G**.

Parameterized classes such as **LIST** are known as **generic classes**; the resulting types, such as **LIST [INTEGER]**, are **generically derived**. "Genericity" is the mechanism making generic classes and generic derivations possible.

Two forms of genericity are available: with *unconstrained* genericity, **G** represents an arbitrary type; with *constrained* genericity, you can demand certain properties of the types represented by **G**, enabling you to do more with **G** in the class text.

End

8.12.1 Syntax: Actual generic parameters

Actual_generics \triangleq "[**Type_list** "]"

Type_list \triangleq {**Type** ", ..."}⁺

8.12.2 Syntax: Formal generic parameters

Formal_generics \triangleq "[**Formal_generic_list** "]"

Formal_generic_list \triangleq {Formal_generic ";..."}⁺

Formal_generic \triangleq [frozen] Formal_generic_name [Constraint]

Formal_generic_name \triangleq [?] Identifier

8.12.3 Validity: Formal Generic rule

Validity code: *VCFG*

A *Formal_generics* part of a *Class_declaration* is valid if and only if every *Formal_generic_name G* in its *Formal_generic_list* satisfies the following conditions:

- 1 *G* is different from the name of any class in the universe.
- 2 *G* is different from any other *Formal_generic_name* appearing in the same *Formal_generics* part.

Informative text

Adding the **frozen** qualification to a formal generic, as in *D[frozen G]* rather than just *C[G]*, means that conformance on the corresponding generically derived classes requires identical actual parameters: whereas *C[U]* conforms to *C[T]* if *U* conforms to *T*, *D[U]* does not conform to *D[T]* if *U* is not *T*.

Adding the **?** mark to a *Formal_generic_name*, as in *? G*, means that the class may declare *self-initializing* variables (variables that will be initialized automatically on first use) of type *G*; this requires that any actual generic parameter that is an attached type must also be self-initializing, that is to say, make *default_create* from *ANY* available for creation.

End

8.12.4 Definition: Generic class; constrained, unconstrained

Any class declared with a *Formal_generics* part (constrained or not) is a **generic class**.

If a formal generic parameter of a generic class is declared with a **Constraint**, the parameter is **constrained**; if not, it is **unconstrained**.

A generic class is itself **constrained** if it has at least one constrained parameter, **unconstrained** otherwise.

Informative text

A generic class does not describe a type but a template for a set of possible types. To obtain an actual type, you must provide an *Actual_generics* list, whose elements are themselves types. This has a name too, per the following definition.

End

8.12.5 Definition: Generic derivation, non-generic type

The process of producing a type from a generic class by providing actual generic parameters is **generic derivation**.

A type resulting from a generic derivation is a **generically derived type**, or just **generic type**.

A type that is not generically derived is a **non-generic type**.

Informative text

It is preferable to stay away from the term “generic instantiation” (sometimes used in place of “generic derivation”) as it creates a risk of confusion with the normal meaning of “instantiation” in object-oriented development: the *run-time process of obtaining an object from a class*.

End

8.12.6 Definition: Self-initializing formal parameter

A *Formal_generic_parameter* is **self-initializing** if and only if its declaration includes the optional **?** mark.

Informative text

This is related to the notion of self-initializing *type*: a type which makes *default_create* from *ANY* available for creation. The rule will be that an actual generic parameter corresponding to a self-initializing formal parameter must itself, if attached, be a self-initializing type.

End

8.12.7 Definition: Constraint, constraining types of a *Formal_generic*

The **constraint** of a formal generic parameter is its *Constraint* part if present, and otherwise *ANY*.

Its **constraining types** are all the types listed in its *Constraining_types* if present, and otherwise just *ANY*.

8.12.8 Syntax: Generic constraints

Constraint \triangleq " \rightarrow " *Constraining_types* [*Constraint_creators*]

Constraining_types \triangleq *Single_constraint* | *Multiple_constraint*

Single_constraint \triangleq *Type* [*Renaming*]

Renaming \triangleq *Rename* **end**

Multiple_constraint \triangleq "{" *Constraint_list* ""

Constraint_list \triangleq {*Single_constraint* "," ...}⁺

Constraint_creators \triangleq **create** *Feature_list* **end**

8.12.9 Validity: Generic Constraint rule

Validity code: *VTGC*

A *Constraint* part appearing in the *Formal_generics* part of a class *C* is valid if and only if it satisfies the following conditions for every *Single_constraint* listing a type *T* in its *Constraining_types*:

- 1 *T* involves no anchored type.
- 2 If a *Renaming* clause **rename** *rename_list* **end** is present, a class definition of the form **class** *NEW* **inherit** *BT* **rename** *rename_list* **end** (preceded by **deferred** if the *base class* of *T* is deferred), where *BT* is the base class of *T*, would be valid.

Informative text

There is no requirement here on the *Constraint_creators* part, although in most cases it will list names (after *Renaming*) of creation procedures of the constraining types. The precise requirement is captured by other rules.

Condition 2 implies that the features listed in the *Constraint_creators* are, after possible *Renaming*, names of features of one or more of the constraining types, and that no clash remains that would violated the rules on inheritance. In particular, you can use the *Renaming* either to merge features if they come from the same seeds, or (the other way around) separate them.

If *T* is based on a deferred class the fictitious class *NEW* should be declared as **deferred** too, otherwise it would be invalid if *T* has deferred features. On the other hand, *NEW* cannot be valid if *T* is based on a frozen class; in this case it is indeed desirable to disallow the use of *T* as a constraint, since the purpose of declaring a class **frozen** is to prevent inheritance from it

End

8.12.10 Definition: Constraining creation features

If *G* is a formal generic parameter of a class, the **constraining creators of** *G* are the features of *G*'s *Constraining_types*, if any, corresponding after possible *Renaming* to the feature names listed in the *Constraining_creators* if present.

Informative text

Constraining creators should be creation procedures, but not necessarily (as seen below) in the constraining types themselves; only their instantiatable descendants are subject to this rule.

End

8.12.11 Validity: Generic Derivation rule

Validity code: VTGD

Let *C* be a generic class. A *Class_type CT* having *C* as base class is valid if and only if it satisfies the following conditions for every actual generic parameter *T* and every *Single_constraint U* appearing in the constraint for the corresponding formal generic parameter *G*:

- 1 The number of Type components in *CT*'s *Actual_generics* list is the same as the number of *Formal_generic* parameters in the *Formal_generic_list* of *C*'s declaration.
- 2 *T* conforms to the type obtained by applying to *U* the *generic substitution* of *CT*.
- 3 If *C* is expanded, *CT* is *generic-creation-ready*.
- 4 If *G* is a self-initializing formal parameter and *T* is attached, then *T* is a *self-initializing type*.

Informative text

In the case of unconstrained generic parameters, only condition 1 applies, since the constraint in that case is *ANY*, which trivially satisfies the other two conditions.

Condition 3 follows from the semantic rule permitting “lazy” creation of entities of expanded types on first use, through *default_create*. *Generic-creation-readiness* (defined next) is a condition on the actual generic parameters that makes such initialization safe if it may involve creation of objects whose type is the corresponding formal parameters.

Condition 4 guarantees that if *C* relies, for some of its variables of type *G*, on automatic initialization on first use, *T* provides it, if attached (remember that this includes the case of expanded types), by making *default_create* from *ANY* available for creation. If *T* is detachable this is not needed, since *Void* will be a suitable initialization value.

End

8.12.12 Definition: Generic-creation-ready type

A type of base class *C* is **generic-creation-ready** if and only if every actual generic parameter *T* of its *deanchored form* satisfies the following conditions:

- 1 If the specification of the corresponding formal generic parameter includes a *Constraint_creators*, the *versions* in *T* of the *constraining creators* for the corresponding formal parameter are *creation procedures*, *available for creation* to *C*, and *T* is (recursively) *generic-creation-ready*.
- 2 If *T* is expanded, it is (recursively) *generic-creation-ready*.

Informative text

Although phrased so that it is applicable to any type, the condition is only interesting for generically derived types of the form *C[... , T, ...]*. Non-generically-derived types satisfy it trivially since there is no applicable *T*.

The role of this condition is to make sure that if class *C[... , G , ...]* may cause a creation operation on a target of type *G* — as permitted only if the class appears as *C[... , G → CONST create cp1, ... end, ...]* — then the corresponding actual parameters, such as *T*, will support the given features — the “constraining creators” — as creation procedures.

It might then appear that *generic-creation-readiness* is a validity requirement on *any* actual generic parameter. But this would be more restrictive than we need. For example *T* might be a deferred type; then it cannot have any creation procedures, but that’s still OK because we cannot

create instances of T , only of its effective descendants. Only if it is possible to **create** an actual object of the type do we require generic-creation-readiness. Overall, we need generic-creation-readiness only in specific cases, including:

- For the creation type of a creation operation: conditions 4 of the Creation Instruction rule and 3 of the Creation Expression rule.
- For a **Parent** in an **Inheritance** part: condition 6 of the Parent rule.
- For an expanded type: condition 3 of the just seen Generic Derivation rule.

End

8.12.13 Semantics: Generically derived class type semantics

A generically derived Class_type of the form $C[\dots]$, where C is a generic class, is expanded if C is an expanded class, reference otherwise. It is its own base type, and its base class is C .

Informative text

So $LINKED_LIST[POLYGON]$ is its own base type, and its base class is $LINKED_LIST$.

End

8.12.14 Definition: Base type of a single-constrained formal generic

The base type of a constrained Formal_generic_name G having as its constraining types a Single_constraint listing a type T is:

- 1 If T is a Class_or_tuple_type: T .
- 2 Otherwise (T is a Formal_generic_name): the base type of T if it can be determined by (recursively) case 1, otherwise ANY .

Informative text

The definition is never cyclic since the only recursive part is the use of case 1 from case 2.

Case 1 is the common one: for $C[G \rightarrow T]$ we use as base type of G , in C , the base type of T . We need case 2 to make sure that this definition is not cyclic, because we permit cases such as $C[G, H \rightarrow D[G]]$, and as a consequence cases such as $C[G \rightarrow H, H \rightarrow G]$ or even $C[G \rightarrow G]$ even though they are not useful; both of these examples yield ANY as base types for the parameters.

As a result of the definition of “constraining types”, the base type of an unconstrained formal generic, such as G in $C[G]$, is also ANY .

End

8.12.15 Definition: Base type of an unconstrained formal generic

The base type of an unconstrained Formal_generic_name type is ANY .

8.12.16 Definition: Reference or expanded status of a formal generic

A Formal_generic_name represents a reference type or expanded type depending on the corresponding status of the associated actual generic parameter in a particular generic derivation.

8.12.17 Definition: Current type

Within a class text, the **current type** is the type obtained from the current class by providing as actual generic parameters, if required, the class's own formal generic parameters.

Informative text

Clearly, the base class of the current type is always the current class.

End

8.12.18 Definition: Features of a type

The features of a type are the features of its base class.

Informative text

These are the features applicable to the type's instances (which are also instances of its base class).

End

8.12.19 Definition: Generic substitution

Every type T defines a mapping σ from names to types known as its **generic substitution**:

- 1 If T is generically derived, σ associates to every **Formal_generic_name** the corresponding actual parameter.
- 2 Otherwise, σ is the identity substitution.

8.12.20 Generic Type Adaptation rule

The signature of an entity or feature f of a type T of base class C is the result of applying T 's generic substitution to the signature of f in C .

Informative text

The signature include both the type of an entity or query, and the argument types for a routine; the rule is applicable to both parts.

End

8.12.21 Definition: Generically constrained feature name

Consider a generic class C , a constrained **Formal_generic_name** G of C , a type T appearing as one of the **Constraining_types** for G , and a feature f of name $fname$ in the base class of T . The **generically constrained names** of f for G in C are:

- 1 If one or more **Single_constraint** clauses for T include a **Rename** part with a clause $fname$ as $ename$, where the **Feature_name** part of $ename$ (an **Extended_feature_name**) is $gname$: all such $gname$.
- 2 Otherwise: just $fname$.

8.12.22 Validity: Multiple Constraints rule

Validity code: *VTMC*

A feature of name $fname$ is applicable in a class C to a target x whose type is a **Formal_generic_name** G constrained by two or more types $CONST_1, CONST_2, \dots$, if and only if it satisfies the following conditions:

- 1 At least one of the $CONST_i$ has a feature available to C whose generically constrained name for G in C is $fname$.
- 2 If this is the case for two or more of the $CONST_i$, all the corresponding features are the same.

8.12.23 Definition: Base type of a multi-constraint formal generic type

The base type of a multiply constrained **Formal_generic_name** type is a type generically derived, with the same actual parameters as the current class, from a fictitious class with none of the optional parts except for **Formal_generics** and an **Inheritance** clause that lists all the constraining types as parents, with the given **Renaming** clause if any, and resolves any conflicts between potentially ambiguous features by further renaming them to new names not available to developers.

8.13 Tuples

Informative text

Based on a bare-bones form of class — with no class names — tuple types provide a concise and elegant solution to a number of issues:

- Writing functions with multiple results, ensuring complete symmetry with multiple arguments.

- Describing sequences of values of heterogeneous types, or “tuples”, such as [[some_integer](#), [some_string](#), [some_object](#)], convenient for example as arguments to printing routines.
- Achieving the effect of routines with a variable number of arguments.
- Achieving the effect of generic classes with a variable number of generic parameters.
- Using simple classes, defined by a few attributes and the corresponding assigner commands — similar to the “structures” or “records” of non-O-O languages, but in line with O-O principles — without writing explicit class declarations.
- Making possible the agent mechanism through which you can handle routines as objects and define higher-order routines.

End

8.13.1 Syntax: Tuple types

$\text{Tuple_type} \triangleq \text{TUPLE} [\text{Tuple_parameter_list}]$

$\text{Tuple_parameter_list} \triangleq "[[" \text{ Tuple_parameters } "]"$

$\text{Tuple_parameters} \triangleq \text{Type_list} \mid \text{Entity_declaration_list}$

8.13.2 Syntax: Manifest tuples

$\text{Manifest_tuple} \triangleq "[[" \text{ Expression_list } "]"$

$\text{Expression_list} \triangleq \{ \text{Expression} \, ", \dots" \}^*$

8.13.3 Definition: Type sequence of a tuple type

The **type sequence** of a tuple type is the sequence of types obtained by listing its parameters, if any, in the order in which they appear, every labeled parameter being listed as many times as it has labels.

Informative text

The type sequence for `TUPLE` is empty; the type sequence for `TUPLE [INTEGER; REAL; POLYGON]` is `INTEGER, REAL, POLYGON`; the type sequence for `TUPLE [i, j: INTEGER; r: REAL; p: POLYGON]` is `INTEGER, INTEGER, REAL, POLYGON`, where `INTEGER` appears twice because of the two labels `i, j`.

End

8.13.4 Definition: Value sequences associated with a tuple type

The **value sequences** associated with a tuple type `T` are sequences of values, each of the type appearing at the corresponding position in `T`'s **type sequence**.

Informative text

Parameter labels play no role in the semantics of tuples and their conformance properties. They never intervene in tuple expressions (such as `[25, -8.75, pol]`). Their only use is to allow name-based access to tuple fields, as `your_tuple.label`, guaranteeing statically the type of the result.

End

8.14 Conformance

Informative text

Conformance is the most important characteristic of the Eiffel type system: it determines when a type may be used in lieu of another.

The most obvious use of conformance is to make assignment and argument passing type-safe: for `x` of type `T` and `y` of type `V`, the instruction `x := y`, and the call `some_routine(y)` with `x` as formal argument, will only be valid if `V` is *compatible* with `T`, meaning that it either *conforms* or *converts* to `T`. Conformance also governs the validity of many other constructs, as discussed below.

Conformance, as the rest of the type system, relies on inheritance. The basic condition for V to conform to T is straightforward:

- The base class of V must be a descendant of the base class of T .
- If V is a generically derived type, its actual generic parameters must conform to the corresponding ones in T : $B[Y]$ conforms to $A[X]$ only if B conforms to A and Y to X .
- If T is expanded, inheritance is not involved: V can only be T itself.

A full understanding of conformance requires the formal rules explained below, which take into account the details of the type system: constrained and unconstrained genericity, special rules for predefined arithmetic types, tuple types, anchored types.

The following discussion introduces the various conformance rules of the language as “definitions”. Although not validity constraints themselves, these rules play a central role in many of the constraints, so that language processing tools such as compilers may need to refer to them in their error messages. For that reason each rule has a validity code of the form VNCx.

End

8.14.1 Definition: Compatibility between types

A type is **compatible** with another if it either conforms or converts to it.

8.14.2 Definition: Compatibility between expressions

An expression b is **compatible with** an expression a if and only if b either conforms or converts to a .

8.14.3 Definition: Expression conformance

An expression exp of type $SOURCE$ **conforms to** an expression ent of type $TARGET$ if and only if they satisfy the following conditions:

- 1 $SOURCE$ conforms to $TARGET$.
- 2 If $TARGET$ is attached, so is $SOURCE$.
- 3 If $SOURCE$ is expanded, its version of the function *cloned* from ANY is available to the current class.

Informative text

So conformance of expressions is more than conformance of their types. Both conditions 2 and 3 are essential. Condition 2 guarantees that execution will never attach a void value to an entity declared of an attached type — a declaration intended precisely to rule out that possibility, so that the entity can be used as target of calls. Condition 3 allows us, in the semantics of attachment, to use a cloning operation when attaching an object with “copy semantics”, without causing inconsistencies.

A later definition will state what it means for an expression b to *convert* to another a . As a special case these properties also apply to entities.

Conformance and convertibility are exclusive of each other, so we study the two mechanisms separately. The rest of the present discussion is devoted to conformance.

End

8.14.4 Validity: Signature conformance

Validity code: **VNCS**

A signature $t = [B_1, \dots, B_n], [S]$ **conforms to** a signature $s = [A_1, \dots, A_n], [R]$ if and only if it satisfies the following conditions:

- 1 Each of the two components of t has the same number of elements as the corresponding component of s .
- 2 Each type in each of the two components of t conforms to the corresponding type in the corresponding component of s .
- 3 Any B_j not identical to the corresponding A_j is detachable.

Informative text

For a signature to conform: the argument types must conform (for a routine); the two signatures must both have a result type or both not have it (meaning they are both queries, or both procedures); and if there are result types, they must conform.

Condition 3 adds a particular rule for “covariant redefinition” of arguments as defined next.

End

8.14.5 Definition: Covariant argument

In a redeclaration of a routine, a formal argument is **covariant** if its type differs from the type of the corresponding argument in at least one of the parents’ versions.

Informative text

From the preceding signature conformance rule, the type of a covariant argument will have to be declared as *detachable*: you cannot redefine $f(x: T)$ into $f(x: U)$ even if U conforms to T ; you may, however, redefine it to $f(x: ?U)$. This forces the body of the redefined version, when applying to x any feature of f , to ensure that the value is indeed attached to an instance of U by applying an Object_test, for example in the form

```
if {x: U} y then y.feature_of_U else ... end
```

This protects the program from *catcalls* — wrongful uses, of a redefined feature, through polymorphism and dynamic binding, to an actual argument of the original, pre-covariant type.

The rule only applies to *arguments*, not results, which do not pose a risk of catcall.

This rule is the reason why the Feature Declaration rule requires that if any routine argument is of an anchored type, that type must be detachable, since anchored declaration is a shorthand for explicit covariance.

End

8.14.6 Validity: General conformance

Validity code: *VNCC*

Let T and V be two types. V **conforms to** T if and only if one of the following conditions holds:

- 1 V and T are identical.
- 2 V conforms directly to T .
- 3 V is *NONE* and T is a detachable reference type.
- 4 V is $B[Y_1, \dots, Y_n]$ where B is a generic class, T is $B[X_1, \dots, X_n]$, and for every X_i the corresponding Y_i is identical to X_i or, if the corresponding formal parameter does not specify **frozen**, conforms (recursively) to X_i .
- 5 For some type U (recursively), V conforms to U and U conforms to T .
- 6 T or V or both are anchored types appearing in the same class C , and the deanchored form of V in C (recursively) conforms to the deanchored form of T .

Informative text

Cases 1 and 2 are immediate: a type conforms to itself, and direct conformance is a case of conformance.

Case 3 introduces the class *NONE* describing void values for references. You may assign such a value to a variable of a reference type not declared as attached (as the role of such declarations is precisely to exclude void values); an expanded target is also excluded since it requires an object.

Case 4 covers the replacement of one or more generic parameters by conforming ones, keeping the same base class: $B[Y]$ conforms to $B[X]$ if Y conforms to X . (This does not yet address conformance to $B[Y_1, \dots, Y_n]$ of a type *CT* based on a class C different from B .) Also note that the **frozen** specification is precisely intended to preclude conformance other than from the given type to itself.

Case 5 is indirect conformance through an intermediate type U .

Finally, case 6 allows us to treat any anchored type, for conformance as for its other properties, as an abbreviation — a “macro” in programmer terminology — for the type of its anchor.

Thanks to this definition of conformance in terms of direct conformance, the remainder of the discussion of conformance only needs to define **direct** conformance rules for the various categories of type.

End

8.14.7 Definition: Conformance path

A **conformance path** from a type U to a type T is a sequence of types T_0, T_1, \dots, T_n ($n \geq 1$) such that T_0 is U , T_n is T , and every T_i (for $0 \leq i < n$) conforms to T_{i+1} . This notion also applies to **classes** by considering the associated base classes.

8.14.8 Validity: Direct conformance: reference types

Validity code: *VNCN*

A Class_type CT of base class C **conforms directly** to a reference type BT if and only if it satisfies the following conditions:

- 1 Applying CT 's generic substitution to one of the conforming parents of C yields BT .
- 2 If BT is attached, so is CT .

Informative text

The restriction to a reference type in this rule applies only to the target of the conformance, BT . The source, CT , may be expanded.

The basic condition, 1, is inheritance. To handle genericity it applies the “generic substitution” associated with every type: for example with a class $C[G, H]$ inheriting from $D[G]$, the type $C[T, U]$ has a generic substitution associating T to G and U to H . So it conforms to the result of applying that substitution to the Parent $D[G]$: the type $D[T]$.

Condition 2 guarantees that we'll never attach a value of a detachable type — possibly void — to a target declared of an attached type; the purpose of such a declaration is to avoid this very case. The other way around, an attached type may conform to a detachable one.

This rule is the foundation of the conformance mechanism, relying on the inheritance structure as the condition governing attachments and redeclarations. The other rules cover refinements (involving in particular genericity), iterations of the basic rule (as with “general conformance”) and adaptations to special cases (such as expanded types).

End

8.14.9 Validity: Direct conformance: formal generic

Validity code: *VNCF*

Let G be a formal generic parameter of a class C , which in the text of C may be used as a Formal_generic_name type. Then:

- 1 No type **conforms directly** to G .
- 2 G **conforms directly** to every type listed in its constraint, and to no other type.

8.14.10 Validity: Direct conformance: expanded types

Validity code: *VNCE*

No type **conforms directly** to an expanded type.

Informative text

From the definition of general conformance, an expanded type ET still conforms, of course, to itself. ET may also conform to reference types as allowed by the corresponding rule (VNCN); the corresponding assignments will use copy semantics. But no other type (except, per General Conformance, for e of type ET , the type **like** e , an abbreviation for ET) conforms to ET .

This rule might seem to preclude mixed-type operations of the kind widely accepted for basic types, such as $f(3)$ where the routine f has a formal argument of type *REAL*, or `your_integer_64 := your_integer_16` with a target of type *INTEGER_64* and a source of type

INTEGER_16. Such attachments, however, involve **conversion** from one type to another. What makes them valid is not conformance but **convertibility**, which does support a broad range of safe mixed-type assignments.

End

8.14.11 Validity: Direct conformance: tuple types

Validity code: **VNCT**

A *Tuple_type U*, of type sequence *us*, **conforms directly** to a type *T* if and only if *T* satisfies the following conditions:

- 1 *T* is a tuple type, of type sequence *ts*.
- 2 The length of *us* is greater than or equal to the length of *ts*.
- 3 For every element *X* of *ts*, the corresponding element of *us* conforms to *X*.

No type conforms directly to a tuple type except as implied by these conditions.

Informative text

Labels, if present, play no part in the conformance.

End

8.15 Convertibility

Informative text

Complementing the conformance mechanism of the previous discussion, convertibility lets you perform assignment and argument passing in cases where conformance does not hold but you still want the operation to succeed after adapting the source value to the target type.

End

8.15.1 Definition: Conversion procedure, conversion type

A procedure whose name appears in a *Converters* clause is a **conversion procedure**.

A type listed in a *Converters* clause is a **conversion type**.

8.15.2 Definition: Conversion query, conversion feature

A query whose name appears in a *Converters* clause is a **conversion query**.

A feature that is either a conversion procedure or a conversion query is a **conversion feature**.

8.15.3 Validity: Conversion principle

No type may both *conform* and *convert* to another.

8.15.4 Validity: Conversion Asymmetry principle

No type *T* may *convert* to another through both a *conversion procedure* and a *conversion query*.

8.15.5 Validity: Conversion Non-Transitivity principle

That *V converts to U* and *U to T* does not imply that *V converts to T*.

8.15.6 Syntax: Converter clauses

Converters \triangleq **convert** *Converter_list*

Converter_list \triangleq {*Converter* ","...}⁺

Converter \triangleq *Conversion_procedure* | *Conversion_query*

Conversion_procedure \triangleq *Feature_name* "(" "{" *Type_list* "}" "

Conversion_query \triangleq *Feature_name* ":" "{" *Type_list* "}" "

8.15.7 Validity: Conversion Procedure rule

Validity code: **VYCP**

A *Conversion_procedure* listing a *Feature_name fn* and appearing in a class *C* with current type *CT* is valid if and only if it satisfies the following conditions, applicable to every type *SOURCE* listed in its *Type_list*:

- 1 *fn* is the name of a creation procedure *cp* of *C*.
- 2 If *C* is not generic, *SOURCE* does not conform to *CT*.
- 3 If *C* is generic, *SOURCE* does not conform to the type obtained from *CT* by replacing every formal generic parameter by its constraint.
- 4 *SOURCE*'s base class is different from the base class of any other conversion type listed for a Conversion_procedure in the Converters clause of *C*.
- 5 The specification of the base class of *SOURCE* does not list a conversion query specifying a type of base class *C*.
- 6 *cp* has exactly one formal argument, of a type *ARG*.
- 7 *SOURCE* conforms to *ARG*.
- 8 *SOURCE* involves no anchored type.

Informative text

Conditions 2 and 3 (the second one covering generic classes) express the crucial requirement, ensuring the Conversion principle: no type that conforms to the current type may convert to it.

In many practical uses of conversion the target class *CX* is expanded; this is the case with *REAL_64*, and with *REAL_32*, to which *INTEGER* also converts. Such cases satisfy condition 2 almost automatically since essentially no other type conforms to an expanded type. But the validity of a conversion specification does not require the enclosing class to be expanded; all that condition 2 states is that the conversion types must not conform to it (more precisely, to the current type).

End

8.15.8 Validity: Conversion Query rule

Validity code: *VYQC*

A Conversion_query listing a Feature_name *fn* and appearing in a class *C* with current type *CT* is valid if and only if it satisfies the following conditions, applicable to every type *TARGET* listed in its Type_list:

- 1 *fn* is the name of a query *f* of *C*.
- 2 If *C* is not generic, *CT* does not conform to *TARGET*.
- 3 If *C* is generic, the type obtained from *CT* by replacing every formal generic parameter by its constraint does not conform to *TARGET*.
- 4 *TARGET*'s base class is different from the base class of any other conversion type listed for a Conversion_query in the Converters clause of *C*.
- 5 The specification of the base class of *TARGET* does not list a conversion procedure specifying a type of base class *C*.
- 6 *f* has no formal argument.
- 7 The result type of *f* conforms to *TARGET*.
- 8 *TARGET* involves no anchored type.

Informative text

Condition 5 is redundant with condition 5 of the Conversion Procedure rule but is included anyway for symmetry. In case of violation, a compiler may refer to either rule.

End

8.15.9 Definition: Converting to a class

A type *T* of base class *CT* **converts to** a class *C* if either:

- The deanchored form of *T* appears as conversion type for a procedure in the Converters clause of *C*.
- A type based on *C* appears as conversion type for a query in the Converters clause of *CT*.

8.15.10 Definition: Converting to and from a type

A type U of base class D **converts to** a **Class_type** T of base class C if and only if either:

- 1 The deanchored form of U is the result of applying the generic substitution of the deanchored form of T to a conversion type for a procedure cp appearing in the **Converters** clause of C .
- 2 The deanchored form of T is the result of applying the generic substitution of the deanchored form of U to a conversion type for a query cq appearing in the **Converters** clause of D .

A **Class_type** T **converts from** a type U if and only if U converts to T .

8.15.11 Definition: Converting “through”

A type U that converts to a type T :

- 1 Converts to T **through a procedure** cp if case 1 of the definition of “converting to a type” applies.
- 2 Converts to T **through a query** cq if case 2 of the definition applies.

These terms also apply to “**converting from**” specifications.

Informative text

From the definitions and validity rules, it's clear that if U converts to T then it's either — but not both — “through a procedure” or “through a query”, and that exactly one routine, cp or f , meets the criteria in each case.

End

8.15.12 Semantics: Conversion semantics

Given an expression e of type U and a variable x of type T , where U converts to T , the effect of a **conversion attachment** of source e and target x is the same as the effect of either:

- 1 If U converts to T through a procedure cp : the creation instruction **create** $x.cp(e)$.
- 2 If U converts to T through a query cq : the assignment $x := e.cq$.

Informative text

This is an “unfolded form” specification expressing the semantics of an operation (conversion attachment) in terms of another: either a creation or a query call. Both of these operations involve an attachment (argument passing or assignment) and so may trigger one other conversion.

End

8.15.13 Definition: Explicit conversion

The Kernel Library class **TYPE** [G] provides a function

adapted alias “[]” ($x: G$): G

which can be used for any type T and any expression exp of a type U compatible with T to produce a T version of exp , written

$\{T\} [exp]$

If U converts to T , this expression denotes the result of converting exp to T , and is called an **explicit conversion**.

Informative text

Explicit conversion involves no new language mechanism, simply a feature of a Kernel Library class and the notion of bracket alias.

For example, assuming a tuple type that converts to **DATE**, you may use

$\{DATE\} [[20, "April", 2005]]$

The `Basic_expression` [20, "April", 2005] is a `Manifest_tuple`. Giving it — through the outermost brackets — as argument to the `adapted` function of `{DATE}` turns it into an expression of type `DATE`. This is permitted for example if class `DATE` specifies a conversion procedure from `TUPLE [INTEGER, STRING, INTEGER]`.

End

8.15.14 Validity: Expression convertibility

Validity code: *VYEC*

An expression `exp` of type `U` **converts to** an entity `ent` of type `T` if and only if `U` **converts to** `T` through a conversion feature `conv` satisfying either of the following two conditions:

- 1 `conv` is precondition-free.
- 2 `exp` statically satisfies the precondition.

8.15.15 Definition: Statically satisfied precondition

A feature precondition is **statically satisfied** if it satisfies any of the following conditions:

- 1 It applies to a boolean, character, integer or real expression involving only constants, states that the expression equals a specific constant value or (in the last three cases) belongs to a specified interval, and holds for that value or interval.
- 2 It applies to the type of an expression, states that it must be one of a specified set of types, and holds for that type.

Informative text

The “constants” of the expression can be manifest constants, or they can be constant actual arguments to a routine — possibly the unfolded form of an assignment, as in `of_type_NATURAL_8 := 1`, whose semantics is that of `create_of_type_natural_from_INTEGER (1)`. Without the notion of “statically satisfied precondition” such instructions would be invalid because `from_INTEGER` in class `NATURAL_8` has a precondition (not every integer is representable as a `NATURAL_8`), and arbitrary preconditions are not permitted for conversion features. This would condemn us to the tedium of writing `{NATURAL_8} 1` and the like for every such case, and would be regrettable since `1 is` as a matter of fact acceptable as a `NATURAL_8`. So the definition of expression convertibility permits a “statically satisfied” precondition, making such cases valid.

It would be possible to generalize the definition by making permissible any precondition that can be assessed statically. But this would leave too much initiative to individual compilers: a “smarter” compiler might accept a precondition that another rejects, leading to incompatibilities. It was judged preferable to limit the rule to the two cases known to be important in practice; if others appear in the future, the rule will be extended.

End

8.15.16 Validity: Precondition-free routine

Validity code: *VYPF*

A feature `r` of a class `C` is **precondition-free** if it is either:

- 1 Immediate in `C`, with either no `Precondition` clause or one consisting of a single `Assertion_clause` (introduced by `require`) whose `Boolean_expression` is the constant `True`.
- 2 Inherited, and such that every precursor of `r` is (recursively) precondition-free, or `r` is redeclared in `C` with a `Precondition` consisting of a single `Assertion_clause` (introduced by `require else`) whose `Boolean_expression` is the constant `True`.

Informative text

A feature is “immediate” if it is declared in the class itself. In the other case, “inherited” feature, it’s OK if the feature had a precondition in the parent, but then the class must redeclare it with a clause `require else True`. A simple `require` without the `else` is not permitted in this case.

A “precursor” of an inherited routine is its version in a parent; there may be more than one as a result of feature merging and repeated inheritance.

End

Informative text

No specific validity rule limits our ability to include a **convert** mark in an **Alias** as long as it applies to a feature with one argument and an **Operator** alias. Of course in an example such as *your_integer + your_real* we expect the argument type **AT**, here **REAL**, to include a feature with the given operator, here **+**, and the target type **CT**, here **INTEGER**, to convert to **AT**. But we don’t require this of *all* types to which **CT** converts, because:

- This would have to be checked for every new type (since **CT** may convert to **AT** not only through its own “from” specification but also through a “to” specification in **AT**).
- In any case, it would be too restrictive: **INTEGER** may well convert to a certain type **AT** for which we don’t use target conversion.

Instead, the validity constraints will simply rule out individual *calls* that would require target conversion if the proper conditions are not met. For example if **REAL** did not have a function specifying **alias** “+” and accepting an integer argument, or if **INTEGER** did not convert to **REAL**, the expression would be invalid.

Remarkably, there is no need for any special validity rule to enforce these properties. All we’ll need is the definition of **target-converted form of a binary expression** in the discussion of expressions. The target-converted form of $x + y$ (or a similar expression for any other binary operator) is $x + y$ itself unless *both* of the following properties hold:

- The declaration of “+” for the type of x specifies **convert**.
- The type of y does **not** conform or convert to the type of the argument of the associated function, here **plus**, so that the usual interpretation of the expression as shorthand for x .**plus**(y) cannot possibly be valid. This is critical since we don’t want any ambiguity: either the usual interpretation or the targeted conversion should be valid, but not both.

Under these conditions the targeted-converted form is $(\{TY\} [x]) + y$, using as first operand the result of converting x to the type **TY** of y . Then:

- The target-converted form is only valid if **TY** has a feature with the “+” alias, and y is acceptable as an argument of this call. The beauty of this is that we don’t need any new validity rule: if any of this condition is not met, the normal validity rules on expressions (involving, through the notion of Equivalent Dot Form, the rules on calls) will make it illegal.
- We don’t need any specific semantic rule either: the normal semantic rules, applied to the target-converted form, yield exactly what we need.

End

8.16 Repeated inheritance

Informative text

Inheritance may be **multiple**: a class may have any number of parents. A more restrictive solution would limit the benefits of inheritance, so central to object-oriented software engineering.

Because of multiple inheritance, it is possible for a class to be a descendant of another in more than one way. This case is known as **repeated** inheritance; it raises interesting issues and yields useful techniques, which the following discussion reviews in detail.

End

8.16.1 Definition: Repeated inheritance, ancestor, descendant

Repeated inheritance occurs whenever (as a result of multiple inheritance) two or more of the ancestors of a class **D** have a common parent **A**.

D is then called a **repeated descendant** of **A**, and **A** a **repeated ancestor** of **D**.

8.16.2 Semantics: Repeated Inheritance rule

Let D be a class and B_1, \dots, B_n ($n \geq 2$) be parents of D based on classes having a common ancestor A . Let f_1, \dots, f_n be features of these respective parents, all having as one of their seeds the same feature f of A . Then:

- 1 Any subset of these features inherited by D under the same final name in D yields a single feature of D .
- 2 Any two of these features inherited under a different name yield two features of D .

Informative text

This is the basic rule allowing us to make sense and take advantage of inheritance, based on the programmer-controlled naming policy: inheriting two features under the same name yields a single feature, inheriting them under two different names yield two features.

End

8.16.3 Definition: Sharing, replication

A repeatedly inherited feature is **shared** if case 1 of the Repeated Inheritance rule applies, and **replicated** if case 2 applies.

8.16.4 Validity: Call Sharing rule

Validity code: *VMCS*

It is valid for a feature f repeatedly inherited by a class D from an ancestor A , such that f is shared under repeated inheritance and not redeclared, to involve a feature g of A other than as the feature of a qualified call if and only if g is, along the corresponding inheritance paths, also shared.

Informative text

If g were duplicated, there would be no way to know which version f should call, or evaluate for the assignment. The “selected” version, discussed below, is not necessarily the appropriate one.

End

8.16.5 Semantics: Replication Semantics rule

Let f and g be two features both repeatedly inherited by a class A and both replicated under the Repeated Inheritance rule, with two respective sets of different names: f_1 and f_2 , g_1 and g_2 .

If the version of f in D is the original version from A and either contains an unqualified call to g or (if f is an attribute) is the target of an assignment whose source involves g , the f_1 version will use g_1 for that call or assignment, and the f_2 version will use g_2 .

Informative text

This rule (which, unlike other semantic rules, clarifies a special case rather than giving the general semantics of a construct) tells us how to interpret calls and assignments if two separate replications have proceeded along distinct inheritance paths.

End

8.16.6 Syntax: Select clauses

Select \triangleq **select** *Feature_list*

Informative text

The **Select** subclause serves to resolve any ambiguities that could arise, in dynamic binding on polymorphic targets declared statically of a repeated ancestor’s type, when a feature from that type has two different versions in the repeated descendant.

End

8.16.7 Validity: Select Subclause rule

Validity code: **VMSS**

A **Select** subclause appearing in the parent part for a class *B* in a class *D* is valid if and only if, for every **Feature_name** *fname* in its **Feature_list**, *fname* is the final name in *D* of a feature that has two or more potential versions in *D*, and *fname* appears only once in the **Feature_list**.

Informative text

This rule restricts the use of **Select** to cases in which it is meaningful: two or more “potential versions”, a term which also has its own precise definition. We will encounter next, in the Repeated Inheritance Consistency constraint, the converse requirement that if there is such a conflict a **Select** *must* be provided.

End

8.16.8 Definition: Version

A feature *g* from a class *D* is a **version** of a feature *f* from an ancestor of *D* if *f* and *g* have a seed in common.

8.16.9 Definition: Multiple versions

A class *D* has *n* **versions** ($n \geq 2$) of a feature *f* of an ancestor *A* if and only if *n* of its features, all with different final names in *D*, are all versions of *f*.

8.16.10 Validity: Repeated Inheritance Consistency constraint

Validity code: **VMRC**

It is valid for a class *D* to have two or more versions of a feature *f* of a proper ancestor *A* if and only if it satisfies one of the following conditions:

- 1 There is at most one conformance path from *D* to *A*.
- 2 There are two or more conformance paths, and the **Parent** clause for exactly one of them in *D* has a **Select** clause listing the name of the version of *f* from the corresponding parent.

8.16.11 Definition: Dynamic binding version

For any feature *f* of a type *T* and any type *U* conforming to *T*, the **dynamic binding version** of *f* in *U* is the feature *g* of *U* defined as follows:

- 1 If *f* has only one version in *U*, then *g* is that feature.
- 2 If *f* has two or more versions in *U*, then the Repeated Inheritance Consistency constraint ensures that either exactly one conformance path exists from *U* to *T*, in which case *g* is the version of *f* in *U* obtained along that path, or that a **Select** subclause name a version of *f*, in which case *g* is that version.

8.16.12 Definition: Inherited features

Let *D* be a class. Let *precursors* be the list obtained by concatenating the lists of features of every parent of *D*; this list may contain duplicates in the case of repeated inheritance. The list *inherited* of **inherited features** of *D* is obtained from *precursors* as follows:

- 1 In the list *precursors*, for any set of two or more elements representing features that are repeatedly inherited in *D* under the same name, so that the Repeated Inheritance rule yields sharing, keep only one of these elements. The Repeated Inheritance Consistency constraint (sharing case) indicates that these elements must all represent the same feature, so that it does not matter which one is kept.
- 2 For every feature *f* in the resulting list, if *D* undefines *f*, replace *f* by a deferred feature with the same signature, specification and header comment.
- 3 In the resulting list, for any set of deferred features with the same final name in *D*, keep only one of these features, with assertions and header comment joined as per the Join Semantics rule. (Keep the signature, which the Join rule requires to be the same for all the features involved after possible redeclaration.)

- 4 In the resulting list, remove any deferred feature such that the list contains an effective feature with the same final name. (This is the case in which a feature *f*, inherited as effective, effects one or more deferred features: of the whole group, only *f* remains.)
- 5 All the features of the resulting list have different names; they are the inherited features of *D* in their parent forms. From this list, produce a new one by replacing any feature that *D* redeclares (through redefinition or effecting) with the result of the redeclaration, and retaining any other feature as it is.
- 6 The result is the list *inherited* of inherited features of *D*.

8.16.13 Semantics: Join-Sharing Reconciliation rule

If a class inherits two or more features satisfying both the conditions of sharing under the Repeated Inheritance rule and those of the Join rule, the applicable semantics is the Repeated Inheritance rule.

8.16.14 Definition: Precursor

A **precursor** of an inherited feature of final name *fname* is any parent feature — appearing in the list *precursors* obtained through case 1 of the definition of “Inherited features” — that the feature mergings resulting from the subsequent cases reduce into a feature of name *fname*.

8.16.15 Validity: Feature Name rule

Validity code: *VMFN*

It is valid for a feature *f* of a class *C* to have a certain final name if and only if it satisfies the following conditions:

- 1 No other feature of *C* has that same feature name.
- 2 If *f* is shared under repeated inheritance, its precursors all have either no *Alias* or the same alias.

Informative text

Condition 1 follows from other rules: the Feature Declaration rule, the Redeclaration rule and the rules on repeated inheritance. It is convenient to state it as a separate condition, as it can help produce clear error messages in some cases of violation.

Two feature names are “the same” if the lower-case version of their identifiers is the same.

The important notion in this condition is “**other feature**”, resulting from the above definition of “inherited features”. When do we consider *g* to be a feature “other” than *f*? This is the case whenever *g* has been declared or redeclared distinctly from *f*, unless the definition of inherited features causes the features to be merged into just one feature of *C*. Such merging may only happen as a result of sharing features under repeated inheritance, or of joining deferred features. Also, remember that if *C* redeclares an inherited feature (possibly resulting from the joining of two or more), this does not introduce any new (“other”) feature. This was explicitly stated by the definition of “introducing” a feature.

Condition 2 complements these requirements by ensuring that sharing doesn’t inadvertently give a feature more than one alias.

The Feature Name rule crowns the discussion of inheritance and feature adaptation by unequivocally implementing the No Overloading Principle: no two features of a class may have the same name. The only permissible case is when the name clash is apparent only, but in reality the features involved are all the same feature under different guises, resulting from a join or from sharing under repeated inheritance.

End

8.16.16 Validity: Name Clash rule

Validity code: *VMNC*

The following properties govern the names of the features of a class *C*:

- 1 It is invalid for *C* to introduce two different features with the same name.
- 2 If *C* introduces a feature with the same name as a feature it inherits as effective, it must rename the inherited feature.

- 3 If *C* inherits two features as effective from different parents and they have the same name, the class must also (except under sharing for repeated inheritance) remove the name clash through renaming.

Informative text

This is not a new constraint but a set of properties that follow from the Feature Name rule and other rules. Instead of Eiffel's customary "This is valid if and only if ..." style, more directly useful to the programmer since it doesn't just tell us how to mess things up but also how to produce guaranteeably *valid* software, the Name Clash rule is of the more discouraging form "You may not validly write ...". It does, however, highlight frequently applicable consequences of the naming policy, and compilers may take advantage of it to report naming errors.

End

8.17 Control structures

Informative text

The previous discussions have described the "bones" of Eiffel software: the module and type structure of systems. Here we begin studying the "meat": the elements that govern the execution of applications.

Control structures are the constructs used to schedule the run-time execution of instructions. There are four of them: sequencing (compound), conditional, multi-branch choice and loop. A complementary construct is the *Debug* instruction.

As made clear by the definition of "non-exception semantics" in the semantic rule for *Compound*, which indirectly governs all control structures (since all instructions are directly or indirectly part of a *Compound*), the default semantics assumes that none of the instructions executed as part of a control structure triggers an *exception*. If an exception does occur, the normal flow of control is interrupted, as described by the rules of exception handling in the discussion of this topic.

End

8.17.1 Semantics: Compound (non-exception) semantics

The effect of executing a *Compound* is:

- If it has zero instructions: to leave the state of the computation unchanged.
- If it has one or more instructions: to execute the first instruction of the *Compound*, then (recursively) to execute the *Compound* obtained by removing the first instruction.

This specification, the **non-exception semantics** of *Compound*, assumes that no *exception* is triggered. If the execution of any of the instructions triggers an exception, the Exception Semantics rule takes effect for the rest of the *Compound*'s instructions.

Informative text

Less formally, this means executing the constituent instructions in the order in which they appear in the *Compound*, each being started only when the previous one has been completed.

Note that a *Compound* can be empty, in which case its execution has no effect. This is useful for examples when refactoring the branches of a *Conditional*: you might temporarily remove all the instructions of the *Else_part*, but not the *Else_part* itself yet as you think it may be needed later.

End

8.17.2 Syntax: Conditionals

Conditional \triangleq **if** Then_part_list [*Else_part*] **end**

Then_part_list \triangleq {Then_part **elseif** ...}⁺

Then_part \triangleq Boolean_expression **then** Compound

Else_part \triangleq **else** Compound

8.17.3 Definition: Secondary part

The **secondary part** of a **Conditional** possessing at least one **elseif** is the **Conditional** obtained by removing the initial "if Then_part_list" and replacing the first **elseif** of the remainder by **if**.

8.17.4 Definition: Prevailing immediately

The execution of a **Conditional** starting with **if condition₁** is said to **prevail immediately** if **condition₁** has value true.

8.17.5 Semantics: Conditional semantics

The effect of a **Conditional** is:

- If it **prevails immediately**: the effect of the first **Compound** in its **Then_part_list**.
- Otherwise, if it has at least one **elseif**: the effect (recursively) of its secondary part.
- Otherwise, if it has an **Else** part: the effect of the **Compound** in that **Else** part.
- Otherwise: no effect.

Informative text

Like the instruction studied next, the **Conditional** is a "multi-branch" choice instruction, thanks to the presence of an arbitrary number of **elseif** clauses. These branches do not have equal rights, however; their conditions are evaluated in the order of their appearance in the text, until one is found to evaluate to true. If two or more conditions are true, the one selected will be the first in the syntactical order of the clauses.

End

8.17.6 Definition: Inspect expression

The **inspect expression** of a **Multi_branch** is the expression appearing after the keyword **inspect**.

8.17.7 Syntax: Multi-branch instructions

Multi_branch \triangleq **inspect** Expression [When_part_list] [Else_part] **end**

When_part_list \triangleq When_part⁺

When_part \triangleq **when** Choices **then** Compound

Choices \triangleq {Choice "; ...}⁺

Choice \triangleq Constant | Manifest_type | Constant_interval | Type_interval

Constant_interval \triangleq Constant ".." Constant

Type_interval \triangleq Manifest_type ".." Manifest_type

8.17.8 Definition: Interval

An **interval** is a **Constant_interval** or **Type_interval**.

8.17.9 Definition: Unfolded form of a multi-branch

To obtain the **unfolded form** of a **Multi_branch** instruction, apply the following transformations in the order given:

- 1 Replace every **constant inspect value** by its **manifest value**.
- 2 If the type *T* of the inspect expression is any **sized variant** of **CHARACTER**, **STRING** or **INTEGER**, replace every inspect value *v* by {*T*} *v*.
- 3 Replace every **interval** by its **unfolded form**.

Informative text

Step 2 enables us, with an inspect expression of a type such as **INTEGER_8**, to use constants in ordinary notation, such as **1**, rather than the heavier **{INTEGER_8} 1**. Unfolded form constructs this proper form for us. The rules on constants make this convention safe: a value that doesn't match the type, such as **1000** here, will cause a validity error.

End

8.17.10 Definition: Unfolded form of an interval

The **unfolded form** of an interval $a..b$ is the following (possibly empty) list:

- 1 If a and b are constants, both of either a character type, a string type or an integer type, and of manifest values va and vb : the list made up of all values i , if any, such that $va \leq i \leq vb$, using character, integer or lexicographical order respectively.
- 2 If a and b are both of type $TYPE[T]$ for some T , and have manifest values va and vb : the list containing every Manifest_type of the system conforming to vb and to which va conforms.
- 3 If neither of the previous two cases apply: an empty list.

Informative text

The “manifest value” of a constant is the value that has been declared for it, ignoring any Manifest_type: for example both `1` and `{INTEGER_8} 1` have the manifest value `1`.

The symbol `..` is not a special symbol of the language but an alias for a feature of the Kernel Library class PART_COMPARABLE, which for any partially or totally ordered set and yielding the set of values between a lower and an upper bound. Here, the bounds must be constant.

A note for implementers: type intervals such as `{U}..{T}`, denoting all types conforming to T and to which U conforms, may seem to raise difficult implementation issues: the set of types, which the unfolded form seems to require that we compute, is potentially large; the validity (Multi-Branch rule) requires that all types in the unfolded form be distinct, which seems to call for tricky computations of intersections between multiple sets; and all this may seem hard to reconcile with incremental compilation, since a type interval may include types from both our own software and externally acquired libraries, raising the question of what happens on delivery of a new version of such a library, possibly without source code. Closer examination removes these worries:

- There is no need actually to compute entire type intervals as defined by the unfolded form. Listing `{U}..{T}` simply means, when examining a candidate type Z , finding out whether Z conforms to T and U to Z .
- To ascertain that such a type interval does not intersect with another `{Y}..{X}`, the basic check is that Y does not conform to T and U does not conform to X .
- If we add a new set of classes and hence types to a previously validated system, a new case of intersection can only occur if either: a new type inherits from one of ours, a case that won't happen for a completely external set of reusable classes and, if it happens, should require re-validating since existing Multi_branch instructions may be affected; or one of ours inherits from a new type, which will happen only when we modify our software *after* receiving the delivery, and again should require normal rechecking.

End

8.17.11 Validity: Interval rule

Validity code: *VOIN*

An Interval is valid if and only if its unfolded form is not empty.

8.17.12 Definition: Inspect values of a multi-branch

The **inspect values** of a Multi_branch instruction are all the values listed in the Choices parts of the instruction's unfolded form.

Informative text

The set of inspect values may be infinite in the case of a string interval, but this poses no problem for either programmers or compilers, meaning simply that matches will be determined through lexicographical comparisons.

End

8.17.13 Validity: Multi-branch rule

Validity code: *VOMB*

A Multi_branch instruction is valid if and only if its unfolded form satisfies the following conditions.

- 1 Inspect values are all valid.
- 2 Inspect values are all constants.
- 3 The manifest values of any two inspect values are different.
- 4 If the inspect expression is of type *TYPE [T]* for some type *T*, all inspect values are types.
- 5 If case 4 does not apply, the inspect expression is one of the sized variants of *INTEGER*, *CHARACTER* or *STRING*.

8.17.14 Semantics: Matching branch

During execution, a **matching branch** of a *Multi_branch* is a *When_part wp* of its unfolded form, satisfying either of the following for the value *val* of its inspect expression:

- 1 *val* \sim *i*, where *i* is one of the non-*Manifest_type* inspect values listed in *wp*.
- 2 *val* denotes a *Manifest_type* listed among the choices of *wp*.

Informative text

The Multi-branch rule is designed to ensure that in any execution there will be at most one matching branch.

In case 1, we look for object equality, as expressed by \sim . Strings, in particular, will be compared according to the function *is_equal* of *STRING*. A void value, even if type-wise permitted by the inspect expression, will never have a matching branch.

In case 2, we look for an exact type match, not just conformance. For conformance, we have type intervals: to match types conforming to some *T*, use $\{NONE\}..{T}$; for types to which *T* conforms, use ${T}..{ANY}$.

End

8.17.15 Semantics: Multi-Branch semantics

Executing a *Multi_branch* with a matching branch consists of executing the *Compound* following the **then** in that branch. In the absence of matching branch:

- 1 If the *Else_part* is present, the effect of the *Multi_branch* is that of the *Compound* appearing in its *Else_part*.
- 2 Otherwise the execution triggers an exception of type *BAD_INSPECT_VALUE*.

8.17.16 Syntax: Loops

Loop \triangleq *Initialization*
 [*Invariant*]
Exit_condition
Loop_body
 [*Variant*]
end

Initialization \triangleq **from** *Compound*

Exit_condition \triangleq **until** *Boolean_expression*

Loop_body \triangleq **loop** *Compound*

8.17.17 Semantics: Loop semantics

The effect of a *Loop* is the effect of executing the *Compound* of its *Initialization*, then its *Loop_body*.

The effect of executing a *Loop_body* is:

- If the *Boolean_expression* of the *Exit_condition* evaluates to true: no effect (leave the state of the computation unchanged).
- Otherwise: the effect of executing the *Compound* clause, followed (recursively) by the effect of executing the *Loop_body* again in the resulting state.

8.17.18 Syntax: Debug instructions

`Debug` \triangleq `debug ["("Key_list ")"] Compound` `end`

8.17.19 Semantics: Debug semantics

A language processing tool must provide an option that makes it possible to enable or disable `Debug` instructions, both globally and for individual keys of a `Key_list`. Such an option may be settable for an entire system, or for individual classes, or both.

Letter case is not significant for a debug key.

The effect of a `Debug` instruction depends on the mode that has been set for the `current class`:

- If the `Debug` option is on generally, or if the instruction includes a `Key_list` and the option is on for at least one of the keys in the list, the effect of the `Debug` instruction is that of its `Compound`.
- Otherwise the effect is that of a null instruction.

8.18 Attributes

Informative text

Attributes are one of the two kinds of feature.

When, in the declaration of a class, you introduce an attribute of a certain type, you specify that, for every instance of the class that may exist at execution time, there will be an associated value of that type.

Attributes are of two kinds: **variable** and **constant**. The difference affects what may happen at run time to the attribute's values in instances of the class: for a variable attribute, the class may include routines that, applied to a particular instance, will change the value; for a constant attribute, the value is the same for every instance, and cannot be changed at run time.

End

8.18.1 Syntax: Attribute bodies

`Attribute` \triangleq `attribute` `Compound`

Informative text

The `Compound` is empty in most usual cases, but it is required for an attribute of an attached type (including the case of an expanded type) that does not provide `default_create` as a creation procedure; it will then serve to initialize the corresponding field, on first use for any particular object, if that use occurs prior to an explicit initialization. To set that first value, assign to `Result` in the `Compound`.

Such a `Compound` is executed at most once on any particular object during a system execution.

End

8.18.2 Validity: Manifest Constant rule

Validity code: *VQMC*

A declaration of a feature `f` introducing a `manifest_constant` is valid if and only if the `Manifest_constant` `m` used in the declaration matches the type `T` declared for `f` in one of the following ways:

- 1 `m` is a `Boolean_constant` and `T` is `BOOLEAN`.
- 2 `m` is a `Character_constant` and `T` is one of the `sized_variants` of `CHARACTER` for which `m` is a valid value.
- 3 `m` is an `Integer_constant` and `T` is one of the `sized_variants` of `INTEGER` for which `m` is a valid value.
- 4 `m` is a `Real_constant` and `T` is one of the `sized_variants` of `REAL` for which `m` is a valid value.
- 5 `m` is a `Manifest_string` and `T` is one of the `sized_variants` of `STRING` for which `m` is a valid value.

- 6 m is a **Manifest_type**, of the form $\{Y\}$ for some type Y , and T is **TYPE [X]** for some **stand-alone type** X to which Y conforms.

Informative text

The “valid values” are determined by each basic type’s semantics; for example 1000 is a valid value for **INTEGER_16** but not for **INTEGER_8**.

In case 6, we require the type listed in a **Manifest_type {Y}** to be *constant*, meaning that it does not involve any formal generic parameter or anchored type, as these may represent different types in different generic derivations or different descendants of the original class. This would not be suitable for a constant attribute, which must have a single, well-defined value.

End

8.19 Objects, values and entities

Informative text

The execution of an Eiffel system consists of creating, accessing and modifying **objects**.

The following presentation discusses the structure of objects and how they relate to the syntactical constructs that denote objects in software texts: **expressions**. At run time, an expression may take on various *values*; every value is either an object or a reference to an object.

Among expressions, **entities** play a particular role. An entity is an identifier (name in the software text), meant at execution time to denote possible values. Some entities are **read-only**: the execution can’t change their initial value. Others, called **variables**, can take on successive values during execution as a result of such operations as creation and assignment.

The description of objects and their properties introduces the *dynamic model* of Eiffel software execution: the run-time structures of the data manipulated by an Eiffel system.

End

8.19.1 Semantics: Type, generating type of an object; generator

Every run-time object is a **direct instance** of exactly one **stand-alone type** of the system, called the **generating type** of the object, or just “the type of the object” if there is no ambiguity.

The **base class** of the generating type is called the object’s **generating class**, or **generator** for short.

8.19.2 Definition: Reference, void, attached, attached to

A **reference** is a value that is either:

- **Void**, in which case it provides no more information.
- **Attached**, in which case it gives access to an object. The reference is said to be **attached to** that object, and the object attached to the reference.

8.19.3 Semantics: Object principle

Every non-**void** value is either an object or a reference **attached** to an object.

8.19.4 Definition: Object semantics

Every run-time object has either **copy semantics** or **reference semantics**.

An object has copy semantics if and only if its **generating type** is an **expanded type**.

Informative text

This property determines the role of the object when used as source of an assignment: with copy semantics, it will be copied onto the target; with reference semantics, a reference will be reattached to it.

End

8.19.5 Definition: Non-basic class, non-basic type, field

Any class other than the basic types is said to be a **non-basic class**. Any type whose base class is non-basic is a **non-basic type**, and its instances are **non-basic objects**.

A direct instance of a non-basic type is a sequence of zero or more values, called **fields**. There is one field for every attribute of the type's base class.

8.19.6 Definition: Subobject, composite object

Any expanded field of an object is a **subobject** of that object.

An object that has a non-basic subobject is said to be **composite**.

8.19.7 Definition: Entity, variable, read-only

An **entity** is an Identifier, or one of two reserved words (**Current** and **Result**), used in one of the following roles:

- 1 Final name of an attribute of a class.
- 2 Local variable of a routine or Inline_agent, including **Result** for a query.
- 3 Formal argument of a routine or inline agent.
- 4 Object Test local.
- 5 **Current**, the predefined entity used to represent a reference to the current object (the target of the latest not yet completed routine call).

Names of non-constant attributes and local variables are **variable** entities, also called just **variables**. Constant attributes, formal arguments, Object Test locals and **Current** are **read-only** entities.

Informative text

Two kinds of operation, creation and reattachment, may modify the value of a variable (a non-constant attribute, part of category 1, or local variable, category 2. In the other four cases — constant attributes, formal arguments (3), Object Test locals (4) and **Current** (5) — you may not directly modify the entities, hence the name *read-only* entity.

The term “*constant* entity” wouldn't do, not so much because you can modify the corresponding objects but because *read-only* entities (other than constant attributes) do change at run time: a qualified call reattaches **Current**, and any routine call reattaches the formal arguments.

Result appearing in the Postcondition of a constant attribute cannot be changed at execution time, but for simplicity is considered part of local variables in all cases anyway.

End

8.19.8 Syntax: Entities and variables

Entity \triangleq Variable | Read_only

Variable \triangleq Variable_attribute | Local

Variable_attribute \triangleq Feature_name

Local \triangleq Identifier | **Result**

Read_only \triangleq Formal | Constant_attribute | **Current**

Formal \triangleq Identifier

Constant_attribute \triangleq Feature_name

8.19.9 Validity: Entity rule

Validity code: *VEEN*

An occurrence of an entity *e* in the text of a class *C* (other than as the feature of a qualified call) is valid if and only if it satisfies one of the following conditions:

- 1 *e* is **Current**.
- 2 *e* is the final name of an attribute of *C*.

- 3 *e* is the local variable **Result**, and the occurrence is in a **Feature_body**, **Postcondition** or **Rescue** part of an **Attribute_or_routine** text for a **query** or an **Inline_agent** whose **signature** includes a result type.
- 4 *e* is **Result** appearing in the **Postcondition** of a **constant attribute**'s declaration.
- 5 *e* is listed in the **Identifier_list** of an **Entity_declaration_group** in a **Local_declarations** part of a feature or **Inline_agent** *fa*, and the occurrence is in a **Local_declarations**, **Feature_body** or **Rescue** part for *fa*.
- 6 *e* is listed in the **Identifier_list** of an **Entity_declaration_group** in a **Formal_arguments** part for a routine *r*, and the occurrence is in a **declaration for** *r*.
- 7 *e* is listed in the **Identifier_list** of an **Entity_declaration_group** in the **Agent_arguments** part of an **Agent** *a*, and the occurrence is in the **Agent_body** of *a*.
- 8 *e* is the **Object-Test Local** of an **Object_test**, and the occurrence is in its scope.

Informative text

“Other than as feature of a qualified call” excludes from the rule any attribute, possibly of another class, used as feature of a qualified call: in *a.b* the rule applies to *a* but not to *b*. The constraint on *b* is the General Call rule, requiring *b* to be the name of a feature in *D*'s base class.

End

8.19.10 Validity: Variable rule

Validity code: *VEVA*

A **Variable** entity *v* is valid in a class *C* if and only if it satisfies one of the following conditions:

- 1 *v* is the **final name** of a **variable attribute** of *C*.
- 2 *v* is the final name of a **local variable** of the immediately enclosing routine or agent.

8.19.11 Definition: Self-initializing type

A type is **self-initializing** if it is one of:

- 1 A detachable type.
- 2 A **self-initializing formal parameter**.
- 3 An attached type (including expanded types and, as a special case of these, basic types) whose **creation procedures** include a **version** of **default_create** from **ANY** available for creation to *C*.

Informative text

A self-initializing type enables us to define a default initialization value:

- Use **Void** for a detachable type (case 1, the easiest but also the least interesting)
- Execute a creation instruction with the applicable version of **default_create** for the most interesting case: 3, attached types, including expanded types. This case also covers basic types, which all have a default value given by the following rule.

A “self-initializing formal parameter” (case 2) is a generic parameter, so we don't exactly know which one of these three semantics will apply; but we do require, through the Generic Derivation rule, that any attached type used as actual generic parameter be self-initializing, meaning in this case that it will provide **default_create**.

In the definition, the “creation procedures” of a *type* are the creation procedures of its base *class* or, for a formal generic parameter, its “constraining creators”, the features listed as available for creation in its constraining type.

The more directly useful notion is that of a self-initializing *variable*, appearing below.

The term “self-initializing” is justified by the following semantic rule, specifying the actual initialization values for every self-initializing type.

End

8.19.12 Semantics: Default Initialization rule

Every self-initializing type *T* has a **default initialization value** as follows:

- 1 For a detachable type: a void reference.
- 2 For a self-initializing attached type: an object obtained by creating an instance of *T* through *default_create*.
- 3 For a self-initializing formal parameter: for every generic derivation, (recursively) the default initialization value of the corresponding actual generic parameter.
- 4 For *BOOLEAN*: the boolean value false.
- 5 For a sized variant of *CHARACTER*: null character.
- 6 For a sized variant of *INTEGER*: integer zero.
- 7 For a sized variant of *REAL*: floating-point zero.
- 8 For *POINTER*: a null pointer.
- 9 For *TYPED_POINTER*: an object representing a null pointer.

Informative text

This rule is the reason why everyone loves self-initializing types: whenever execution catches an entity that hasn't been explicitly set, it can (and, thanks to the Entity Semantics rule, will) set it to a well-defined default value. This idea gains extra flexibility, in the next definition, through the notion of attributes with an explicit initialization.

End

8.19.13 Definition: Self-initializing variable

A variable is **self-initializing** if one of the following holds:

- 1 Its type is a self-initializing type.
- 2 It is an attribute declared with an *Attribute* part such that the entity *Result* is properly set at the end of its *Compound*.

Informative text

If a variable is self-initializing, we don't need to worry about finding it with an undefined value at execution time: if it has not yet been the target of an attachment operation, automatic initialization can take over and set it to a well-defined default value. That value is, in case 1, the default value for its type, and in case 2 the result of the attribute's own initialization. That initialization must ensure that *Result* is "properly set" as defined next (partly recursively from the above definition).

End

8.19.14 Definition: Evaluation position, precedes

An **evaluation position** is one of:

- In a *Compound*, one of its *Instruction* components.
- In an *Assertion*, one of its *Assertion_clause* components.
- In either case, a special **end position**.

A position *p* **precedes** a position *q* if they are both in the same *Compound* or *Assertion*, and either:

- *p* and *q* are both *Instruction* or *Assertion_clause* components, and *p* appears before *q* in the corresponding list.
- *q* is the end position and *p* is not.

Informative text

This notion is needed to ensure that entities are properly set before use.

In a compound *i1; i2; i3* we have four positions; *i1* precedes *i2*, *i3* and the end position, and so on.

The relation as defined only applies to **first-level** components of the compound: if *i2* itself contains a compound, for example if it is of the form **if c then i4; i5 end**, then *i4* is not an evaluation position of the outermost compound, and so has no “precedes” relation with any of *i1*, *i2* and *i3*.

End

8.19.15 Definition: Setter instruction

A **setter instruction** is an assignment or creation instruction.

If *x* is a variable, a setter instruction is a **setter for *x*** if its assignment target or creation target is *x*.

8.19.16 Definition: Properly set variable

At an evaluation position *ep* in a class *C*, a variable *x* is **properly set** if one of the following conditions holds:

- 1 *x* is self-initializing.
- 2 *ep* is an evaluation position of the **Compound** of a feature or **Inline_agent** of the **Internal form**, one of whose instructions precedes *ep* and is a setter for *x*.
- 3 *x* is a variable attribute, and is (recursively) properly set at the end position of every creation procedure of *C*.
- 4 *ep* is an evaluation position in a **Compound** that is part of an instruction *ep'*, itself belonging to a **Compound**, and *x* is (recursively) properly set at position *ep'*.
- 5 *ep* is in a **Postcondition** of a routine or **Inline_agent** of the **Internal form**, and *x* is (recursively) properly set at the end position of its **Compound**.
- 6 *ep* is an **Assertion_clause** containing **Result** in the **Postcondition** of a constant attribute

Informative text

The key cases are 2, particularly useful for local variables but also applicable to attributes, and 3, applicable to attributes when we cannot deduce proper initialization from the enclosing routine but find that every creation procedure will take care of it. Case 4 accounts for nested compounds. For assertions other than postconditions, which cannot use variables other than attributes, 3 is the only applicable condition. The somewhat special case 6 is a consequence of our classification of **Result** among local variables even in the **Postcondition** of a constant attribute.

As an artefact of the definition’s phrasing, every variable attribute is “properly set” in any effective routine of a deferred class, since such a class has no creation procedures. This causes no problem since a failure to set the attribute properly will be caught, in the validity rule below, for versions of the routine in effective descendants.

End

8.19.17 Validity: Variable Initialization rule

Validity code: VEV1

It is valid for an **Expression**, other than the target of an **Assigner_call**, to be also a **Variable** if it is properly set at the evaluation position defined by the closest enclosing **Instruction** or **Assertion_clause**.

Informative text

This is the fundamental requirement guaranteeing that the value will be defined if needed.

Because of the definition of “properly set”, this requirement is pessimistic: some examples might be rejected even though a “smart” compiler might be able to prove, by more advanced control and data flow analysis, that the value will always be defined. But then the same software might be rejected by another compiler, less “smart” or simply using different criteria. On purpose, the definition limits itself to basic schemes that all compilers can implement.

If one of your software elements is rejected because of this rule, it’s a sign that your algorithms fail to initialize a certain variable before use, or at least that the proper initialization is not clear enough. To correct the problem, you may:

- Add a version of **default_create** to the class, as creation procedure.

- Give the attribute a specific initialization through an explicit **Attribute** part that sets **Result** to the appropriate value.

End

8.19.18 Definition: Variable setting and its value

A **setting** for a variable x is any one of the following run-time events, defining in each case the **value** of the setting:

- 1 Execution of a **setter for x** . (*Value*: the object attached to x by the setter, or a void reference if none.)
- 2 If x is a **variable attribute** with an **Attribute** part: evaluation of that part, implying execution of its **Compound**. (*Value*: the object attached to **Result** at the end position of that **Compound**, or a void reference if none.)
- 3 If the type T of x is **self-initializing**: assignment to x of T 's default initialization value. (*Value*: that initialization value.)

Informative text

As a consequence of case 2, an attribute a that is self-initializing through an **Attribute** part ap is *not* set until execution of ap has reached its end position. In particular, it is not invalid (although definitely unusual and perhaps strange) for the instructions ap to use the value a : as with a recursive call in a routine, this will start the computation again at the beginning of ap . For attributes as for routines, this raises the risk of infinite recursion (perhaps higher for attributes since they have no arguments) and it is the programmer's responsibility to avoid this by ensuring that before a recursive call the context will have sufficiently changed to ensure eventual termination. No language rule can ensure this (in either the routine or attribute cases) since this would amount to solving the "halting problem", a provably impossible task.

Another consequence of the same observation is that if the execution of ap triggers an exception, and hence does not reach its end position, any later attempt to access a will also restart the execution of ap from the beginning. This might trigger the same exception, or succeed if the conditions of the execution have changed.

End

8.19.19 Definition: Execution context

At any time during execution, the current **execution context** for a variable is the period elapsed since:

- 1 For an attribute: the creation of the current object.
- 2 For a local variable: the start of execution of the current routine.

8.19.20 Semantics: Variable Semantics

The value produced by the run-time evaluation of a **variable x** is:

- 1 If the execution context has previously executed at least one setting for x : the value of the latest such setting.
- 2 Otherwise, if the type T of x is **self-initializing**: assignment to x of T 's default initialization value, causing a setting of x .
- 3 Otherwise, if x is a **variable attribute** with an **Attribute** part: evaluation of that part, implying execution of its **Compound** and hence a setting for x .
- 4 Otherwise, if x is **Result** in the **Postcondition** of a constant attribute: the value of the attribute.

Informative text

This rule is phrased so that the order of the first three cases is significant: if there's already been an assignment, no self-initialization is possible; and if T has a default value, the **Attribute** part won't be used.

The Variable Initialization rule ensures that one of these cases will apply, so that x will always have a well-defined result for evaluation. This property was our main goal, and its achievement concludes the discussion of variable semantics.

End

8.19.21 Semantics: Entity Semantics rule

Evaluating an entity yields a **value** as follows:

- 1 For **Current**: a value attached to the current object.
- 2 For a formal argument of a routine or **Inline_agent**: the value of the corresponding actual at the time of the current call.
- 3 For a constant attribute: the value of the associated **Manifest_constant** as determined by the Manifest Constant Semantics rule.
- 4 For an Object-Test Local: as determined by the Object-Test Local Semantics rule.
- 5 For a variable: as determined by the Variable Semantics rule.

Informative text

This rule concludes the semantics of entities by gathering all cases. It serves as one of the cases of the semantics of expressions, since an entity can be used as one of the forms of **Expression**.

The Object-Test Local Semantics rule appears in the discussion of the **Object_test** construct.

End

8.20 Creating objects

Informative text

The dynamic model, whose major properties were reviewed in the preceding presentations, is highly flexible; your systems may create objects and attach them to entities at will, according to the demands of their execution. The following discussion explores the two principal mechanisms for producing new objects: the **Creation_instruction** and its less frequently encountered sister, the **Creation_expression**.

A closely related mechanism — **cloning** — exists for duplicating objects. This will be studied separately, with the mechanism for copying the contents of an object onto another.

The creation constructs offer considerable flexibility, allowing you to rely on language-defined initialization mechanisms for all the instances of a class, but also to override these defaults with your own conventions, to define any number of alternative initialization procedures, and to let each creation instruction provide specific values for the initialization. You can even instantiate an entity declared of a generic type — a non-trivial problem since, for x declared of type **G** in a class **C[G]**, we don't know what actual type **G** denotes in any particular case, and how one creates and initializes instances of that type.

In using all these facilities, you should never forget the methodological rule governing creation, as expressed by the following principle.

End

8.20.1 Semantics: Creation principle

Any execution of a creation operation must produce an object that satisfies the invariant of its generating class.

Informative text

Such is the theoretical role of creation: to make sure that any object we create starts its life in a state satisfying the corresponding invariant. The various properties of creation, reviewed next, are designed to ensure this principle.

End

8.20.2 Definition: Creation operation

A **creation operation** is a creation instruction or expression.

8.20.3 Validity: Creation Precondition rule

Validity code: *VGCP*

A **Precondition** of a routine *r* is **creation-valid** if and only if its unfolded form *uf* satisfies the following conditions:

- 1 The predefined entity **Current** does not appear in *uf*.
- 2 No **Unqualified_call** appears in *uf*.
- 3 Every feature whose final name appears in the *uf* is available to every class to which *r* is available for creation.

Informative text

This definition is not itself a validity constraint, but is used by condition 5 of the Creation Clause rule below; giving it a code as for a validity constraint enables compilers to provide a precise error message in case of a violation.

Requiring preconditions to be creation-valid will ensure that a creation procedure doesn't try to access, in the object being created, fields whose properties are not guaranteed before initialization.

The definition relies on the "unfolded form" of an assertion, which reduces it to a boolean expression with clauses separated by **and then**. Because the unfolded form uses the Equivalent Dot Form, condition 3 also governs the use of operators: with *plus alias* "+", the expression *a + b* will be acceptable only if the feature *plus* is available for creation as stated.

End

8.20.4 Syntax: Creators parts

Creators \triangleq **Creation_clause**⁺

Creation_clause \triangleq **create** [**Clients**] [**Header_comment**] **Creation_procedure_list**

Creation_procedure_list \triangleq {**Creation_procedure** ";..."}⁺

Creation_procedure \triangleq **Feature_name**

8.20.5 Definition: Unfolded Creators part of a class

The **unfolded creators part** of a class *C* is a **Creators** defined as:

- 1 If *C* has a **Creators** part *c*.
- 2 If *C* is **deferred**: an empty **Creators** part.
- 3 Otherwise, a **Creators** part built as follows, *dc_name* being the final name in *C* of its version of default_create from *ANY*:

create
dc_name

Informative text

For generality the definition is applicable to any class, even though for a deferred class (case 2) it would be invalid to include a **Creators** part. This causes no problem since the rules never refer to a deferred class actually extended with its unfolded creators part.

Case 3 reflects the convention that an absent **Creators** part stands for **create dc_name** — normally **create default_create**, but **dc_name** may be another name if the class or one of its proper ancestors has renamed **default_create**.

End

8.20.6 Validity: Creation Clause rule

Validity code: **VGCC**

A **Creation_clause** in the **unfolded creators part** of a class **C** is valid if and only if it satisfies the following conditions, the last four for every **Feature_name cp_name** in the clause's **Feature_list**:

- 1 **C** is **effective**.
- 2 **cp_name** appears only once in the **Feature_list**.
- 3 **cp_name** is the final name of some procedure **cp** of **C**.
- 4 **cp** is not a **once routine**.
- 5 The precondition of **cp**, if any, is **creation-valid**.

Informative text

As a result of conditions 1 and 4, a creation procedure may only be of the **do** form (the most common case) or **External**.

The prohibition of **once** creation procedures in condition 4 is a consequence of the Creation principle: with a once procedure, the first object created would satisfy the invariant (assuming the creation procedure is correct), but subsequent creation instructions would not execute the call, and hence would limit themselves to the default initializations, which might not ensure the invariant.

As a corollary of condition 4, a class that has no explicit **Creators** part may not redefine **default_create** into a once routine, or inherit **default_create** as a once routine from one of its deferred parents. (Effective parents would themselves violate the condition and hence be invalid.)

End

8.20.7 Definition: Creation procedures of a class

The **creation procedures** of a class are all the features appearing in any **Creation_clause** of its **unfolded creators part**.

Informative text

If there is an explicit **Creators** part, the creation procedures are the procedures listed there. Otherwise there is only one creation procedure: the class's version of **default_create**.

The following property is a consequence of the definitions of “unfolded creators part” and “creation procedures of a class”.

End

8.20.8 Creation procedure property

An **effective class** has at least one **creation procedure**.

Informative text

Those explicitly listed if any, otherwise **default_create**.

End

8.20.9 Definition: Creation procedures of a type

The **creation procedures** of a type **T** are:

- 1 If **T** is a **Formal_generic_name**, the **constraining creators for T**.
- 2 Otherwise, the **creation procedures** of **T**'s **base class**.

Informative text

The definition of case 2 is not good enough for case 1, because in the scheme **class** *D* [*G* → *CONST* **create** *cp1*, *cp2*, ... **end**] it would give us, as creation procedures of *G*, the creation procedures of *CONST*, and what we want is something else: the set of procedures *cp1*, *cp2*, ... specifically listed after *CONST* — the “*constraining creators for G*”. These are indeed procedures of *CONST*, but they are not necessarily *creation* procedures of *CONST*, especially since *CONST* can be deferred. What matters is that they must be creation procedures in any instantiatable descendant of *CONST* used as actual generic parameter for *G*.

End

8.20.10 Definition: Available for creation; general creation procedure

A creation procedure of a class *C*, listed in a *Creation_clause* *cc* of *C*'s unfolded creators part, is **available for creation** to the descendants of the classes given in the *Clients* restriction of *cc*, if present, and otherwise to all classes.

If there is no *Clients* restriction, the procedure is said to be a **general creation procedure**.

8.20.11 Syntax: Creation instructions

Creation_instruction \triangleq **create** [*Explicit_creation_type*] *Creation_call*

Explicit_creation_type \triangleq "{ *Type* }"

Creation_call \triangleq *Variable* [*Explicit_creation_call*]

Explicit_creation_call \triangleq ". " *Unqualified_call*

8.20.12 Definition: Creation target, creation type

The **creation target** (or just “target” if there is no ambiguity) of a *Creation_instruction* is the *Variable* of its *Creation_call*.

The **creation type** of a creation instruction, denoting the type of the object to be created, is:

- The *Explicit_creation_type* appearing (between braces) in the instruction, if present.
- Otherwise, the type of the instruction's target.

8.20.13 Semantics: Creation Type theorem

The creation type of a creation instruction is always effective.

8.20.14 Definition: Unfolded form of a creation instruction

Consider a *Creation_instruction* *ci* of creation type *CT*. The **unfolded form** of *ci* is a creation instruction defined as:

- 1 If *ci* has an *Explicit_creation_call*, then *ci* itself.
- 2 Otherwise, a *Creation_instruction* obtained from *ci* by making the *Creation_call* explicit, using as feature name the final name in *CT* of *CT*'s version of *ANY*'s *default_create*.

8.20.15 Validity: Creation Instruction rule

Validity code: VGCI

A *Creation_instruction* of creation type *CT*, appearing in a class *C*, is valid if and only if it satisfies the following conditions:

- 1 *CT* conforms to the target's type.
- 2 The feature of the *Creation_call* of the instruction's unfolded form is available for creation to *C*.
- 3 That *Creation_call* is argument-valid.
- 4 *CT* is generic-creation-ready.

Informative text

In spite of its compactness, the Creation Instruction rule suffices in fact to capture all properties of creation instructions thanks to the auxiliary definitions of “*creation type*”, “*unfolded form*” of both a *Creation_instruction* and a *Creators* part, “*available for creation*” and others. The rule captures in particular the following cases:

- The procedure-less form **create** *x* is valid only if *CT*'s version of *default_create* is available for creation to *C*; this is because in this case the unfolded form of the instruction is **create** *x*, *dc_name*, where *dc_name* is *CT*'s name for *default_create*. On *CT*'s side the condition implies that there is either no *Creators* part (so that *CT*'s own unfolded form lists *dc_name* as creation procedure), or that it has one making it available for creation to *C* (through a *Creation_clause* with either no *Clients* specification or one that lists an ancestor of *C*).
- If *CT* is a *Formal_generic_name*, its creation procedures are those listed in the **create** subclause after the constraint. So **create** *x* is valid if and only if the local version of *default_create* is one of them, and **create** *x*, *cp* (...) only if *cp* is one of them.
- If *CT* is generically derived, and its base class needs to perform creation operations on targets of some of the formal generic types, the last condition (generic-creation readiness) ensures that the corresponding actual parameters are equipped with appropriate creation procedures.

The very brevity of this rule may make it less suitable for one of the applications of validity constraints: enabling compilers to produce precise diagnostics in case of errors. For this reason a complementary rule, conceptually redundant since it follows from the Creation Instruction rule, but providing a more explicit view, appears next. It is stated in “*only if*” style rather than the usual “*if and only if*” of other validity rules, since it limits itself to a set of necessary validity conditions.

End

8.20.16 Validity: Creation Instruction properties

Validity code: *VGCP*

A *Creation_instruction* *ci* of creation type *CT*, appearing in a class *C*, is valid only if it satisfies the following conditions, assuming *CT* is not a *Formal_generic_name* and calling *BCT* the base class of *CT* and *dc* the version of *ANY*'s *default_create* in *BCT*:

- 1 *BCT* is an effective class.
- 2 If *ci* includes a *Type* part, the type it lists (which is *CT*) conforms to the type of the instruction's target.
- 3 If *ci* has no *Creation_call*, then *BCT* either has no *Creators* part or has one that lists *dc* as one of the procedures available to *C* for creation.
- 4 If *BCT* has a *Creators* part which doesn't list *dc*, then *ci* has a *Creation_call*.
- 5 If *ci* has a *Creation_call* whose feature *f* is not *dc*, then *BCT* has a *Creators* part which lists *f* as one of the procedures available to *C* for creation.
- 6 If *ci* has a *Creation_call*, that call is argument-valid.

If *CT* is a *Formal_generic_name*, the instruction is valid only if it satisfies the following conditions:

- 7 *CT* denotes a constrained generic parameter.
- 8 The *Constraint* for *CT* specifies one or more procedures as constraining creators.
- 9 If *ci* has no *Creation_call*, one of the constraining creators is the *Constraint*'s version of *default_create* from *ANY*.
- 10 If *ci* has a *Creation_call*, one of the constraining creators is the feature of the *Creation_call*.

Informative text

Compiler writers may refer, in error messages, to either these “Creation Instruction Properties” or the earlier “Creation Instruction rule” of which they are consequences. For the language definition, **the official rule is the Creation Instruction rule**, which provides a necessary and sufficient set of validity conditions.

End

8.20.17 Semantics: Creation Instruction Semantics

The effect of a creation instruction of target *x* and creation type *TC* is the effect of the following sequence of steps, in order:

- 1 If there is not enough memory available for a new direct instance of *TC*, trigger an exception of type *NO_MORE_MEMORY* in the routine that attempted to execute the instruction. The remaining steps do not apply in this case.
- 2 Create a new direct instance of *TC*, with reference semantics if *CT* is a reference type and copy semantics if *CT* is an expanded type.
- 3 Call, on the resulting object, the feature of the Unqualified_call of the instruction's unfolded form.
- 4 Attach *x* to the object.

Informative text

The rules requires the *effect* described by this sequence of steps; it does not require that the implementation literally carry out the steps. In particular, if the target is expanded and has already been set to an object value, the implementation (in the absence of cycles in the client relation between expanded classes) may **not have to allocate new memory**; instead, it may be able simply to reuse the memory previously allocated to that object. (Because only expanded types conform to an expanded type, no references may exist to the previous object, and hence it is not necessary to preserve its value.) In that case, there will always at step 1 be “enough memory available for a new direct instance” — the memory being reused — and so the exception cannot happen.

One might expect, between steps 2 and 3, a step of *default initialization* of the fields of the new object, since this is the intuitive semantics of the language: integers initialized to zero, detachable references to void etc. There is no need, however, for such a step since the Variable Semantics rule implies that an attribute or other variable, unless previously set by an explicit attachment, is automatically set on first access. The rule implies for example that an integer field will be set to zero. More generally, the semantics of the language guarantees that in every run-time circumstance any object field and local variable, even if never explicitly assigned to yet, always has a well-defined value when the computation needs it.

About step 3, remember that the notion of “unfolded form” allows us to consider that every creation instruction has an Unqualified_call; in the procedure-less form **create** *x*, this is a call to default_create.

Also note the order of steps: attachment to the target *x* is the last operation. Until then, *x* retains its earlier value, void if *x* is a previously unattached reference.

In step 2, “not enough memory available” is a precise notion (the definition appears below); it means that even after possible *garbage collection* the memory available for the system's execution is not sufficient for the requested object creation.

End

8.20.18 Syntax: Creation expressions

Creation_expression \triangleq **create** Explicit_creation_type [Explicit_creation_call]

8.20.19 Definition: Properties of a creation expression

The **creation type** and **unfolded form** of a creation expression are defined as for a creation instruction.

8.20.20 Validity: Creation Expression rule

Validity code: **VGCE**

A **Creation_expression** of creation type **CT**, appearing in a class **C**, is valid if and only if it satisfies the following conditions:

- 1 The feature of the **Creation_call** of the expression's unfolded form is available for creation to **C**.
- 2 That **Creation_call** is argument-valid.
- 3 **CT** is generic-creation-ready.

8.20.21 Validity: Creation Expression Properties

Validity code: **VG CX**

A **Creation_expression** **ce** of creation type **CT**, appearing in a class **C**, is valid only if it satisfies the following conditions, assuming **CT** is not a **Formal_generic_name** and calling **BCT** the base class of **CT** and **dc** the version of **ANY**'s default_create in **BCT**:

- 1 **BCT** is an effective class.
- 2 If **ce** has no **Explicit_creation_call**, then **BCT** either has no **Creators** part or has one that lists **dc** as one of the procedures available to C for creation.
- 3 If **BCT** has a **Creators** part which doesn't list **dc**, then **ce** has an **Explicit_creation_call**.
- 4 If **ce** has an **Explicit_creation_call** whose feature **f** is not **dc**, then **BCT** has a **Creators** part which lists **f** as one of the procedures available to C for creation.
- 5 If **ce** has an **Explicit_creation_call**, that call is argument-valid.

If **CT** is a **Formal_generic_name**, the expression is valid only if it satisfies the following conditions:

- 6 **CT** denotes a constrained generic parameter.
- 7 The **Constraint** for **CT** specifies one or more procedures as constraining creators.
- 8 If **ce** has no **Creation_call**, one of the constraining creators is the **Constraint**'s version of default_create from **ANY**.
- 9 If **ce** has a **Creation_call**, one of the constraining creators is the feature of the **Creation_call**.

Informative text

As with the corresponding "Creation Instruction Properties", this is not an independent rule but a set of properties following from previous constraints, expressed with more detailed requirements that may be useful for error reporting by compilers.

End

8.20.22 Semantics: Creation Expression Semantics

The value of a creation expression of creation type **TC** is — except if step 1 below triggers an exception, in which case the expression has no value — a value attached to a new object as can be obtained through the following sequence of steps:

- 1 If there is not enough memory available for a new direct instance of **TC**, trigger an exception of type **NO_MORE_MEMORY** in the routine that attempted to execute the expression. In this case the expression has no value and the remaining steps do not apply.
- 2 Create a new direct instance of **TC**, with reference semantics if **CT** is a reference type and copy semantics if **CT** is an expanded type.
- 3 Call, on the resulting object, the feature of the **Unqualified_call** of the expression's unfolded form.

Informative text

The notes appearing after the Creation Instruction Semantics rule also apply here.

End

8.20.23 Definition: Garbage Collection, not enough memory available

Authors of Eiffel implementation are required to provide **garbage collection**, defined as a mechanism that can reuse for allocating new objects the memory occupied by unreachable objects, guaranteeing the following two properties:

- 1 *Consistency*: the garbage collector never reclaims an object unless it is unreachable.
- 2 *Completeness*: no allocation request for an object of a certain size *s* will fail if there exists an unreachable object of size $\geq s$.

Not enough memory available for a certain size *s* means that even after possible application of the garbage collection mechanism the memory available to the program is not sufficient for allocating an object of size *s*.

8.21 Comparing and duplicating objects

Informative text

The just studied **Creation** instruction is the basic language mechanism for obtaining new objects at run time; it produces fresh direct instances of a given class, initialized from scratch.

Sometimes you will need instead to copy the contents of an existing object onto those of another. This is the **copying** operation.

A variant of copying is **cloning**, which produces a fresh object by duplicating an existing one.

For both copying and cloning, the default variants are "shallow", affecting only one object, but **deep** versions are available to duplicate an object structure recursively.

A closely related problem is that of *comparing* two objects for shallow or deep equality.

The copying, cloning and comparison operations rely on only one language construct (the object equality operator \sim) and are entirely defined through language constructs but through routines that developer-defined classes inherit from the universal class ANY. This makes it possible, through feature redefinitions, to adapt the semantics of copying, cloning and comparing objects to the specific properties of any class.

End

8.21.1 Object comparison features from ANY

The features whose **contract views** appear below are provided by class ANY.

default_is_equal (other: like Current)

- Is *other* attached to object field-by-field equal
- to current object?

ensure

- same_type: **Result implies same_type (other)**
- symmetric: **Result = other.default_is_equal (Current)**
- consistent: **Result implies is_equal (other)**

is_equal (other: ? like Current)

- Is *other* attached to object considered equal
- to current object?

ensure

- same_type: **Result implies same_type (other)**
- symmetric: **Result = other.is_equal (Current)**

consistent: *default_is_equal (other)* **implies Result**

The original version of *is_equal* in *ANY* has the same effect as *default_is_equal*.

Informative text

These are the two basic object comparison operations. The difference is that *default_is_equal* is frozen, always returning the value of field-by-field identity comparison (for non-void *other*); any class may, on the other hand, redefine *is_equal*, in accordance with the pre- and postcondition, to reflect a more specific notion of equality.

Both functions take an argument of an attached type, so there is no need to consider void values.

End

8.21.2 Syntax: Equality expressions

Equality \triangleq Expression Comparison Expression

Comparison \triangleq "=" | "/=" | "~" | "/~"

8.21.3 Semantics: Equality Expression Semantics

The *Boolean_expression* $e \sim f$ has *value* true if and only if the *values* of e and f are both *attached* and such that *e.is_equal (f)* holds.

The *Boolean_expression* $e = f$ has *value* true if and only if the values of e and f are one of:

- 1 Both void.
- 2 Both attached to the same object with *reference semantics*.
- 3 Both attached to objects with *copy semantics*, and such that $e \sim f$ holds.

Informative text

The form with \sim always denotes object equality. The form with $=$ denotes reference equality if applicable, otherwise object equality. Both rely, for object equality, on function *is_equal* — the version that can be redefined locally in any class to account for a programmer-defined notion of object equality adapted to the specific semantics of the class.

End

8.21.4 Semantics: Inequality Expression Semantics

The expression $e \neq f$ has *value* true if and only if $e = f$ has *value* false.

The expression $e \neq \sim f$ has *value* true if and only if $e \sim f$ has *value* false.

8.21.5 Copying and cloning features from *ANY*

The features whose *contract views* appear below are provided by class *ANY* as *secret features*.

copy (other: ? like Current)

- Update current object using fields of object
- attached to *other*, to yield equal objects.

require

- exists: *other* \neq Void
- same_type: *other.same_type (Current)*

ensure

- equal: *is_equal (other)*

frozen *default_copy (other: ? like Current)*

- Update current object using fields of object
- attached to *other*, to yield identical objects.

require

- exists: *other* \neq Void

same_type: *other.same_type* (*Current*)

ensure
equal: *default_is_equal* (*other*)

frozen cloned: like *Current*
-- New object equal to current object
-- (relies on *copy*)

ensure
equal: *is_equal* (*Result*)

frozen default_cloned: like *Current*
-- New object equal to current object
-- (relies on *default_copy*)

ensure
equal: *default_is_equal* (*Result*)

The original versions of *copy* and *cloned* in *ANY* have the same effect as *default_copy* and *default_cloned* respectively.

Informative text

Procedure *copy* is called in the form *x.copy* (*y*) and overrides the fields of the object attached to *x*. Function *cloned* is called as *x.cloned* and returns a new object, a “clone” of the object attached to *x*. These features can be adapted to a specific notion of copying adapted to any class, as long as they produce a result equal to the source, in the sense of the — also redefinable — function *is_equal*. You only have to redefine *copy*, since *cloned* itself is frozen, with the guarantee that it will follow any redefined version of *copy*; the semantics of *cloned* is to create a new object and apply *copy* to it.

In contrast, *default_copy* and *default_cloned*, which produce field-by-field identical copies of an object, are frozen and hence always yield the original semantics as defined in *ANY*.

All these features are **secret in their original class** *ANY*. The reason is that exporting copying and cloning may violate the intended semantics of a class, and concretely its invariant. For example the correctness of a class may rely on an invariant property such as

some_circumstance implies (*some_attribute = Current*)

stating that under *some_circumstance* (a boolean property) the field corresponding to *some_attribute* is cyclic (refers to the current object itself). Copying or cloning an object will usually not preserve such a property. The class should then definitely not export *default_copy* and *default_cloned*, and should not export *copy* and *cloned* unless it redefines *copy* in accordance with this invariant; such redefinition may not be possible or desirable. Because these features are secret by default, software authors must decide, class by class, whether to re-export them.

End

8.21.6 Deep equality, copying and cloning

The feature *is_deep_equal* of class *ANY* makes it possible to compare object structures recursively; the features *deep_copy* and *deep_cloned* duplicate an object structure recursively.

Informative text

The default versions of the earlier features — *default_is_equal*, *default_copy*, *default_cloned* and the original versions of their non-*default* variants — are “shallow”: they compare or copy only one source object. The “deep” versions recursively compare or copy entire object structures.

Detailed descriptions of the “deep” features appear in the specification of [ELKS](#).

End

8.22 Attaching values to entities

Informative text

At any instant of a system's execution, every entity of the system has a certain attachment status: it is either attached to a certain object, or void (attached to no object). Initially, all entities of reference types are void; one of the effects of a **Creation instruction** is to attach its target to an object.

The attachment status of an entity may change one or more times during system execution through a **attachment** operations, in particular:

- The association of an actual argument of a routine to the corresponding formal argument at the time of a call.
- The **Assignment** instruction, which may attach an entity to a new object, or remove the attachment.

The validity and semantic properties of these two mechanisms are essentially the same; we study them jointly here.

End

8.22.1 Definition: Reattachment, source, target

A **reattachment** operation is one of:

- 1 An **Assignment** $x := y$, then y is the attachment's source and x its target.
- 2 The run-time association, during the execution of a routine call, of an actual argument (the source) to the corresponding formal argument (the target).

Informative text

We group assignment and argument passing into the same category, reattachment, because their validity and semantics are essentially the same:

- Validity in both cases is governed by the type system: the source must conform to the target's type, or at least convert to it. The Conversion principle guarantees that these two cases are exclusive.
- The semantics in both cases is to attach the target to the value of the source or a copy of that value.

End

8.22.2 Syntax: Assignments

Assignment \triangleq Variable **":="** Expression

8.22.3 Validity: Assignment rule

Validity code: *VBAR*

An **Assignment** is valid if and only if its source expression is compatible with its target entity.

Informative text

To be "compatible" means to conform or convert.

This also applies to actual-formal association: the actual argument in a call must conform or convert to the formal argument. The applicable rule is **argument validity**, part of the general discussion of call validity.

End

8.22.4 Semantics: Reattachment principle

After a reattachment to a target entity t of type TT , the object attached to t , if any, is of a type conforming to TT .

8.22.5 Semantics: Attaching an entity, attached entity

Attaching an entity *e* to an object *O* is the operation ensuring that the value of *e* becomes **attached to** *O*.

Informative text

Although it may seem tautological at first, this definition simply relates the two terms “attach”, denoting an operation that can change an entity, and “attached to an object”, denoting the state of such an entity — as determined by such operations. These are key concepts of the language since:

- A reattachment operation (see next) may “attach” its target to a certain object as defined by the semantic rule; a creation operation creates an object and similarly “attaches” its creation target to that object.
- Evaluation of an entity, per the Entity Semantics rule, uses (partly directly, partly by depending on the Variable Semantics rule and through it on the definition of “value of a variable setting”) the object *attached* to that entity. This is only possible by ensuring, through other rules, that prior to any such attempt on a specific entity there will have been operations to “attach” the entity or make it void.

End

8.22.6 Semantics: Reattachment Semantics

The effect of a reattachment of source expression *source* and target entity *target* is the effect of the first of the following steps whose condition applies:

- 1 If *source* converts to *target*: perform a conversion attachment from *source* to *target*.
- 2 If the value of *source* is a void reference: make *target*’s value void as well.
- 3 If the value of *source* is attached to an object with copy semantics: create a clone of that object, if possible, and attach *target* to it.
- 4 If the value of *source* is attached to an object with reference semantics: attach *target* to that object.

Informative text

As with other semantic rules describing the “effect” of a sequence of steps, only that effect counts, not the exact means employed to achieve it. In particular, the creation of a clone in step 3 is — as also noted in the discussion of creation — often avoidable in practice if the target is expanded and already initialized, so that the instruction can reuse the memory of the previous object.

Case 1 indicates that a conversion, if applicable, overrides all other possibilities. In those other cases, if follows from the Assignment rule that *source* must **conform to** *target*.

Case 2 is, from the validity rules, possible only if both *target* and *source* are declared of *detachable* types.

In case 3, a “clone” of an object is obtained by application of the function *cloned* from *ANY*; expression conformance ensures that *cloned* is available (exported) to the type of *target*; otherwise, cloning could produce an inconsistent object.

The cloning might be impossible for lack of memory, in which case the semantics of the cloning operation specifies triggering an exception, of type *NO_MORE_MEMORY*. As usual with exceptions, the rest of case 3 does not then apply.

In case 4 we simply reattach a reference. Because of the validity rules (no reference type conforms to an expanded type), the target must indeed be of an reference type.

This rule defines the *effect* of a construct through a sequence of cases, looking for the first one that matches. As usual with semantic rules, this only specifies the result, but does not imply that the implementation must try all of them in order.

End

8.22.7 Semantics: Assignment Semantics

The effect of a reassignment $x := y$ is determined by the Reattachment Semantics rule, with source y and target x .

Informative text

The other cases where Reattachment Semantics applies is actual-formal association, per step 5 of the General Call rule.

On the other hand, the semantics of `Object_test`, a construct which also allows a `Read_only` entity to denote the same value as an expression, is simple enough that it does not need to refer to reattachment.

End

8.22.8 Definition: Dynamic type

The **dynamic type** of an expression x , at some instant of execution, is the type of the object to which x is attached, or `NONE` if x is void.

8.22.9 Definition: Polymorphic expression; dynamic type and class sets

An expression that has two or more possible dynamic types is said to be **polymorphic**.

The set of possible dynamic types for an expression x is called the **dynamic type set** of x . The set of base classes of these types is called the **dynamic class set** of x .

8.22.10 Syntax: Assigner calls

`Assigner_call` \triangleq Expression `":"=` Expression

Informative text

The left-hand side is surprisingly general: any expression. The validity rule will constrain it to be of a form that can be interpreted as a qualified call to a query, such as $x.a$, or $x.f(i, j)$; but the syntactic form can be different, using for example bracket syntax as in $a[i, j] := x$.

You could even use operator syntax, as in

$$a + b := c$$

assuming that, in the type of a , the function *plus alias* "+" has been defined with an assigner command, maybe a procedure `subtract`. Then the left side $a + b$ is just an abbreviation for the query call

$$a.plus(b)$$

and the `Assigner_call` is just an abbreviation for the procedure call

$$a.subtract(c, b)$$

End

8.22.11 Validity: Assigner Call rule

Validity code: `VBAC`

An `Assigner_call` of the form $target := source$, where $target$ and $source$ are expressions, is valid if and only if it satisfies the following conditions:

- 1 $source$ is compatible with $target$.
- 2 The Equivalent Dot Form of $target$ is a qualified `Object_call` whose feature has an assigner command.

8.22.12 Semantics: Assigner Call semantics

The effect of an `Assigner_call` $target := source$, where the Equivalent Dot Form of $target$ is $x.f$ or $x.f(args)$ and f has an assigner command p , is, respectively, $x.p(source)$ or $x.p(source, args)$.

Informative text

This confirms that the construct is just an abbreviation for a procedure call.

End

8.23 Feature call

Informative text

In Eiffel's model of computation, the fundamental way to do something with an object is to apply to it an operation which — because the model is class-based, and behind every run-time object lurks some class of the system's text — must be a feature of the appropriate class.

This is feature call, one of the most important constructs in Eiffel's object-oriented approach, and the topic of the following discussions.

End

8.23.1 Validity: Call Use rule

Validity code: **VUCN**

A Call of feature f denotes:

- 1 If f is a query (attribute or a function): an expression.
- 2 If f is a procedure: an instruction.

8.23.2 Syntax: Feature calls

Call \triangleq Object_call | Non_object_call

Object_call \triangleq [Target "."] Unqualified_call

Unqualified_call \triangleq Feature_name [Actuals]

Target \triangleq Local | Read_only | Call | Parenthesized_target

Parenthesized_target \triangleq "(" Expression ")"

Non_object_call \triangleq "{" Type "}" "." Unqualified_call

Informative text

A call is most commonly of the form $a.b\dots$ where $a, b\dots$ are features, possibly with arguments. Target allows a Call to apply to an explicit target object (rather than the current object); it can itself be a Call, allowing multidot calls. Other possible targets are a local variable, a Read_only (including formal arguments and Current) a "non-object call" (studied below), or a complex expression written as a Parenthesized_target ($\{...\}$).

End

8.23.3 Syntax: Actual arguments

Actuals \triangleq "(" Actual_list ")"

Actual_list \triangleq {Expression "," ...}⁺

8.23.4 Definition: Unqualified, qualified call

An Object_call is **qualified** if it has a Target, **unqualified** otherwise.

Informative text

In equivalent terms, a call is "unqualified" if and only if it consists of just an Unqualified_call component.

The call $f(a)$ is unqualified, $x.f(a)$ is qualified.

Another equivalent definition, which does not explicitly refer to the syntax, is that a call is qualified if it contains one or more dots, unqualified if it has no dots — counting only dots at the dot level, not those that might appear in arguments; for example $f(a.b)$ is unqualified.

End

8.23.5 Definition: Target of a call

Any Object_call has a **target**, defined as follows:

- 1 If it is qualified: its Target component.
- 2 If it is unqualified: **Current**.

Informative text

The target is an expression; in $a(b, c).d$ the target is $a(b, c)$ and in $(| a(b, c) + x |).d$ the target (case 1) is $a(b, c) + x$. In a multidot case the target includes the **Call** deprived of its last part, for example $x.f(args).g$ in $x.f(args).g.h(args1)$.

End

8.23.6 Definition: Target type of a call

Any **Call** has a **target type**, defined as follows:

- 1 For an **Object_call**: the type of its target. (In the case of an **Unqualified_call** this is the current type.)
- 2 For a **Non_object_call** having a type T as its **Type** part: T .

8.23.7 Definition: Feature of a call

For any **Call** the “**feature of the call**” is defined as follows:

- 1 For an **Unqualified_call**: its **Feature_name**.
- 2 For a **qualified_call** or **Non_object_call**: (recursively) the feature of its **Unqualified_call** part.

Informative text

Case 1 tells us that the feature of $f(args)$ is f and the feature of g , an **Unqualified_call** to a feature without arguments, is g .

The term is a slight abuse of language, since f and g are feature names rather than features. The actual feature, deduced from the semantic rules given below and involving dynamic binding, is the dynamic feature of the call.

It follows from case 2 that the feature of a qualified call $x.f(args)$ is f . The recursive phrasing addresses the multidot case: the feature of $x.f(args).g.h(args1)$ is h .

End

8.23.8 Definition: Imported form of a Non_object_call

The **imported form** of a **Non_object_call** of **Type** T and feature f appearing in a class C is the **Unqualified_call** built from the original **Actuals** if any and, as **feature of the call**, a fictitious new feature added to C and consisting of the following elements:

- 1 A name different from those of other features of C .
- 2 A **Declaration_body** obtained from the **Declaration_body** of f by replacing every type by its deanchored form, then applying the generic substitution of T .

Informative text

This definition in “unfolded” style allows us to view $\{T\}.f(args)$ appearing in a class C as if it were just $f(args)$, an **Unqualified_call**, but appearing in C itself, assuming we had moved f over — “imported” it — to C .

In item 2 we use the “deanchored form” of the argument types and result, since a type **like** a that makes sense in T would be meaningless in C . As defined in the discussion of anchored types, the deanchored version precisely removes all such local dependencies, making the type understandable instead in any other context.

End

8.23.9 Validity: Non-Object Call rule

Validity code: *VUNO*

A **Non_object_call** of **Type** T and feature $fname$ in a class C is valid if and only if it satisfies the following conditions:

- 1 $fname$ is the final name of a feature f of T .

- 2 f is available to C .
- 3 f is either a constant attribute or an external feature whose assertions, if any, use neither **Current** nor any unqualified calls.
- 4 The call's imported form is a valid Unqualified_call.

Informative text

Condition 2 requires f to have a sufficient export status for use in C ; there will be a similar requirement for Object_call. Condition 3 is the restriction to constants and externals. Condition 4 takes care of the rest by relying on the rules for Unqualified_call.

End

8.23.10 Semantics: Non-Object Call Semantics

The effect of a Non_object_call is that of its imported form.

8.23.11 Validity: Export rule

Validity code: *VUEX*

An Object_call appearing in a class C , with $fname$ as the feature of the call, is **export-valid** for C if and only if it satisfies the following conditions.

- 1 $fname$ is the final name of a feature of the target type of the call.
- 2 If the call is qualified, that feature is available to C .

Informative text

For an unqualified call f or $f(args)$, only condition 1 is applicable, requiring simply (since the target type of an unqualified call is the current type) that f be a feature, immediate or inherited, of the current class.

For a qualified call $x.f$ with x of type T , possibly with arguments, condition 2 requires that the base class of T make the feature available to C : export it either generally or selectively to C or one of its ancestors. (Through the Non-Object Call rule this also governs the validity of a Non_object_call $\{T\}.f$.)

As a consequence, $s(...)$ might be permitted and $x.s(...)$ invalid, even if x is **Current**. The semantics of qualified and unqualified calls is indeed slightly different; in particular, with invariant monitoring on, a qualified call will — even with **Current** as its target — check the class invariant, but an unqualified call won't.

End

8.23.12 Validity: Export Status principle

The export status of a feature f :

- Constrains all qualified calls $x.f(...)$, including those in which the type of x is the current type, or is **Current** itself.
- Does not constrain unqualified calls.

Informative text

This is a validity property, but it has no code since it is not a separate rule, just a restatement for emphasis of condition 2 of the Export rule.

End

8.23.13 Validity: Argument rule

Validity code: *VUAR*

An **export-valid** call of target type ST and feature $fname$ appearing in a class C where it denotes a feature sf is **argument-valid** if and only if it satisfies the following conditions:

- 1 The number of actual arguments is the same as the number of formal arguments declared for sf .

- 2 Every actual argument of the call is compatible with the corresponding formal argument of *sf*.

Informative text

Condition 2 is the fundamental type rule on argument passing, which allowed the discussion of direct reattachment to treat **Assignment** and actual-formal association in the same way. An expression is *compatible* with an entity if its type either conforms or converts to the entity's type.

End

8.23.14 Validity: Target rule

Validity code: *VUTA*

An **Object_call** is **target-valid** if and only if either:

- 1 It is unqualified.
- 2 Its target is an attached expression.

Informative text

Unqualified calls (case 1) are always target-valid since they are applied to the current object, which by construction is not void.

For the target expression *x* to be “*attached*”, in case 2, means that the program text guarantees — statically, that is to say through rules enforced by compilers — that *x* will never be void at run time. This may be because *x* is an entity declared as attached (so that the validity rules ensure it can never be attached a void value) or because the context of the call precludes voidness, as in if *x* /= **Void then x.f(...)** **end** for a local variable *x*. The precise definition will cover all these cases.

End

8.23.15 Validity: Class-Level Call rule

Validity code: *VUCC*

A call of target type *ST* is **class-valid** if and only if it is export-valid, argument-valid and target-valid.

8.23.16 Definition: Void-Unsafe

A language processing tool may, as a temporary migration facility, provide an option that waives the target validity requirement in class validity. Systems processed under such an option are **void-unsafe**.

Informative text

Void-unsafe systems are not valid Eiffel systems. Since void safety was not enforced by previous versions of Eiffel, compilers may need, all the same, to provide an option that temporarily lifts this requirement. Including the notion of “void-unsafe” in the language definition enforces a consistent way for various compilers to provide this transition facility.

End

8.23.17 Definition: Target Object

The **target object** of an execution of an **Object_call** is:

- 1 If the call is qualified: the object attached to its target.
- 2 If it is unqualified: the current object.

8.23.18 Semantics: Failed target evaluation of a void-unsafe system

In the execution of an (invalid) system compiled in void-unsafe mode through a language processing tool offering such a migration option, an attempt to execute a call triggers, if it evaluates the target to a void reference, an exception of type *VOID_TARGET*.

8.23.19 Definition: Dynamic feature of a call

Consider an execution of a call of feature *fname* and target object *O*. Let *ST* be its target type and *DT* the type of *O*. The **dynamic feature** of the call is the dynamic binding version in *DT* of the feature of name *fname* in *ST*.

Informative text

Behind the soundness of this definition stands a significant part of the validity machinery of the language:

- The rules on reattachment imply that *DT* conforms to *ST*.
- The Export rule imply that *fname* is the name of a feature of *ST* (meaning a feature of the base class of *ST*).
- As a consequence, this feature has a version in *DT*; it might have several, but the definition of “dynamic binding version” removes any ambiguity.

Combining the last two semantic definitions enables the rest of the semantic discussion to take for granted, for any execution of a qualified call, that we know both the target object and the feature to execute. In other words, we’ve taken care of the two key parts of *Object_call* semantics, although we still have to integrate a few details and special cases.

End

8.23.20 Definition: Freshness of a once routine call

During execution, a call whose feature is a once routine *r* is **fresh** if and only if every feature call started so far satisfies any of the following conditions:

- 1 It did not use *r* as dynamic feature.
- 2 It was in a different thread, and *r* has the once key “*THREAD*” or no once key.
- 3 Its target was not the current object, and *r* has the once key “*OBJECT*”.
- 4 After it was started, a call was executed to one of the refreshing features of *onces* from *ANY*, including among the keys to be refreshed at least one of the once keys of *r*.

Informative text

Case 2 indicates that “once per thread” is the default in the absence of an explicit once key

End

8.23.21 Definition: Latest applicable target and result of a non-fresh call

The **latest applicable target** of a non-fresh call to a once routine *df* to a target object *O* is the last value to which it was attached in the call to *df* most recently started on:

- 1 If *df* has the once key “*OBJECT*”: *O*.
- 2 Otherwise, if *df* has the once key “*THREAD*” or no once key: any target in the current thread.
- 3 Otherwise: any target in any thread.

If *df* is a function, the **latest applicable result** of the call is the last value returned by a fresh call using as target object its latest applicable target.

8.23.22 Semantics: Once Routine Execution Semantics

The effect of executing a once routine *df* on a target object *O* is:

- 1 If the call is fresh: that of a non-once call made of the same elements, as determined by Non-once Routine Execution Semantics.
- 2 If the call is not fresh and the last execution of *f* on the latest applicable target triggered an exception: to trigger again an identical exception. The remaining cases do not then apply.
- 3 If the call is not fresh and *df* is a procedure: no further effect.

- 4 If the call is not fresh and *df* is a function: to attach the local variable **Result** to the latest applicable result of the call.

Informative text

Case 2 is known as “*once an once exception, always a once exception*”. If a call to a once routine yields an exception, then all subsequent calls for the same applicable target, which would normally yield no further effect (for a procedure, case 3) or return the same value (for a function, case 4) should follow the same general idea and, by re-triggering the exception, repeatedly tell the client — if the client is repeatedly asking — that the requested effect or value is impossible to provide.

There is a little subtlety in the definition of “latest applicable target” as used in case 4. For a once function that has already been evaluated (is not fresh), the specification does not state that subsequent calls return the result of the first, but that they yield the value of the predefined entity **Result**. Usually this is the same, since the first call returned its value through **Result**. But if the function is **recursive**, a new call may start before the first one has terminated, so the “result of the first call” would not be a meaningful notion. The specification states that in this case the recursive call will return whatever value the first call has obtained so far for **Result** (starting with the default initialization). A recursive once function is a bit bizarre, and of little apparent use, but no validity constraint disallows it, and the semantics must cover all valid cases.

End

8.23.23 Semantics: Current object, current routine

At any time during the execution of a system there is a **current object CO** and a **current routine cr** defined as follows:

- 1 At the start of the execution: **CO** is the root object and *cr* is the root procedure.
- 2 If *cr* executes a qualified call: the call’s target object becomes the new current object, and its dynamic feature becomes the new current routine. When the qualified call terminates, the earlier current object and routine resume their roles.
- 3 If *cr* executes an unqualified call: the current object remains the same, and the dynamic feature of the call becomes the current routine for the duration of the call as in case 2.
- 4 If *cr* starts executing any construct whose semantics does not involve a call: the current object and current routine remain the same.

8.23.24 Semantics: Current Semantics

The value of the predefined entity **Current** at any time during execution is the current object if the current routine belongs to an expanded class, and a reference to the current object otherwise.

8.23.25 Semantics: Non-Once Routine Execution Semantics

The effect of executing a non-once routine *df* on a target object O is the effect of the following sequence of steps:

- 1 If *df* has any local variables, including **Result** if *df* is a function, save their current values if any call to *df* has been started but not yet terminated.
- 2 Execute the body of *df*.
- 3 If the values of any local variables have been saved in step 1, restore the variables to their earlier values.

8.23.26 Semantics: General Call Semantics

The effect of an Object_call of feature *sf* is, in the absence of any exception, the effect of the following sequence of steps:

- 1 Determine the target object O through the applicable definition.
- 2 Attach **Current** to **O**.
- 3 Determine the dynamic feature df of the call through the applicable definition.

- 4 For every actual argument *a*, if any, in the order listed: obtain the value *v* of *a*; then if the type of *a* converts to the type of the corresponding formal in *sf*, replace *v* by the result of the applicable conversion. Let *arg_values* be the resulting sequence of all such *v*.
- 5 Attach every formal argument of *df* to the corresponding element of *arg_values* by applying the Reattachment Semantics rule.
- 6 If the call is qualified and class invariant monitoring is on, evaluate the class invariant of *O*'s base type on *O*.
- 7 If precondition monitoring is on, evaluate the precondition of *df*.
- 8 If *df* is not an attribute, not a once routine and not external, apply Non-Once Routine Execution Semantics to *O* and *df*.
- 9 If *df* is a once routine, apply the Once Routine Execution Semantics to *O* and *df*.
- 10 If *df* is an external routine, execute that routine on the actual arguments given, if any, according to the rules of the language in which it is written.
- 11 If *df* is a self-initializing attribute and has not yet been initialized, initialize it through the Default Initialization rule.
- 12 If the call is qualified and class invariant monitoring is on, evaluate the class invariant of *O*'s base type on *O*.
- 13 If postcondition monitoring is on, evaluate the postcondition of *df*.

An exception occurring during any of these steps causes the execution to skip the remaining parts of this process and instead handle the exception according to the Exception Semantics rule.

8.23.27 Definition: Type of a Call used as expression

Consider a call denoting an expression. Its **type** with respect to a type *CT* of base class *C* is:

- 1 For an unqualified call, its feature *f* being a query of *CT*: the result type of the version of *f* in *C*, adapted through the generic substitution of *CT*.
- 2 For a qualified call *a.e* of Target *a*: (recursively) the type of *e* with respect to the type of *a*.
- 3 For a Non_object_call: (recursively) the type of its imported form.

8.23.28 Semantics: Call Result

Consider a Call *c* whose feature is a query. An execution of *c* according to the General Call Semantics yields a **call result** defined as follows, where *O* is the target object determined at step 1 of the rule and *df* the dynamic feature determined at step 3:

- 1 If *df* is a non-external, non-once function: the value attached to the local variable **Result** of *df* at the end of step 2 of Non-Once Routine Execution Semantics.
- 2 If *df* is a once function: the value attached to **Result** as a result of the application of Once Routine Execution Semantics.
- 3 If *df* is an attribute: the corresponding field in *O*.
- 4 If *df* is an external function: the result returned by the function according to the external language's rule.

8.23.29 Semantics: Value of a call expression

The **value** of a Call *c* used as an expression is, at any run-time moment, the result of executing *c*.

8.24 Eradicating void calls

Informative text

In the object-oriented style of programming the basic unit of computation is a qualified feature call *x.f(args)*

which applies the feature *f*, with the given arguments *args*, to the object attached to *x*. But *x* can be a reference, and that reference can be void. Then there is *no* object attached to *x*. An attempt to execute the call would fail, triggering an exception.

If permitted to occur, void calls are a source of instability and crashes in object-oriented programs. For other potential run-time accidents such as type mismatches, the compilation process spots the errors and refuses to generate executable code until they've all been corrected. Can we do the same for void calls?

Eiffel indeed provides a coordinated set of techniques that guarantee the absence of void calls at execution time. The actual rules are specific conditions of more general validity constraints — in particular on attachment and qualified calls — appearing elsewhere; in the following discussion we look at them together from the viewpoint of ensuring their common goal: precluding void calls.

The basic idea is simple. Its the combination of three rules:

- A qualified call $x.f(args)$ is **target-valid** — a required part of being plain valid — if the type of x is **attached**, “Attached” is here a static property, deduced from the declaration of x (or, if it is a complex expression, of its constituents).
- A reference type with a name in the usual form, T , is attached. To obtain a **detachable** type — meaning that *Void* is a valid value — use $?T$.
- The validity rules ensure that attached types — those without a $?$ — deserve their name: an entity declared as $x: T$ can never take on a void value at execution time. In particular, you may not assign to x a detachable value, or if x is a formal argument to a routine you may not call it with a detachable actual. (With a detachable target, the other way around, you are free to use an attached or detachable source.)

End

8.24.1 Syntax: Object test

$\text{Object_test} \triangleq \{ \text{Identifier} : \text{Type} \} \text{ Expression}$

Informative text

An *Object_test* of the form $\{x: T\} \text{exp}$, where *exp* is an expression, T is a type and x is a name different from those of all entities of the enclosing context, is a boolean-valued expression; its value is true if and only *exp* is attached to an instance of T (hence, non-void). In addition, evaluating the expression has the effect of letting x denote that value of *exp* over the execution of a neighboring part of the text known as the **scope** of the *Object_test*. For example, in **if** $\{x: T\} \text{exp}$ **then** $c1$ **else** $c2$ **end** the scope of the *Object_test* is the compound in the **then** part, $c1$. Within $c1$, you may use x as a *Read_only* entity, knowing that it has the value *exp* had on evaluation of the *Object_test*, that this value is of type T , and that it cannot be changed during the execution of $c1$. The following rules define these notions precisely.

End

8.24.2 Definition: Object-Test Local

The **Object-Test Local** of an *Object_test* is its *Identifier* component.

8.24.3 Validity: Object Test rule

Validity code: *VUOT*

An *Object_test ot* of the form $\{x: T\} \text{exp}$ is valid if and only if it satisfies the following conditions:

- 1 x does not have the same lower name as any feature of the enclosing class, or any formal argument or local variable of any enclosing feature or *Inline_agent*, or, if *ot* appears in the scope of any other *Object_test*, its Object-Test Local.
- 2 T is an attached type.

Informative text

Condition 2 reflects the intent of an *Object_test*: to test whether an expression is *attached* to an instance of a given type. It would make no sense then to use a detachable type.

End

8.24.4 Definition: Conjunctive, disjunctive, implicative; Term, semistrict term

Consider an **Operator_expression** e of boolean type, which after resolution of any ambiguities through precedence rules can be expressed as $a_1 \S a_2 \S \dots \S a_n$ for $n \geq 1$, where \S represents boolean operators and every a_i , called a **term**, is itself a valid **Boolean_expression**. Then e is:

- **Conjunctive** if every \S is either **and** or **and then**.
- **Disjunctive** if every \S is either **or** or **or else**.
- **Implicative** if $n = 2$ and \S is **implies**.

A term a_i is **semistrict** if in the corresponding form it is followed by a **semistrict operator**.

8.24.5 Definition: Scope of an Object-Test Local

The scope of the **Object-Test Local** of an **Object_test** ot includes any applicable program element from the following:

- 1 If ot is a **semistrict term** of a **conjunctive expression**: any subsequent terms.
- 2 If ot is a term of an **implicative expression**: the next term.
- 3 If **not** ot is a semistrict term of a disjunctive expression e : any subsequent terms.
- 4 If ot is a term of a conjunctive expression serving as the **Boolean_expression** in the **Then_part** in a **Conditional**: the corresponding **Compound**.
- 5 If **not** ot is a term of a **disjunctive expression** serving as the **Boolean_expression** in the **Then_part** in a **Conditional**: any subsequent **Then_part** and **Else_clause**.
- 6 If **not** ot is a term of a disjunctive expression serving as the **Exit_condition** in a **Loop**: the **Loop_body**.
- 7 If ot is a term of a conjunctive expression used as **Unlabeled_assertion_clause** in a **Precondition**: the subsequent components of the **Attribute_or_routine**.
- 8 If ot is a term of a conjunctive expression used as **Unlabeled_assertion_clause** in a **Check**: the subsequent components of its enclosing **Compound**.

Informative text

The definition ensures that, for an **Object_test** $\{x: T\}$ exp , we can rest assured that, throughout its scope, x will never at run time have a void value, and hence can be used as the target of a call.

End

8.24.6 Semantics: Object Test semantics

The value of an **Object_test** $\{x: T\}$ exp is true if the value of exp is attached to an instance of T , false otherwise.

Informative text

In particular, if x is void (which is possible only if T is a detachable type), the result will be false.

End

8.24.7 Semantics: Object-Test Local semantics

For an **Object_test** $\{x: T\}$ exp , the value of x , defined only over its **scope**, is the value of exp at the time of the **Object_test**'s evaluation.

8.24.8 Definition: Read-only void test

A **read-only void test** is a **Boolean_expression** of one of the forms $e = \text{Void}$ and $e \neq \text{Void}$, where e is a **read-only entity**.

8.24.9 Definition: Scope of a read-only void test

The **scope** of a **read-only void test** appearing in a class text, for e of type T , is the **scope** that the **Object-Test Local** ot would have if the void test were replaced by:

- 1 For $e = \text{Void}$: **not** $(\{ot: T\} e)$.

2 For $e \neq \text{Void}$: {ot: T } e .

Informative text

This is useful if T is a detachable type, providing a simple way to generalize the notion of scope to common schemes such as `if $e \neq \text{Void}$ then ...`, where we know that e cannot be void in the `Then_part`. Note that it is essential to limit ourselves to read-only entities; for a variable, or an expression involving a variable, anything could happen to the value during the execution of the scope even if e is initially not void.

Of course one could always write an `Object_test` instead, but the void test is a common and convenient form, if only because it doesn't require repeating the type T of e , so it will be important to handle it as part of the Certified Attachment Patterns discussed next.

End

8.24.10 Definition: Certified Attachment Pattern

A **Certified Attachment Pattern** (or **CAP**) for an expression exp whose type is detachable is an occurrence of exp in one of the following contexts:

- 1 exp is an Object-Test Local and the occurrence is in its scope.
- 2 exp is a read-only entity and the occurrence is in the scope of a void test involving exp .

Informative text

A CAP is a scheme that has been proved, or certified by sufficiently many competent people (or computerized proof tools), to ensure that exp will never have a void run-time value in the covered scope.

- The CAPs listed here are the most frequently useful and seem beyond doubt. Here too compilers could be "smart" and find other cases making $exp.f$ safe. The language specification explicitly refrains, however, from accepting such supposed compiler improvements: other than the risk of mistake in the absence of a public discussion, this would result in some Eiffel texts being accepted by certain compilers and rejected by others. Instead, a compiler that *accepts* a call to a detachable target that is not part of one of the official CAPs listed above is **non-conformant**.
- The list of CAPs may grow in the future, as more analysis is applied to actual systems, leading to the identification, and certification by human or automatic means, of safe patterns for using targets of detachable types.

End

8.24.11 Definition: Attached expression

An expression exp of type T is **attached** if it satisfies any of the following conditions:

- 1 T is attached.
- 2 T is expanded.
- 3 exp appears in a Certified Attachment Pattern for exp .

Informative text

This is the principal result of this discussion: the condition under which an expression is *target-valid*, that is to say, can be used as target of a call because its value is guaranteed never to be void at any time of evaluation. It is in an Expanded type's nature to abhor a void; attached types are devised to avoid void too; and Certified Attachment Patterns catch a detachable variable when it is provably not detached.

End

8.25 Typing-related properties

Informative text

This Part does not define any new rules, only a few definitions that facilitate discussion of type issues.

End

8.25.1 Definition: Catcall

A **catcall** is a run-time attempt to execute a **Call**, such that the feature of the call is not applicable to the target of the call.

Informative text

The role of the type system is to ensure that a valid system can never, during its execution, produce a catcall.

“Cat” is an abbreviation for “Changed Availability or Type”, two language mechanisms that, if not properly controlled by the type system, could cause catcalls.

End

8.25.2 Validity: Descendant Argument rule

Validity code: *VUDA*

Consider a call of target type *ST* and feature *fname* appearing in a class *C*. Let *sf* be the feature of final name *fname* in *ST*. Let *DT* be a type conforming to *ST*, and *df* the version of *sf* in *DT*. The call is **descendant-argument-valid** for *DT* if and only if it satisfies the following conditions:

- 1 The call is **argument-valid**.
- 2 Every actual argument conforms, after conversion to the corresponding formal argument of *sf* if applicable, to the corresponding formal argument of *df*.

8.25.3 Validity: Single-level Call rule

Validity code: *VUSC*

A call of target *x* is **system-valid** if for any element *D* of the dynamic class set of *x* it is **export-valid** for *D* and **descendant-argument-valid** for *D*.

Informative text

The common goal of the type mechanisms and rules of the language is to ensure that every call is both class-valid and system-valid.

End

8.26 Exception handling

Informative text

During the execution of an Eiffel system, various abnormal events may occur. A hardware or operating system component may be unable to do its job; an arithmetic operation may result in overflow; an improperly written software element may produce an unacceptable outcome.

Such events will usually trigger a signal, or **exception**, which interrupts the normal flow of execution. If the system's text does not include any provision for the exception, execution will terminate. The system may, however, be programmed so as to *handle* exceptions, which means that it will respond by executing specified actions and, if possible, resuming execution after correcting the cause of the exception.

End

8.26.1 Definition: Failure, exception, trigger

Under certain circumstances, the execution or evaluation of a construct specimen may be unable to proceed as defined by the construct's semantics. It is then said to result in a **failure**.

If, during the execution of a feature, the execution of one of its components fails, this prevents continuing its execution normally; such an event is said to **trigger** an **exception**.

Informative text

Examples of exception causes include:

- Assertion violation (in an assertion monitoring mode).
- Failure of a called routine.
- Impossible operation, such as a **Creation** instruction attempted when not enough memory is available, or an arithmetic operation which would cause an overflow or underflow in the platform's number system.
- Interruption signal sent by the machine for example after a user has hit the "break" key or the window of the current process has been resized.
- An exception explicitly raised by the software itself.

Common exception types that do *not* arise in Eiffel, other than through mistakes in the definition of the language as specified by the Standard, are "void calls" (attempts to execute a feature on a void target) and "catcalls" (attempt to execute a feature on an unsuitable object).

End

8.26.2 Syntax: Rescue clauses

Rescue \triangleq **rescue** Compound

Retry \triangleq **retry**

8.26.3 Validity: Rescue clause rule

Validity code: **VXRC**

It is valid for an **Attribute_or_routine** to include a **Rescue** clause if and only if its **Feature_body** is an **Attribute** or an **Effective_routine** of the **Internal** form.

Informative text

An **Internal** body is one which begins with either **do** or **once**. The other possibilities are **Deferred**, for which it would be improper to define an exception handler since the body does not specify an algorithm, and an **External** body, where the algorithm is specified outside of the Eiffel system, which then lacks the information it would need to handle exceptions.

End

8.26.4 Validity: Retry rule

Validity code: **VXRT**

A **Retry** instruction is valid if and only if it appears in a **Rescue** clause.

Informative text

Because this constraint requires the **Retry** physically to appear within the **Rescue** clause, it is not possible for a **Rescue** to call a procedure containing a **Retry**. In particular, a redefined version of **default_rescue** (see next) may not contain a **Retry**.

End

8.26.5 Definition: Exception-correct

A routine is **exception-correct** if any branch of the **Rescue** clause not terminating with a **Retry** ensures the **invariant**.

8.26.6 Semantics: Default Rescue Original Semantics

Class **ANY** introduces a non-frozen procedure **default_rescue** with no argument and a null effect.

Informative text

As the following semantic rules indicate, an exception not handled by an explicit **Rescue** clause will cause a call to *default_rescue*. Any class can redefine this procedure to implement a default exception handling policy for routines of the class that do not have their own **Rescue** clauses.

End

8.26.7 Definition: Rescue block

Any **Internal** or **Attribute** feature *f* of a class *C* has a **rescue block**, a **Compound** defined as follows, where *rc* is *C*'s **version** of *ANY*'s *default_rescue*:

- 1 If *f* has a **Rescue** clause: the **Compound** contained in that clause.
- 2 If *r* is not *rc* and has no **Rescue** clause: a **Compound** made of a single instruction: an **Unqualified_call** to *rc*.
- 3 If *r* is *rc* and has no **Rescue** clause: an empty **Compound**.

Informative text

The semantic rules rely on this definition to define the effect of an exception as if every routine had a **Rescue** clause: either one written explicitly, or an implicit one calling *default_rescue*. To this effect they refer not to **rescue** clauses but to rescue blocks.

Condition 3 avoids endless recursion in the case of *default_rescue* itself.

End

8.26.8 Semantics: Exception Semantics

An **exception triggered** during an execution of a feature *f* causes, if it is neither **ignored** nor **continued**, the effect of the following sequence of events.

- 1 Attach the value of *last_exception* from *ANY* to a direct instance of a descendant of the Kernel Library class **EXCEPTION** corresponding to the type of the exception.
- 2 Unlike in the **non-exception semantics** of **Compound**, do not execute the remaining instructions of *f*.
- 3 If the recipient of the exception is *f*, execute the **rescue block** of *f*.
- 4 If case 3 applies and the rescue block executes a **Retry**, this terminates the processing of the exception. Execution continues with a new execution of the **Compound** in the **Feature_body** of *f*.
- 5 If neither case 3 nor case 4 applies (in particular in case 3 if the rescue block executes to the end without executing a **Retry**), this terminates the processing of the current exception and the current execution of *f*, causing a **failure** of that execution. If the execution of *f* was caused by a call to *f* from another feature, trigger an exception of type **ROUTINE_FAILURE** in the calling routine, to be handled (recursively) according to the present rule. If there is no such calling feature, *f* is the **root procedure**; terminate its execution as having failed.

Informative text

As usual in rules specifying the “effect” of an event in terms of a sequence of steps, all that counts is that effect; it is not required that the execution carry out these exact steps, or carry them in this exact order.

In step 1, the **Retry** will only re-execute the **Feature_body** of *r*, with all entities set to their current value; it does **not** repeat argument passing and local variable initialization. This may be used to ensure that the execution takes a different path on a new attempt.

In most cases, the “recipient” of the exception (case 3) is the current routine, *f*. For exception occurring in special places, such as when evaluating an assertion, the next rule, Exception Cases, tells us whether *f* or its caller is the “recipient”.

In the case of a **Feature_body** of the **Once** form, the above semantics only applies to the first call to every applicable target, where a **Retry** may execute the body two or more times. If that first call fails, triggering a routine failure exception, the applicable rule for subsequent calls is not the above Exception Semantics (since the routine will not execute again) but the **Once Routine Execution Semantics**, which specifies that any such calls must trigger the exception again.

End

8.26.9 Definition: Type of an exception

The **type** of a **triggered exception** is the **generating type** of the object to which the value of **last_exception** is attached per step 1 of the Expression Semantics rule.

8.26.10 Semantics: Exception Cases

The **triggering** of an **exception** in a feature **f** called by a feature **caller** results in the setting of the following properties, accessible through features of the exception class instance to which the value of **last_exception** is attached, as per the following table, where:

- The **Recipient** is either **f** or **caller**.
- “**Type**” indicates the type of the exception (a descendant of **EXCEPTION**).
- If **f** is the **root procedure**, executed during the original system creation call, the value of **caller** as given below does not apply.

	Recipient	Type
Exception during evaluation of invariant on entry	<i>caller</i>	[Type of exception as triggered]
Invariant violation on entry	<i>caller</i>	INVARIANT_ENTRY_VIOLATION
Exception during evaluation of precondition	<i>caller</i>	[Type of exception as triggered]
Exception during evaluation of Old expression on entry	See <u>Old Expression Semantics</u>	
Precondition violation	<i>caller</i>	PRECONDITION_VIOLATION
Exception in body	<i>f</i>	[Type of exception as triggered]
Exception during evaluation of invariant on exit	<i>f</i>	[Type of exception as triggered]
Invariant violation on exit	<i>f</i>	INVARIANT_EXIT_VIOLATION
Exception during evaluation of postcondition on exit	<i>f</i>	[Type of exception as triggered]
Postcondition violation	<i>f</i>	POSTCONDITION_VIOLATION

Informative text

This rule specifies the precise effect of an exception occurring anywhere during execution (including some rather extreme cases, such as the occurrence of an exception in the evaluation of an assertion). Whether the “recipient” is **f** or **caller** determines whether the execution of the current routine can be “retried”: per case 3 of the Exception Semantics rule, a **Retry** is applicable only if the recipient is itself. Otherwise a **ROUTINE_FAILURE** will be triggered in the **caller**.

In the case of an **Old** expression, a special rule, given earlier, requires the exception to be remembered, during evaluation of the expression on entry to the routine, for re-triggering during evaluation of the postcondition on exit, but only if the expression turns out to be needed then.

End

8.26.11 Semantics: Exception Properties

The value of the query **original** of class **EXCEPTION**, applicable to **last_exception**, is an **EXCEPTION** reference determined as follows after the **triggering** of an **exception** of type **TEX**:

- 1 If *TEX* does not conform to *ROUTINE_FAILURE*: a reference to the current *EXCEPTION* object.
- 2 If *TEX* conforms to *ROUTINE_FAILURE*: the previous value of *original*.

Informative text

The reason for this query is that when a routine fails, because execution of a routine *f* has triggered an exception and has not been able to handle it through a *Retry*, the consequence, per case 5 of the Exception Semantics rule, is to trigger a new exception of type *ROUTINE_FAILURE*, to which *last_exception* now becomes attached. Without a provision for *original*, the “real” source of the exception would be lost, as *ROUTINE_FAILURE* exceptions get passed up the call chain. Querying *original* makes it possible, for any other routine up that chain, to find out the *Ur*-exception that truly started the full process.

End

8.26.12 Definition: Ignoring, continuing an exception

It is possible, through routines of the Kernel Library class *EXCEPTION*, to ensure that exceptions of certain types be:

- **Ignored**: lead to no change of non-exception semantics.
- **Continued**: lead to execution of a programmer-specified routine, then to continuation of the execution according to non-exception semantics.

Informative text

The details of what types of exceptions can be ignored and continued, and how to achieve these effects, belong to the specification of class *EXCEPTION* and its descendants.

End

8.27 Agents, iteration and introspection

Informative text

Objects represent information equipped with operations. These are clearly defined concepts; no one would mistake an operation for an object.

For some applications — graphics, numerical computation, iteration, writing contracts, building development environments, “*reflection*” (*a system’s ability to explore its own properties*) — you may find the operations so interesting that you will want to define objects to represent them, and pass these objects around to software elements, which can use these objects to execute the operations whenever they want. Because this separates the place of an operation’s definition from the place of its execution, the definition can be incomplete, since you can provide any missing details at the time of any particular execution.

You can create **agent** objects to describe such partially or completely specified computations. Agents combine the power of higher-level functionals — operations acting on other operations — with the safety of Eiffel’s static typing system.

End

8.27.1 Definition: Operands of a call

The **operands** of a call include its target (explicit in a qualified call, implicit in an unqualified call), and its arguments if any.

8.27.2 Definition: Operand position

The target of a call has **position** 0. The *i*-th actual argument, for any applicable *i*, has **position** *i*.

8.27.3 Definition: Construction time, call time

The **construction time** of an agent object is the time of evaluation of the agent expression defining it.

Its **call time** is when a call to its associated operation is executed.

8.27.4 Syntactical forms for a call agent

A call agent is of the form

agent *agent_body*

where *agent_body* is a **Call**, **qualified** (as in *x.r(...)*) or **unqualified** (as in *f(...)*) with the following possible variants:

- You may replace any argument by a question mark **?**, making the argument open.
- You may replace the target, by **{TYPE}** where **TYPE** is the name of a type, making the target open.
- You may remove the argument list **(...)** altogether, making all arguments open.

Informative text

This is not a formal syntax definition, but a summary of the available forms permitted by the syntax and validity rules that follow.

End

8.27.5 Syntax: Agents

Agent \triangleq **Call_agent** | **Inline_agent**

Call_agent \triangleq **agent** **Call_agent_body**

Inline_agent \triangleq **agent** [**Formal_arguments**] [**Type_mark**] [**Attribute_or_routine**] [**Agent_actu**als]

8.27.6 Syntax: Call agent bodies

Call_agent_body \triangleq **Agent_qualified** | **Agent_unqualified**

Agent_qualified \triangleq **Agent_target** ". " **Agent_unqualified**

Agent_unqualified \triangleq **Feature_name** [**Agent_actu**als]

Agent_target \triangleq **Entity** | **Parenthesized** | **Manifest_type**

Agent_actuals \triangleq "(" **Agent_actu**al_list ")"

Agent_actual_list \triangleq {**Agent_actu**al "; " ...}+

Agent_actual \triangleq **Expression** | **Placeholder**

Placeholder \triangleq [**Manifest_type**] "?"

8.27.7 Definition: Target type of an call agent

The **target type** of a **Call_agent** is:

- 1 If there is no **Agent_target**, the **current type**.
- 2 If there is an **Agent_target** and it is an **Entity** or **Parenthesized**, its type.
- 3 If there is an **Agent_target** and it is a **Manifest_type**, the type that it lists (in braces).

8.27.8 Validity: Call Agent rule

Validity code: *VPCA*

A **Call_agent** involving a **Feature_name** *fn*, appearing in a class *C*, with target type *T0*, is valid if and only if it satisfies the following conditions:

- 1 *fn* is the name of a feature *f* of *T0*.
- 2 If there is an **Agent_target**, *f* is export-valid for *T0* in *C*.
- 3 If the **Agent_actu**als part is present, the number of elements in its **Agent_actu**al_list is equal to the number of formals of *f*.
- 4 Any **Agent_actu**al of the **Expression** kind is of a type compatible with the type of the corresponding formal in *f*.

8.27.9 Definition: Associated feature of an inline agent

Every inline agent *ia* of a class *C* has an **associated feature**, defined as a fictitious routine *f* of *C*, such that:

- 1 The name of *f* is chosen not to conflict with any other feature name in *C* and its descendants.
- 2 The formal arguments of *f* are those of *ia*.
- 3 *f* is secret (available for call to no class).
- 4 The Attribute_or_routine part of *f* is defined by the Attribute_or_routine part of *ia*.
- 5 *f* is a function if *ia* has a Type_mark (its return type being given by the Type in that Type_mark), a procedure otherwise.

8.27.10 Validity: Inline Agent rule

Validity code: *VPIA*

An Inline_agent *a* of associated feature *f*, is valid in the text of a class *C* if and only if it satisfies the following conditions:

- 1 *f*, if added to *C*, would be valid.
- 2 *f* is not deferred.

8.27.11 Validity: Inline Agent Requirements

Validity code: *VPIR*

An Inline_agent *a* must satisfy the following conditions:

- 1 No formal argument or local variable of *a* has the same name as a feature of the enclosing class.
- 2 Every entity appearing in the Routine part of *a* is the name of one of: a formal argument of *a*; a local variable of *a*; a feature of the enclosing class; **Current**.
- 3 The Feature_body of *a*'s Routine is not of the Deferred form.

Informative text

These conditions are stated as another validity rule permitting compilers to issue more understandable error messages. It is not in the usual "if and only if" form (since the preceding rule, the more official one, takes care of this), but the requirements given cover the most obvious possible errors.

End

8.27.12 Definition: Call-agent equivalent of an inline agent

The **call-agent equivalent** of an inline agent *ia* is the Call_agent agent *f*

where *f* is the associated feature of *ia*.

8.27.13 Semantics: Semantics of inline agents

The semantic properties of an inline agent are those of its call-agent equivalent.

8.27.14 Semantics: Use of **Result** in an inline function agent

In an agent of the Inline_agent form denoting a function, the local variable **Result** denotes the result of the agent itself.

8.27.15 Definition: Open and closed operands

The **open operands** of a Call_agent include:

- 1 Any Agent_actual that is a Placeholder.
- 2 The Agent_target if it is present and is a Manifest_type.

The **closed operands** include all non-open operands.

8.27.16 Definition: Open and closed operand positions

The **open operand positions** of an Agent are the operand positions of its open operands, and the **closed operand positions** those of its closed operands.

8.27.17 Definition: Type of an agent expression

Consider a `Call_agent` a , with a `target` of type TO . Let $i1, \dots, im$ ($m \geq 0$) be its `open operand positions`, if any, and let T_{i1}, \dots, T_{im} be the types of f 's formal arguments at positions $i1, \dots, im$ (taking T_{i1} to be TO if $i1 = 0$).

The type of a is:

- `PROCEDURE` [TO , `TUPLE` [T_{i1}, \dots, T_{im}]] if f is a `procedure`.
- `FUNCTION` [TO , `TUPLE` [T_{i1}, \dots, T_{im}], R] if f is a `function` of result type R other than `BOOLEAN`.
- `PREDICATE` [TO , `TUPLE` [T_{i1}, \dots, T_{im}]] if f is a function of result type `BOOLEAN`.

8.27.18 Semantics: Agent Expression semantics

The value of an agent expression a at a certain `construction time` yields a reference to an instance DO of the type of a , containing information identifying:

- The `associated feature` of a .
- Its `open operand positions`.
- The values of its `closed operands` at the time of evaluation.

8.27.19 Semantics: Effect of executing call on an agent

Let DO be an agent object with associated feature f and open positions $i1, \dots, im$ ($m \geq 0$). The information in DO enables a call to the procedure `call`, executed at any `call time` posterior to DO 's construction time, with target DO and (if required) actual arguments a_{i1}, \dots, a_{im} , to perform the following:

- Produce the same effect as a call to f , using the `closed operands` at the `closed operand positions` and a_{i1}, \dots, a_{im} , evaluated at call time, at the `open operand positions`.
- In addition, if f is a `function`, setting the value of the query `last_result` for DO to the result returned by such a call.

8.28 Expressions

Informative text

Through the various forms of `Expression`, software texts can include denotations of run-time values — objects and references.

Previous discussions have already introduced some of the available variants of the construct: `Formal`, `Local`, `Call`, `Old`, `Manifest_tuple`, `Agent`. The present one gives the full list of permissible expressions and the precise form of all but one of the remaining categories: operator expressions, equality and locals. The last category, constants, has its own separate presentation, just after this one.

End

8.28.1 Syntax: Expressions

`Expression` \triangleq `Basic_expression` | `Special_expression`

`Basic_expression` \triangleq `Read_only` | `Local` | `Call` | `Precursor` | `Equality` | `Parenthesized` | `Old` | `Operator_expression` | `Bracket_expression` | `Creation_expression`

`Special_expression` \triangleq `Manifest_constant` | `Manifest_tuple` | `Agent` | `Object_test` | `Once_string` | `Address`

`Parenthesized` \triangleq "(" `Expression` ")"

`Address` \triangleq "\$" `Variable`

`Once_string` \triangleq `once` `Manifest_string`

`Boolean_expression` \triangleq `Basic_expression` | `Boolean_constant` | `Object_test`

8.28.2 Definition: Subexpression, operand

The `subexpressions` of an expression e are e itself and (recursively) all the following expressions:

- 1 For a `Parenthesized` (a) or a `Parenthesized_target` ($[a]$): the subexpressions of a .

- 2 For an **Equality** or **Binary_expression** $a \S b$, where \S is an operator: the subexpressions of a and of b .
- 3 For a **Unary_expression** $\diamond a$, where \diamond is an operator: the subexpressions of a .
- 4 For a **Call**: the subexpressions of the **Actuals** part, if any, of its **Unqualified_part**.
- 5 For a **Precursor**: the subexpressions of its unfolded form.
- 6 For an **Agent**: the subexpression of its **Agent_actuals** if any.
- 7 For a **qualified** call: the subexpressions of its **target**.
- 8 For a **Bracket_expression** $f[a_1, \dots a_n]$: the subexpressions of f and those of all of $a_1, \dots a_n$.
- 9 For an **Old** expression **old** a : a .
- 10 For a **Manifest_tuple** $[a_1, \dots a_n]$: the subexpressions of all of $a_1, \dots a_n$.

In cases 2 and 3, the **operands** of e are a and (in case 2) b .

8.28.3 Semantics: Parenthesized Expression Semantics

If e is an expression, the value of the **Parenthesized** (e) is the value of e .

8.28.4 Syntax: Operator expressions

Operator_expression \triangleq **Unary_expression** | **Binary_expression**

Unary_expression \triangleq **Unary Expression**

Binary_expression \triangleq **Expression Binary Expression**

8.28.5 Operator precedence levels

- 13 **.** (Dot notation, in **qualified** and non-object calls)
- 12 **old** (In postconditions)
not + - **Used as unary**
All free unary operators
- 11 All free binary operators.
- 10 **^** (Used as binary: power)
- 9 *** / // ** (As binary: multiplicative arithmetic operators)
- 8 **+ -** **Used as binary**
- 7 **..** (To define an interval)
- 6 **= /< ~ /~ < > <= >=** (As binary: relational operators)
- 5 **and and then**
(Conjunctive boolean operators)
- 4 **or or else xor**
(Disjunctive boolean operators)
- 3 **implies** (Implicative boolean operator)
- 2 **[]** (Manifest tuple delimiter)
- 1 **;** (Optional semicolon between an **Assertion_clause** and the next)

Informative text

This precedence table includes the operators that may appear in an **Operator_expression**, the equality and inequality symbols used in **Equality** expressions, as well as other symbols and keywords which also occur in expressions and hence require disambiguating: the semicolon in its role as separator for **Assertion_clause**; the **old** operator which may appear in an **Old** expression as part of a Postcondition; the dot **.** of dot notation, which binds tighter than any other operator.

The operators listed include both standard operators and predefined operators ($=$, $/=$, \sim , $/-$). For a free operator, you cannot set the precedence: all free unaries appear at one level, and all free binaries at another level.

End

8.28.6 Definition: Parenthesized Form of an expression

The **parenthesized form** of an expression is the result of rewriting every subexpression of one of the forms below, where \S and \ddagger are different binary operators, \diamond and \clubsuit different unary operators, and a , b , c arbitrary operands, as follows:

- 1 For $a \S b \S c$ where \S is not the power operator \wedge : $(a \S b) \S c$ (left associativity).
- 2 For $a \wedge b \wedge c$: $a \wedge (b \wedge c)$ (right associativity).
- 3 For $a \S b \ddagger c$: $(a \S b) \ddagger c$ if the precedence of \ddagger is lower than the precedence of \S or the same, and $a \S (b \ddagger c)$ otherwise.
- 4 For $\diamond \clubsuit a$: $\diamond (\clubsuit a)$
- 5 For $\diamond a \S b$: $(\diamond a) \S b$
- 6 For $a \S \diamond b$: $a \S (\diamond b)$
- 7 For a subexpression e to which none of the previous patterns applies: e unchanged.

8.28.7 Definition: Target-converted form of a binary expression

The **target-converted form** of a Binary_expression $x \S y$, where the one-argument feature of alias \S in the base class of x has the Feature_name f , is:

- 1 If the declaration of f includes a **convert** mark and the type TY of y is not compatible with the type of the formal argument of f : $((TY) [x]) \S y$.
- 2 Otherwise: the original expression, $x \S y$.

Informative text

$((TY) [x])$ denotes x converted to type TY . This definition allows us, if the feature from x 's type TX cannot accept a TY argument but has explicitly been specified, through the **convert** mark, to allow for target conversion, and TY does include the appropriate feature accepting a TX argument, to use that feature instead.

The archetypal example is *your_integer + your_real* which, with the appropriate **convert** mark in the "+" feature in *INTEGER*, we can interpret as $((REAL) [your_integer]) + your_real$, where "+" represents the *plus* feature from *REAL*.

End

8.28.8 Validity: Operator Expression rule

Validity code: *VWEO*

A Unary_expression $\S x$ or Binary_expression $x \S y$, for some operator \S , is valid if and only if it satisfies the following conditions:

- 1 A feature of the base class of x is declared as **alias** " \S ".
- 2 The expression's Equivalent Dot Form is a valid **Call**.

8.28.9 Semantics: Expression Semantics (strict case)

The **value** of an Expression, other than a Binary_expression whose Binary is semistrict, is the value of its Equivalent Dot Form.

Informative text

This semantic rule and the preceding validity constraint make it possible to forego any specific semantics for operator expressions (except in one special case) and define the value of any expression through other semantic rules of the language, in particular the rules for calls and entities.

This applies in particular to arithmetic and relational operators (for which the feature declarations are in basic classes such as *INTEGER* and *REAL*) and to boolean operators (class *BOOLEAN*): in principle, although not necessary as implemented by compilers, *a + b* is just a feature call like any other.

The excluded case — covered by a separate rule — is that of a binary expression using one of the three **semistrict** operators: **and then**, **or else**, **implies**. This is because the value of an expression such as *a and then b* is not entirely defined by its Equivalent Dot Form *a.conjoined_semistrict (b)*, which needs to evaluate *b*, whereas the **and then** form explicitly ignores *b* when *a* has value *False*, as the value of the whole expression is *False* even if *b* does not have a defined value, a case which should not be treated as an error.

End

8.28.10 Definition: Semistrict operators

A **semistrict operator** is any one of the three operators **and then**, **or else** and **implies**, applied to operands of type *BOOLEAN*.

8.28.11 Semantics: Operator Expression Semantics (semistrict cases)

For *a* and *b* of type *BOOLEAN*:

- The value of *a and then b* is: if *a* has value false, then false; otherwise the value of *b*.
- The value of *a or else b* is: if *a* has value true, then true; otherwise the value of *b*.
- The value of *a implies b* is: if *a* has value false, then true; otherwise the value of *b*.

Informative text

The semantics of other kinds of expression, and Eiffel constructs in general, is **compositional**: the value of an expression with subexpressions *a* and *b*, for example *a + b* (where *a* and *b* may themselves be complex expressions), is defined in terms of the values of *a* and *b*, obtained from the same set of semantic rules, and of the connecting operators, here *+*. Among expressions, those involving semistrict operators are the only exception to this general style. The above rule is not strictly compositional since it tells us that in certain cases of evaluating an expression involving *b* we should not consider the value of *b*. It's not just that we *may* ignore the value of *b* in some cases — which would also be true of *a and b* (strict) when *a* is false — but that we *must* ignore it lest it prevents us from evaluating the expression as a whole.

It's this lack of full compositionality that makes the above rule more **operational** than the semantic specification of other kinds of expression. Their usual form is "*the value of an expression of the form X is Y*", where *Y* only refers to values of subexpressions of *X*. Such rules normally don't mention order of execution. They respect compositionality and leave compilers free to choose any operand evaluation order, in particular for performance. Here, however, order matters: the final requirement of the rule *requires* that the computation first evaluate *a*. We need this operational style to reflect the special nature of nonstrict operators, letting us sometimes get a value for an expression whose second operand does not have any.

End

8.28.12 Syntax: Bracket expressions

Bracket_expression \triangleq *Bracket_target* "[" Actuals "]"

Bracket_target \triangleq *Target* | *Once_string* | *Manifest_constant* | *Manifest_tuple*

Informative text

Target covers every kind of expression that can be used as target of a call, including simple variants like *Local* variables and formal arguments, as well as *Call*, representing the application of a query to a target that may itself be the result of applying calls.

End

8.28.13 Validity: Bracket Expression rule

Validity code: **VWBR**

A **Bracket_expression** $x [j]$ is valid if and only if it satisfies the following conditions:

- 1 A feature of the **base class** of x is declared as **alias** "[j]".
- 2 The expression's Equivalent Dot Form is a valid **Call**.

8.28.14 Definition: Equivalent Dot Form of an expression

Any **Expression** e has an **Equivalent Dot Form**, not involving (in any of its **subexpressions**) any **Bracket_expression** or **Operator_expression**, and defined as follows, where C denotes the **base class** of x , pe denotes the **Parenthesized Form** of e , and x', y', c' denote the Equivalent Dot Forms (obtained recursively) of x, y, c :

- 1 If pe is a **Unary_expression** § $x: x'.f$, where f is the **Feature_name** of the no-argument feature of alias § in C .
- 2 If pe is a **Binary_expression** of **target-converted form** $x § y: x'.f(y')$ where f is the **Feature_name** of the one-argument feature of alias § in C .
- 3 If pe is a **Bracket_expression** $x [y]: x'.f(y')$ where f is the **Feature_name** of the feature declared as **alias** "[j]" in C .
- 4 If pe has no **subexpression** other than itself: pe .
- 5 In all other cases: (recursively) the result of replacing every **subexpression** of e by its Equivalent Dot Form.

8.28.15 Validity: Boolean Expression rule

Validity code: **VWBE**

A **Basic_expression** is valid as a **Boolean_expression** if and only if it is of type **BOOLEAN**.

8.28.16 Validity: Identifier rule

Validity code: **VWID**

An **Identifier** appearing in an expression in a class C , other than as the **feature** of a qualified **Call**, must be the **name** of a feature of C , or a **local variable** of the enclosing feature or inline agent if any, or a formal argument of the enclosing feature or inline agent if any, or the **Object-Test Local** of an **Object_test**.

Informative text

The restriction "other than as the feature of a qualified **Call**" excludes an identifier appearing immediately after a dot to denote a feature being called on a target object: in $a + b.c(d)$, the rule applies to a, b (target of a **Call**) and d (actual argument), but not to c (feature of a qualified **Call**). For c the relevant constraint is the **Call** rule, which among other conditions requires c to be a feature of the base class of b 's type.

The Identifier rule is not a full "if and only if" rule; in fact it is conceptually superfluous since it follows from earlier, more complete constraints. Language processing tools may find it convenient as a simple criterion for detecting the most common case of invalid **Identifier** in expression.

End

8.28.17 Definition: Type of an expression

The type of an **Expression** e is:

- 1 For the predefined **Read_only Current**: the **current type**.
- 2 For a routine's **Formal** argument : the type declared for e .
- 3 For an Object-Test local: its declared type.
- 4 For **Result**, appearing in the text of a query f : the result type of f .
- 5 For a **local variable** other than **Result**: the type declared for e .
- 6 For a **Call**: the type of e as determined by the **Expression Call Type** definition with respect to the current type.
- 7 For a **Precursor**: (recursively) the type of its **unfolded form**.
- 8 For an **Equality**: **BOOLEAN**.

- 9 For a **Parenthesized** (*f*): (recursively) the type of *f*.
- 10 For **old** *f*: (recursively) the type of *f*.
- 11 For an **Operator_expression** or **Bracket_expression**: (recursively) the type of the **Equivalent Dot Form** of *e*.
- 12 For a **Manifest_constant**: as given by the definition of the **type of a manifest constant**.
- 13 For a **Manifest_tuple** [*a*₁, ... *a*_{*n*}] (*n* ≥ 0): **TUPLE** [*T*₁, ... *T*_{*n*}] where each *T*_{*i*} is (recursively) the type of *a*_{*i*}.
- 14 For an **Agent**: as given by the definition of the **type of an agent expression**.
- 15 For an **Object_test**: **BOOLEAN**.
- 16 For a **Once_string**: **STRING**.
- 17 For an **Address** *\$v*: **TYPED_POINTER** [*T*] where *T* is (recursively) the type of *v*.
- 18 For a **Creation_expression**: the **Explicit_creation_type**.

Informative text

Case 6, which refers to a definition given in the discussion of calls, also determines case 11, operator and bracket expressions.

End

8.29 Constants

Informative text

Expressions, just studied, include the special case of constants, whose values cannot directly be changed by execution-time actions. This discussion goes through the various kinds. Particular attention will be devoted to the various forms, single- and multi-line, of *string* constant.

Along with constants proper, we will study two notations for “manifest” objects given by the list of their items: manifest tuples and manifest arrays, both using the syntax [*item*₁, ... *item*_{*n*}].

End

8.29.1 Syntax: Constants

Constant \triangleq **Manifest_constant** | **Constant_attribute**

Constant_attribute \triangleq **Feature_name**

8.29.2 Validity: Constant Attribute rule

Validity code: *VWCA*

A **Constant_attribute** appearing in a class *C* is valid if and only if its **Feature_name** is the **final** name of a **constant attribute** of *C*.

8.29.3 Syntax: Manifest constants

Manifest_constant \triangleq [**Manifest_type**] **Manifest_value**

Manifest_type \triangleq "{ Type }"

Manifest_value \triangleq **Boolean_constant** |

Character_constant |

Integer_constant |

Real_constant |

Manifest_string |

Manifest_type

Sign \triangleq "+" | "-"

Integer_constant \triangleq [**Sign**] **Integer**

Character_constant \triangleq "" **Character** ""

Boolean_constant \triangleq *True* | *False*

Real_constant \triangleq [**Sign**] **Real**

8.29.4 Syntax (non-production): Sign Syntax rule

If present, the **Sign** of an **Integer_constant** or **Real_constant** must immediately precede the associated **Integer** or **Real**, with no intervening **tokens** or **components** (such as **breaks** or **comments**).

8.29.5 Syntax (non-production): Character Syntax rule

The quotes of a **Character_constant** must immediately precede and follow the **Character**, with no intervening **tokens** or **components** (such as **breaks** or **comments**).

Informative text

In general, breaks or comment lines may appear between components prescribed by a BNF-E production, making the last two rules necessary to complement the grammar: for signed constants, you must write `-5`, not `- 5` etc. This helps avoid confusion with operators in arithmetic expressions, which may of course be followed by spaces, as in `a - b`. Similarly, you must write a character constant as `'A'`, not `' A '`.

To avoid any confusion about the syntax of **Character_constant**, it is important to note that a character code such as `%N` (New Line) constitutes a single **Character** token.

End

8.29.6 Definition: Type of a manifest constant

The type of a **Manifest_constant** of **Manifest_value** *mv* is:

- 1 For **{T}** *mv*, with the optional **Manifest_type** present: *T*. The remaining cases assume this optional component is absent, and only involve *mv*.
- 2 If *mv* is a **Boolean_constant**: **BOOLEAN**.
- 3 If *mv* is a **Character_constant**: **CHARACTER**.
- 4 If *mv* is an **Integer_constant**: **INTEGER**.
- 5 If *mv* is a **Real_constant**: **REAL**.
- 6 If *mv* is a **Manifest_string**: **STRING**.
- 7 If *mv* is a **Manifest_type** **{T}**: **TYPE [T]**.

Informative text

As a consequence of cases 3 to 6, the type of a character, string or numeric constant is never one of the sized variants but always the fundamental underlying type (**CHARACTER**, **INTEGER**, **REAL**, **STRING**). Language mechanisms are designed so that you can use such constants without hassle — for example, without explicit conversions — even in connection with specific variants. For example:

- You can assign an integer constant such as 10 to a target of a type such as **INTEGER_8** as long as it fits (as enforced by validity rules).
- You can use such a constant for discrimination in a **Multi_branch** even if the expression being discriminated is of a specific sized variant; here too the compatibility is enforced statically by the validity rules.

Case 7 involves the Kernel Library class **TYPE**.

End

8.29.7 Validity: Manifest-Type Qualifier rule

Validity code: *VWMQ*

It is valid for a **Manifest_constant** to be of the form **{T} v** (with the optional **Manifest_type** qualifier present) if and only if the type *U* of *v* (as determined by cases 2 to 7 of the definition of the **type of a manifest constant**) is one of **CHARACTER**, **STRING**, **INTEGER** and **REAL**, and *T* is one of the **sized variants** of *U*.

Informative text

The rule states no restriction on the value, even though an example such as `{INTEGER_8} 256` is clearly invalid, since 256 is not representable as an `INTEGER_8`. The Manifest Constant rule addresses this.

End

8.29.8 Semantics: Manifest Constant Semantics

The **value** of a `Manifest_constant` `c` listing a `Manifest_value` `v` is:

- 1 If `c` is of the form `{T} v` (with the optional `Manifest_type` qualifier present): the value of type `T` denoted by `v`.
- 2 Otherwise (`c` is just `v`): the value denoted by `v`.

8.29.9 Definition: Manifest value of a constant

The **manifest value** of a constant is:

- 1 If it is a `Manifest_constant`: its `value`.
- 2 If it is a constant attribute: (recursively) the manifest value of the `Manifest_constant` listed in its declaration.

Informative text

As the following syntax indicates, there are two ways to write a manifest string:

- A `Basic_manifest_string`, the most common case, is a sequence of characters in double quotes, as in `"This text"`. Some of the characters may be special character codes, such as `%N` representing a new line. This variant is useful for such frequent applications as object names, texts of simple messages to be displayed, labels of buttons and other user interface elements, generally using fairly short and simple sequences of characters. You may write the string over several lines by ending an interrupted line with a percent character `%` and starting the next one, after possible blanks and tabs, by the same character.
- A `Verbatim_string` is a sequence of lines to be taken exactly as they are (hence the name), bracketed by `{` at the end of the line that precedes the sequence and `}` at the beginning of the line that follows the sequence (or `[` and `]` to left-align the lines). No special character codes apply. This is useful for embedding multi-line texts; applications include `description` entries of `Notes` clauses, inline C code, SQL or XML queries to be passed to some external program.

End

8.29.10 Syntax: Manifest strings

`Manifest_string` \triangleq `Basic_manifest_string` | `Verbatim_string`

`Basic_manifest_string` \triangleq `'"'" String_content '"'"`

`String_content` \triangleq `{Simple_string Line_wrapping_part ...}+`

`Verbatim_string` \triangleq `Verbatim_string_opener` `Line_sequence` `Verbatim_string_closer`

`Verbatim_string_opener` \triangleq `'"'" [Simple_string] Open_bracket`

`Verbatim_string_closer` \triangleq `Close_bracket [Simple_string] '"'"`

`Open_bracket` \triangleq `"[" | "["`

`Close_bracket` \triangleq `"]" | "]"`

Informative text

In the “basic” case, most examples of **String_content** involve just one **Simple_string** (a sequence of printable characters, with no new lines, as defined in the description of lexical components). For generality, however, **String_content** is defined as a repetition, with successive **Simple_string** components separated by **Line_wrapping_part** to allow writing a string on several lines. Details below.

In the “verbatim” case, **Line_sequence** is a lexical construct denoting a sequence of lines with arbitrary text. The reason for the **Verbatim_string_opener** and the **Verbatim_string_closer** is to provide an escape sequence for an extreme case (a **Line_sequence** that begins with **]**), but most of the time the opener is just “[or “{ and the closer **]** or **”**. The difference between brackets and braces is that with “{ ... }” the **Line_sequence** is kept exactly as is, whereas with “[...]” the lines are left-aligned (stripped of any common initial blanks and tabs). Details below.

End

8.29.11 Syntax (non-production): Line sequence

A **specimen** of **Line_sequence** is a sequence of one or more **Simple_string components**, each separated from the next by a single **New_line**.

8.29.12 Syntax (non-production): Manifest String rule

In addition to the properties specified by the grammar, every **Manifest_string** must satisfy the following properties:

- 1 The **Simple_string** components of its **String_content** or **Line_sequence** may not include a double quote character except as part of the character code **%** (denoting a double quote).
- 2 A **Verbatim_string_opener** or **Verbatim_string_closer** may not contain any **break character**.

Informative text

Like other “non-production” syntax rules, the last two rules capture simple syntax requirements not expressible through BNF-E productions.

Because a **Line_sequence** is made of simple strings separated by a single **New_line** in each case, a line in a **Verbatim_string** that looks like a comment is not a comment but a substring of the **Verbatim_string**.

End

8.29.13 Definition: Line_wrapping_part

A **Line_wrapping_part** is a sequence of characters consisting of the following, in order: **%** (percent character); zero or more blanks or tabs; **New_line**; zero or more blanks or tabs; **%** again.

Informative text

This construct requires such a definition since it can’t be specified through a context-free syntax formalism such as BNF-E.

The use of **Line_wrapping_part** as separator between a **Simple_string** and the next in a **Basic_manifest_string** allows you to split a string across lines, with a **%** at the end of an interrupting line and another one at the beginning of the resuming line. The definition allows blanks and tabs before the final **%** of a **Line_wrapping_part** although they will not contribute to the contents of the string. This makes it possible to apply to the **Basic_manifest_string** the same indentation as to the neighboring elements. The definition also permits blanks and tabs after the

initial % of a `Line_wrapping_part`, partly for symmetry and partly because it doesn't seem justified to raise an error just because the compiler has detected such invisible but probably harmless characters.

End

8.29.14 Semantics: Manifest string semantics

The value of a `Basic_manifest_string` is the sequence of characters that it includes, in the order given, excluding any `line wrapping parts`, and with any `character code` replaced by the corresponding character.

8.29.15 Validity: Verbatim String rule

Validity code: *VWVS*

A `Verbatim_string` is valid if and only if it satisfies the following conditions, where α is the (possibly empty) `Simple_string` appearing in its `Verbatim_string_opener`:

- 1 The `Close_bracket` is `]` if the `Open_bracket` is `[`, and `}` if the `Open_bracket` is `{`.
- 2 Every character in α is `printable`, and not a double quote `"`.
- 3 If α is not empty, the string's `Verbatim_string_closer` includes a `Simple_string` identical to α .

8.29.16 Semantics: Verbatim string semantics

The value of a `Line_sequence` is the string obtained by concatenating the characters of its successive lines, with a "new line" character inserted between any adjacent ones.

The value of a `Verbatim_string` using braces `{ }` as `Open_bracket` and `Close_bracket` is the value of its `Line_sequence`.

The value of a `Verbatim_string` using braces `[]` as `Open_bracket` and `Close_bracket` is the value of the `left-aligned form` of its `Line_sequence`.

Informative text

This semantic definition is **platform-independent**: even if an environment has its own way of separating lines (such as two characters, carriage return `%R` and new line `%N`, on Windows) or represents each line as a separate element in a sequence (as in older operating systems still used on mainframes), the semantics yields a single string — a single character sequence — where each successive group of characters, each representing a line of the original, is separated from the next one by a single `%N`.

End

8.29.17 Definition: Prefix, longest break prefix, left-aligned form

A **prefix** of a string s is a string p of some length n ($n \geq 0$) such that the first n characters of s are the corresponding characters of p .

The **longest break prefix** of a sequence of strings ls is the longest string bp containing no characters other than `spaces` and `tabs`, such that bp is a prefix of every string in ls . (The longest break prefix is always defined, although it may be an empty string.)

The **left-aligned form** of a sequence of strings ls is the sequence of strings obtained from the corresponding strings in ls by removing the first n characters, where n is the length of the longest break prefix of ls ($n \geq 0$).

8.30 Basic types

Informative text

The term "basic type" covers a number of expanded class types describing elementary values: booleans, characters, integers, reals, machine-level addresses. The corresponding classes — *BOOLEAN*; *CHARACTER*; *INTEGER*; *REAL* and variants specifying explicit sizes; *POINTER* — are part of ELKS, the Eiffel Library Kernel Standard.

The following presentation explains the general concepts behind the design and use of these classes.

End

8.30.1 Definition: Basic types and their sized variants

A **basic type** is any of the types defined by the following ELKS classes:

- *BOOLEAN*.
- *CHARACTER*, *CHARACTER_8*, *CHARACTER_32*, together called the “**sized variants of CHARACTER**”.
- *INTEGER*, *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *NATURAL*, *NATURAL_8*, *NATURAL_16*, *NATURAL_32*, *NATURAL_64*, together called the “**sized variants of INTEGER**”.
- *REAL*, *REAL_32*, *REAL_64*, together called the “**sized variants of REAL**”.
- *POINTER*.

8.30.2 Definition: Sized variants of *STRING*

The sized variants of *STRING* are *STRING*, *STRING_8* and *STRING_32*.

8.30.3 Semantics: Boolean value semantics

Class *BOOLEAN* covers the two truth values.

The reserved words *True* and *False* denote the corresponding constants.

8.30.4 Semantics: Character types

The reference class *CHARACTER_GENERAL* describes properties of characters independently of the character code.

The expanded class *CHARACTER_32* describes Unicode characters; the expanded class *CHARACTER_8* describes 8-bit (ASCII-like) characters.

The expanded class *CHARACTER* describes characters with a length and encoding settable through a compilation option. The recommended default is Unicode.

8.30.5 Semantics: Integer types

The reference class *INTEGER_GENERAL* describes integers, signed or not, of arbitrary length. The expanded classes *INTEGER_xx*, for *xx* = 8, 16, 32 or 64, describe signed integers stored on *xx* bits. The expanded classes *NATURAL_xx*, for *xx* = 8, 16, 32 or 64, describe unsigned integers stored on *xx* bits.

The expanded classes *INTEGER* and *NATURAL* describe integers, respectively signed and unsigned, with a length settable through a compilation option. The recommended default is 64 bits in both cases.

8.30.6 Semantics: Floating-point types

The reference class *REAL_GENERAL* describes floating-point numbers with arbitrary precision. The expanded classes *REAL_xx*, for *xx* = 32 or 64, describe IEEE floating-point numbers with *xx* bits of precision.

The expanded class *REAL* describes floating-point numbers with a precision settable through a compilation option. The recommended default is 64 bits.

8.30.7 Semantics: Address semantics

The expanded class *POINTER* describes addresses of data beyond the control of Eiffel systems.

8.31 Interfacing with C, C++ and other environments

Informative text

Object technology as realized in Eiffel is about **combining components**. Not all of these components are necessarily written in the same language; in particular, as organizations move to Eiffel, they will want to reuse their existing investment in components from other languages, and make their Eiffel systems interoperate with non-Eiffel software.

Eiffel is a "pure" O-O language, not a hybrid between object principles and earlier approaches such as C, and at the same time an **open** framework for combining software written in various languages. These two properties might appear contradictory, as if consistent use of object technology meant closing oneself off from the rest of the programming world. But it's exactly the reverse: a hybrid approach, trying to be O-O as well as something completely different, cannot succeed at both since the concepts are too distant. Eiffel instead strives, by providing a coherent object framework — with such principles as Uniform Access, Command-Query Separation, Single Choice, Open-Closed and Design by Contract — to be a *component combinator* capable of assembling software bricks of many different kinds.

The following presentation describes how Eiffel systems can integrate components from other languages and environments.

End

8.31.1 Syntax: External routines

`External` \triangleq `external` `External_language` [`External_name`]

`External_language` \triangleq `Unregistered_language` | `Registered_language`

`Unregistered_language` \triangleq `Manifest_string`

`External_name` \triangleq `alias` `Manifest_string`

Informative text

The `External` clause is the mechanism that enables Eiffel to interface with other environments and serve as a "component combinator" for software reuse and particularly for taking advantage of legacy code.

By default the mechanism assumes that the external routine has the same name as the Eiffel routine. If this is not the case, use an `External_name` of the form `alias "ext_name"`. The name appears as a `Manifest_string`, in quotes, not an identifier, because external languages may have different naming conventions; for example an underscore may begin a feature name in C but not in Eiffel, and some languages are case-sensitive for identifiers whereas Eiffel is not.

Instead of calling a pre-existing foreign routine, it is possible to include `inline` C or C++ code; the `alias` clause will host that code, which can access Eiffel objects through the arguments of the external routine.

The language name (`External_language`) can be an `Unregistered_language`: a string in quotes such as `"Cobol"`. Since the content of the string is arbitrary, there is no guarantee that a particular Eiffel environment will support the corresponding language interface. This is the reason for the other variant, `Registered_language`: every Eiffel compiler must support the language names `"C"`, `"C++"` and `dll`. Details of the specific mechanisms for every such `Registered_language` appear below.

Some of the *validity* rules below include a provision, unheard of in other parts of the language specification, allowing Eiffel language processing tools to rely on *non-Eiffel tools* to enforce some conditions. A typical example is a rule that requires an external name to denote a suitable foreign function; often, this can only be ascertained by a compiler for the foreign language. Such rules should be part of the specification, but we can't impose their enforcement on an Eiffel compiler without asking it also to become a compiler of C, C++ etc.; hence this special tolerance.

The general *semantics* of executing external calls appeared as part of the general semantics of calls. The semantic rules of the present discussion address specific cases, in particular inline C and C++.

End

8.31.2 Semantics: Address semantics

The value of an **Address** expression is an address enabling foreign software to access the associated **Variable**.

Informative text

The manipulations that the foreign software can perform on such addresses depend on the foreign programming language. It is the implementation's responsibility to ensure that such manipulations do not violate Eiffel semantic properties.

End

8.31.3 Syntax: Registered languages

Registered_language \triangleq C_external | C++_external | DLL_external

8.31.4 Syntax: External signatures

External_signature \triangleq **signature** [External_argument_types] [: External_type]

External_argument_types \triangleq "(" External_type_list ")"

External_type_list \triangleq {External_type "," ...}*

External_type \triangleq Simple_string

8.31.5 Validity: External Signature rule

Validity code: VZES

An **External_signature** in the declaration of an external **routine** *r* is valid if and only if it satisfies the following conditions:

- 1 Its **External_type_list** contains the same number of elements as *r* has formal arguments.
- 2 The final optional component (: **External_type**) if present if and only if *r* is a **function**.

A **language processing tool** may delegate enforcement of these requirements to non-Eiffel tools on the chosen **platform**.

Informative text

The rule does not prescribe any particular relationship between the argument and result types declared for the Eiffel routine and the names appearing in the **External_type_list** and the final **External_type** if any, since the precise correspondence depends on foreign language properties beyond the scope of Eiffel rules.

The specification of a non-external routine never includes C-style empty parenthesization: for a declaration or call of a routine without arguments you write *r*, not *r* (). The syntax of **External_argument_types**, however, permits () for compatibility with other languages' conventions.

The last part of the rule allows Eiffel tools to rely on non-Eiffel tools if it is not possible, from within Eiffel, to check the properties of external routines. This provision also applies to several of the following rules.

End

8.31.6 Semantics: External signature semantics

An **External_signature** specifies that the associated external routine:

- Expects arguments of number and types as given by the **External_argument_types** if present, and no arguments otherwise.
- Returns a result of the **External_type** appearing after the colon, if present, and otherwise no result.

8.31.7 Syntax: External file use

```

External_file_use  $\triangleq$  use External_file_list
External_file_list  $\triangleq$  {External_file "," ...}+
External_file  $\triangleq$  External_user_file | External_system_file
External_user_file  $\triangleq$  ' "' Simple_string ' "'
External_system_file  $\triangleq$  "<"Simple_string ">"

```

Informative text

As the syntax indicates, you may specify as many external files as you like, preceded by **use** and separated by commas. You may specify two kinds of files:

- “System” files, used only in a C context, appear between angle brackets <> and refer to specific locations in the C library installation.
- The name of a “user” file appears between double quotes, as in `"/path/user/her_include.h"`, and will be passed on literally to the operating system. Do not forget, when using double quotes, that this is all part of an Eiffel **Manifest_string**: you must either code them as %" or, more conveniently, write the string as a **Verbatim_string**, the first line preceded by "[and the last line followed by]".

End

8.31.8 Validity: External File rule

Validity code: *VZEF*

An **External_file** is valid if and only if its **Simple_string** satisfies the following conditions:

- 1 When interpreted as a file name according to the conventions of the underlying **platform**, it denotes a file.
- 2 The file is accessible for reading.
- 3 The file's content satisfies the rules of the applicable foreign language.

A **language_processing_tool** may delegate enforcement of these conditions to non-Eiffel tools on the chosen **platform**.

Informative text

Condition 3 means for example that if you pass an include file to a C function the content must be C code suitable for inclusion by a C “include” directive. Such a requirement may be beyond the competence of an Eiffel compiler, hence the final qualification enabling Eiffel tools to rely, for example, on compilation errors produced by a C compiler.

The “conventions of the underlying platforms” cited in condition 1 govern the rules on file names (in particular the interpretation of path delimiters such as / and \ on Unix and Windows) and, for an **External_system_file** name of the form `<some_file.h>`, the places in the file system where `some_file.h` is to be found.

End

8.31.9 Semantics: External file semantics

An **External_file_use** in an external **routine** declaration specifies that foreign language tools, to process the routine (for example to compile its original code), require access to the listed files.

8.31.10 Syntax: C externals

```

C_external  $\triangleq$  ' "' C
           '[inline]
           [External_signature] [External_file_use]
           ' "'

```

Informative text

The `C_external` mechanism makes it possible, from Eiffel, to use the mechanisms of C. The syntax covers two basic schemes:

- You may rely on an existing C function. You will not, in this case, use `inline`. If the C function's name is different from the lower name of the Eiffel routine, specify it in the `alias (External_name)` clause; otherwise you may just omit that clause.
- You may also write C code *within* the Eiffel routine, putting that code in the `alias` clause and specifying `inline`.

In the second case the C code can directly manipulate the routine's formal arguments and, through them, Eiffel objects. The primary application (rather than writing complex processing in C code in an Eiffel class, which would make little sense) is to provide access to existing C libraries without having to write and maintain any new C files even if some "glue code" is necessary, for example to perform type adaptations. Such code, which should remain short and simple, will be directly included and maintained in the Eiffel classes providing the interface to the legacy code.

The `alias` part is a `Manifest_string` of one of the two available forms:

- It may begin and end with a double quote `"`; then any double quote character appearing in it must be preceded by a percent sign, as `%"`; line separations are marked by the special code for "new line", `%N`.
- If the text extends over more than one line, it is more convenient to use a `Verbatim_string`: a sequence of lines to be taken exactly as they are, preceded by `"[` at the end of a line and followed by `]"` at the beginning of a line.

In this `Manifest_string`, you may refer to any formal argument `a` of the external routine through the notation `$a` (a dollar sign immediately followed by the name of the argument). For `a` you may use either upper or lower case, lower being the recommended style as usual.

End

8.31.11 Validity: C external rule

Validity code: *VZCC*

A `C_external` for the declaration of an external `routine r` is valid if and only if it satisfies the following conditions:

- 1 At least one of the optional `inline` and `External_signature` components is present.
- 2 If the `inline` part is present, the external routine includes an `External_name component`, of the form `alias C_text`.
- 3 If case 2 applies, then for any occurrence in `C_text` of an `Identifier a` immediately preceded by a dollar sign `$` the lower name of `a` is the lower name of a formal argument of `r`.

8.31.12 Semantics: C Inline semantics

In an external `routine er` of the `inline` form, an `External_name` of the form `alias C_text` denotes the algorithm defined, according to the semantics of the C language, by a C function that has:

- As its signature, the `signature` specified by `er`.
- As its body, `C_text` after replacement of every occurrence of `$a`, where the `lower name` of `a` is the lower name of one of the formal arguments of `er`, by `a`.

8.31.13 Syntax: C++ externals

```
C++_external ≙ ' "' C++
  inline
  [External_signature]
  [External_file_use]
  ' "'
```

Informative text

As in the C case, you may directly write C++ code which can access the external routine's argument and hence Eiffel objects. Such code can, among other operations, create and delete C++ objects using C++ constructors and destructors.

Unlike in the C case, this inline facility is the *only* possibility: you cannot rely on an existing function. The reason is that C++ functions — if not “static” — require a target object, like Eiffel routines. By directly writing appropriate inline C++ code, you will take care of providing the target object whenever required.

End

8.31.14 Validity: C++ external rule

Validity code: VZC+

A C++_external part for the declaration of an external routine *r* is valid if and only if it satisfies the following conditions:

- 1 The external routine includes an External_name component, of the form *alias C++_text*.
- 2 For any occurrence in C++_text of an Identifier *a* immediately preceded by a dollar sign \$, the lower name of *a* is the lower name of a formal argument of *r*.

8.31.15 Semantics: C++ Inline semantics

In an external routine *er* of the C++_external form, an External_name of the form *alias C++_text* denotes the algorithm defined, according to the semantics of the C++ language, by a C++ function that has:

- As its signature, the signature specified by *er*.
- As its body, C++_text after replacement of every occurrence of \$*a*, where the lower name of *a* is the lower name of one of the formal arguments of *er*, by *a*.

8.31.16 Syntax: DLL externals

```
DLL_external ≙ ' " dll
  [windows]
  DLL_identifier
  [DLL_index]
  [External_signature]
  [External_file_use]
  ' "
```

DLL_identifier ≙ Simple_string

DLL_index ≙ Integer

Informative text

Through a DLL_external you may define an Eiffel routine whose execution calls an external mechanism from a Dynamic Link Library, not loaded until first use.

The mechanism assumes a dynamic loading facility, such as exist on modern platforms; it is specified to work with any such platform.

End

8.31.17 Validity: External DLL rule

Validity code: VZDL

A DLL_external of DLL_identifier *i* is valid if and only if it satisfies the following conditions:

- 1 When interpreted as a file name according to the conventions of the underlying platform, *i* denotes a file.
- 2 The file is accessible for reading.
- 3 The file's content denotes a dynamically loadable module.

8.31.18 Semantics: External DLL semantics

The routine to be executed (after loading if necessary) in a call to a `DLL_external` is the dynamically loadable routine from the file specified by the `DLL_identifier` and, within that file, by its name and the `DLL_index` if present.

8.32 Lexical components

Informative text

The previous discussions have covered the syntax, validity and semantics of software systems. At the most basic level, the texts of these systems are made of **lexical** components, playing for Eiffel classes the role that words and punctuation play for the sentences of human language. All construct descriptions relied on lexical components — identifiers, reserved words, special symbols ... — but their structure has not been formally defined yet. It is time now to cover this aspect of the language, affecting its most elementary components.

End

8.32.1 Syntax (non-production): Character, character set

An Eiffel text is a sequence of **characters**. Characters are either:

- All 32-bit, corresponding to Unicode and to the Eiffel type `CHARACTER_32`.
- All 8-bit, corresponding to 8-bit extended ASCII and to the Eiffel type `CHARACTER_8`.

Compilers and other language processing tools must offer an option to select one **character set** from these two. The same or another option determines whether the type `CHARACTER` is equivalent to `CHARACTER_32` or `CHARACTER_8`.

Informative text

In manifest strings and character constants, characters can be coded either directly, as a single-key entry, or through a multiple-key character code such as `%N` (denoting new-line) or `%/59/`. The details appear below.

End

8.32.2 Definition: Letter, alpha_betic, numeric, alpha_numeric, printable

A **letter** is any character belonging to one of the following categories:

- 1 Any of the following fifty-two, each a lower-case or upper-case element of the Roman alphabet:
`abcdefghijklmnopqrst u v w x y z`
`ABCDEFGHIJKLMN OPQRSTUVWXYZ`
- 2 If the underlying character set is 8-bit extended ASCII, the characters of codes 192 to 255 in that set.
- 3 If the underlying character set is Unicode, all characters defined as letters in that set.

An **alpha_betic character** is a letter or an underscore `_`.

A **numeric character** is one of the ten characters `0 1 2 3 4 5 6 7 8 9`.

An **alpha_numeric character** is alpha_betic or numeric.

A **printable character** is any of the characters listed as printable in the definition of the character set (Unicode or extended ASCII).

Informative text

In common English usage, “alphabetic” and “alphanumeric” characters do not include the underscore. The spellings “alpha_betic” and “alpha_numeric” are a reminder that we accept underscores in both identifiers, as in `your_variable`, and numeric constants, as in `8_961_226`.

“Printable” characters exclude such special characters as new line and backspace.

Case 2 of the definition of “letter” refers to the 8-bit extended ASCII character set. Only the 7-bit ASCII character set is universally defined; the 8-bit extension has variants corresponding to alphabets used in various countries. Codes 192 to 255 generally cover letters equipped with *diacritical marks* (accents, umlauts, cedilla). As a result, if you use an 8-bit letter not in the 7-bit character set, for example to define an identifier with a diacritical mark, it may — without any effect on its Eiffel semantics — display differently depending on the “*locale*” settings of your computer.

End

8.32.3 Definition: Break character, break

A **break character** is one of the following characters:

- Blank (also known as space).
- Tab.
- New Line (also known as Line Feed).
- Return (also known as Carriage Return).

A **break** is a sequence of one or more break characters that is not part of a **Character_constant**, of a **Manifest_string** or of a **Simple_string component** of a **Comment**.

8.32.4 Semantics: Break semantics

Breaks serve a purely **syntactical** role, to separate **tokens**. The effect of a break is independent of its makeup (its precise use of spaces, tabs and newlines). In particular, the separation of a class text into lines has no effect on its semantics.

Informative text

Because the above definition of “break” excludes break characters appearing in **Character_constant**, **Manifest_string** and **Comment** components, the semantics of these constructs may take such break characters into account.

End

8.32.5 Definition: Expected, free comment

A comment is **expected** if it appears in a **construct** as part of the style guidelines for that construct. Otherwise it is **free**.

8.32.6 Syntax (non-production): “Blanks or tabs”, new line

A **specimen** of **Blanks_or_tabs** is any non-empty sequence of **characters**, each of which is a blank or a tab.

A specimen of **New_line** is a New Line.

8.32.7 Syntax: Comments

Comment \triangleq “-” {**Simple_string** **Comment_break** ...}*
Comment_break \triangleq **New_line** [**Blanks_or_tabs**] “-”

Informative text

This syntax implies that two or more successive comment lines, with nothing other than new lines to separate them, form a single comment.

End

8.32.8 Syntax (non-production): Free Comment rule

It is permitted to include a **free comment** between any two successive **components** of a **specimen** of a **construct** defined by a BNF-E **production**, except if excluded by specific syntax rules.

Informative text

An example of construct whose specimens may not include comments is Line_sequence, defined not by a BNF-E production but by another “non-production” syntax rule: no comments may appear between the successive lines of such a sequence — or, as a consequence, of a Verbatim_string. Similarly, the Alias Syntax rule excludes any characters — and hence comments — between an Alias_name and its enclosing quotes.

End

8.32.9 Header comment rule

A feature Header_comment is an abbreviation for a Note clause of the form

note

what: Explanation

where Explanation is a Verbatim_string with [and] as Open_bracket and Close_bracket and a Line_sequence made up of the successive lines (Simple_string) of the comment, each deprived of its first characters up to and including the first two consecutive dash characters, and of the space immediately following them if any.

Informative text

Per the syntax, a comment is a succession of Simple_string components, each prefixed by "--" itself optionally preceded, in the second and subsequent lines if any, by a Blank_or_tabs. To make up the Verbatim_string we remove the Blank_or_tabs and dashes; we also remove one immediately following space, to account for the common practice of separating the dashes from the actual comment text, as in

-- A comment.

End

8.32.10 Definition: Symbol, word

A **symbol** is either a special symbol of the language, such as the semicolon “;” and the “.” of dot notation, or a standard operator such as “+” and “*”.

A **word** is any token that is not a symbol. Examples of words include identifiers, keywords, free operators and non-symbol operators such as **or else**.

8.32.11 Syntax (non-production): Break rule

It is permitted to write two adjacent tokens without an intervening break if and only if they satisfy one of the following conditions:

- 1 One is a word and the other is a symbol.
- 2 They are both symbols, and their concatenation is not a symbol.

Informative text

Without this rule, adjacent words not separated by a break — as in *ifxthen* — or adjacent symbols would be ambiguous.

End

8.32.12 Semantics: Letter Case rule

Letter case is significant for the following constructs: Character_constant and Manifest_string except for special character codes, Comment.

For all other constructs, letter case is not significant: changing a letter to its lower-case or upper-case counterpart does not affect the semantics of a specimen of the construct.

8.32.13 Definition: Reserved word, keyword

The following names are **reserved words** of the language.

agent	alias	all	and	as	assign	attribute
check	class	convert	create	Current	debug	deferred
do	else	elseif	end	ensure	expanded	export
external	False	feature	from	frozen	if	implies
inherit	inspect	invariant	like	local	loop	not
note	obsolete	old	once	only	or	Precursor
redefine	rename	require	rescue	Result	retry	select
separate	then	True	TUPLE	undefine	until	variant
Void	when	xor				

The reserved words that serve as purely syntactical markers, not carrying a direct semantic value, are called **keywords**; they appear in the above list in all lower-case letters.

Informative text

The non-keyword reserved words, such as **True**, have a semantics of their own (**True** denotes one of the two boolean values).

The Letter Case rule applies to reserved words, so the decision to write keywords in all lower case is simply a style guideline. Non-keyword reserved words are most closely related to constants and, like constants, have — in the recommended style — a single upper-case letter, the first; **TUPLE** is most closely related to types and is all upper-case.

End

8.32.14 Syntax (non-production): Double Reserved Word rule

The reserved words **and then** and **or else** are each made of two components separated by one or more blanks (but no other break characters). Every other reserved word is a sequence of letters with no intervening break character.

8.32.15 Definition: Special symbol

A **special symbol** is any of the following character sequences:

```

— : ; , ? ! ' " $ . - > :=
= /= ~ /~ ( ) ( | ) [ ] { }

```

8.32.16 Syntax (non-production): Identifier

An **Identifier** is a sequence of one or more alpha_numeric characters of which the first is a letter.

8.32.17 Validity: Identifier rule

Validity code: VIID

An **Identifier** is valid if and only if it is not one of the language's reserved words.

8.32.18 Definition: Predefined operator

A **predefined operator** is one of:

```

= /= ~ /~

```

Informative text

These operators — all “special symbols” — appear in **Equality** expressions. Their semantics, reference or object equality or inequality, is defined by the language (although you can adapt the effect of **~** and **/~** since they follow redefinitions of **is_equal**). As a consequence you may not use them as **Alias** for your own features.

End

8.32.19 Definition: Standard operator

A **standard unary operator** is one of:

```

+ -

```

A **standard binary operator** is any one of the following one- or two-character symbols:

+ - * / ^ < >
<= >= // \\
..

Informative text

All the standard operators appear as **Operator** aliases for numeric and relational features of the Kernel Library, for example *less_than alias* "<" in *INTEGER* and many other classes. You may also use them as **Alias** in your own classes.

End

8.32.20 Definition: Operator symbol

An **operator symbol** is any non-alpha numeric printable character that satisfies any of the following properties:

- 1 It does not appear in any of the special symbols.
- 2 It appears in any of the standard (unary or binary) operators but is neither a dot . nor an equal sign =.
- 3 It is a tilde ~, percent %, question mark ?, or exclamation mark !.

Informative text

Condition 1 avoids ambiguities with special symbols such as quotes. Conditions 2 and 3 override it when needed: we do for example accept as operator symbols +, a standard operator, and \ which appears in a standard operator — but not a dot or an equal sign, which have a set meaning.

End

8.32.21 Definition: Free operator

A **free operator** is sequence of one or more characters satisfying the following properties:

- 1 It is not a special symbol, standard operator or predefined operator.
- 2 Every character in the sequence is an operator symbol.
- 3 Every subsequence that is not a standard operator or predefined operator is distinct from all special symbols.

A **Free_unary** is a free operator that is distinct from all standard unary operators.

A **Free_binary** is a free operator that is distinct from all standard binary operators.

Informative text

Condition 3 gives us maximum flexibility without ambiguity; for example:

- You may **not** use `---` as an operator because, its subsequence `--` clashes with the special symbol introducing comments.
- You may similarly **not** use `---` because the full sequence (which of course is a subsequence too) could still be interpreted as making the rest of the line a comment.
- You **may**, however, use a single `-`, or define a free operator such as `-*` which does not cause any such confusion.
- You may **not** use `?`, `!`, `=` or `~`, but you **may** use operators containing these characters, for example `! =`.
- You **may** use a percent character `%` by itself or in connection with other operator symbols. No confusion is possible with character codes such as `%B` and `%/123/`. (If you use a percent character in an **Alias** specification, its occurrences in the **Alias_name** string must be written as `%%` according to the normal rules for special characters in strings. For example you may define a feature *remainder alias* "%%%" to indicate that it has `%` as an **Operator** alias. But any use of the operator outside of such a string is written just `%`, for example in the expression `a % b` which in this case would be a shorthand for `a.remainder(b)`.)

Alpha_numeric characters are not permitted. For example, you may not use `+b` as an operator: otherwise `a+b` could be understood as consisting of one identifier and one operator.

End

8.32.22 Syntax (non-production): Manifest character

A **manifest character** — specimen of construct **Character** — is one of the following:

- 1 Any key associated with a printable character, except for the percent key `%`.
- 2 The sequence `%k`, where `k` is a one-key code taken from the list of special characters.
- 3 The sequence `%/code/`, where `code` is an unsigned integer in any of the available forms — decimal, binary, octal, hexadecimal — corresponding to a valid character code in the chosen character set.

Informative text

Form 1 accepts any character on your keyboard, provided it matches the character set you have selected (Unicode or extended ASCII), with the exception of the percent, used as special marker for the other two cases.

Form 2 lets you use predefined percent codes, such as `%B` for backspace, for the most commonly needed special characters. The set of supported codes follows.

Form 3 allows you to denote any Unicode or Extended ASCII character by its integer code; for example `%/59/` represents a semicolon (the character of code 59). Since listings for character codes — for example in Unicode documentation — often give them in base 16, you may use the `0xNNN` convention for hexadecimal integers: the semicolon example can also be expressed as `%/0x3B/`, where `3B` is the hexadecimal code for 59.

Since the three cases define all the possibilities, a percent sign is illegal in a context expecting a **Character** unless immediately followed by one of the keys of the following table or by `/code/` where `code` is a legal character code. For example `%?` is illegal (no such special character); so is `%0x/FFFFFF/` (not in the Unicode range).

End

8.32.23 Special characters and their codes

Character	Code	Mnemonic name
@	<code>%A</code>	A t-sign
BS	<code>%B</code>	B ackspace
^	<code>%C</code>	C ircumflex
\$	<code>%D</code>	D ollar
FF	<code>%F</code>	F orm feed
\	<code>%H</code>	B ackslas H
~	<code>%L</code>	Ti l d e
NL (LF)	<code>%N</code>	N ewline
`	<code>%Q</code>	B ack Q uote
CR	<code>%R</code>	C arriage R eturn
#	<code>%S</code>	S harp
HT	<code>%T</code>	H orizontal T ab
NUL	<code>%U</code>	N U L
	<code>%V</code>	V ertical bar
%	<code>%%</code>	P ercent
'	<code>%'</code>	S ingle quote
"	<code>%"</code>	D ouble quote
[<code>%[</code>	O pening bracket

]	%)	Closing bracket
{	%<	Opening brace
}	%>	Closing brace

Informative text

A few of these codes, such as the last four, are present on many keyboards, but sometimes preempted to represent letters with diacritical marks; using %< rather than [guarantees that you always get a bracket.

End

8.32.24 Syntax (non-production): Percent variants

The percent forms of **Character** are available for the manifest characters of a **Character_constant** and of the **Simple_string** components of a **Manifest_string**, but not for any other token.

Informative text

The characters "of" such a constant do not include the single ' or double " quotes, which you must enter as themselves.

End

8.32.25 Semantics: Manifest character semantics

The value of a **Character** is:

- 1 If it is a printable character *c* other than %: *c*.
- 2 If it is of the form %*k* for a one-key code *k*: the corresponding character as given by the table of special characters.
- 3 If it is of the form %/*code*/: the character of code *code* in the chosen character set.

8.32.26 Syntax (non-production): String, simple string

A **string** — specimen of construct **String** — is a sequence of zero or more manifest characters.

A **simple string** — specimen of **Simple_string** — is a **String** consisting of at most one line (that is to say, containing no embedded new-line manifest character).

8.32.27 Semantics: String semantics

The value of a **String** or **Simple_string** is the sequence of the values of its characters.

8.32.28 Syntax: Integers

Integer \triangleq [**Integer_base**] **Digit_sequence**

Integer_base \triangleq "0" **Integer_base_letter**

Integer_base_letter \triangleq "b" | "c" | "x" | "B" | "C" | "X"

Digit_sequence \triangleq **Digit**⁺

Digit \triangleq "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" |
 "a" | "b" | "c" | "d" | "e" | "f" |
 "A" | "B" | "C" | "D" | "E" | "F" | "_"

Informative text

To introduce an integer base, use the digit **0** (zero) followed by a letter denoting the base: **b** for binary, **c** for octal, **x** for hexadecimal. Per the Letter Case rule the upper-case versions of these letters are permitted, although lower-case is the recommended style.

Similarly, you may write the hexadecimal digits of the last two lines in lower or upper case. Here upper case is the recommended style, as in **0xA5**.

End

8.32.29 Validity: Integer rule

Validity code: *VIIIN*

An **Integer** is valid if and only if it satisfies the following conditions:

- 1 It contains no **breaks**.
- 2 Neither the first nor the last **Digit** of the **Digit_sequence** is an underscore “_”.
- 3 If there is no **Integer_base** (decimal integer), every **Digit** is either one of the decimal digits **0** to **9** (zero to nine) or an underscore.
- 4 If there is an **Integer_base** of the form **0b** or **0B** (binary integer), every **Digit** is either **0**, **1** or an underscore.
- 5 If there is an **Integer_base** of the form **0c** or **0C** (octal integer), every **Digit** is either one of the octal digits **0** to **7** or an underscore.

Informative text

The rule has no requirement for the hexadecimal case, which accepts all the digits permitted by the syntax.

Integer is a purely lexical construct and does not include provision for a sign; the construct **Integer_constant** denotes possibly signed integers.

End

8.32.30 Semantics: Integer semantics

The value of an **Integer** is the integer constant denoted in ordinary mathematical notation by the **Digit_sequence**, without its underscores if any, in the corresponding base: binary if the **Integer** starts with **0b** or **0B**, octal if it starts with **0c** or **0C**, hexadecimal if it starts with **0x** or **0X**, decimal otherwise.

Informative text

This definition always yields a well-defined mathematical value, regardless of the number of digits. It is only at the level of **Integer_constant** that the value may be flagged as invalid, for example `{NATURAL_8} 256`, or `999 ... 999` with too many digits to be representable as either an `INTEGER_32` or an `INTEGER_64`.

The semantics ignores any underscores, which only serve to separate groups of digits for clarity. With decimal digits, the recommended style, if you include underscores, is to use groups of three from the right.

End

8.32.31 Syntax (non-production): Real number

A **real** — specimen of **Real** — is made of the following elements, in the order given:

- An optional decimal **Integer**, giving the integral part.
- A required “.” (dot).
- An optional decimal **Integer**, giving the fractional part.
- An optional exponent, which is the letter **e** or **E** followed by an optional **Sign** (+ or –) and a decimal **Integer**.

No intervening character (**blank** or otherwise) is permitted between these elements. The integral and fractional parts may not both be absent.

Informative text

As with integers, you may use underscores to group the digits for readability. The recommended style uses groups of three in both the integral and decimal parts, as in `45_093_373.567_21`. If you include an exponent, **E**, rather than **e**, is the recommended form.

End

8.32.32 Semantics: Real semantics

The value of a **Real** is the real number that would be expressed in ordinary mathematical notation as $i.f10^e$, where i is the integral part, f the fractional part and e the exponent (or, in each case, zero if the corresponding part is absent).

Draft 5.10, 25 August 2006 (Santa Barbara). Extracted from ongoing work on future third edition of "Eiffel: The Language". Copyright Bertrand Meyer 1986-2005. Access restricted to purchasers of the first or second printing (Prentice Hall, 1991). Do not reproduce or distribute.

PART VIII: BACK MATTER

This part of the book contains the index, plus chapters intended for logistical reasons and not for inclusion in the final text.

Index

Symbols

+, -, <, >, [], .. and other operation symbols, see under **alias**

-> 357

A

abs

INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,
INTEGER_GENERAL 985
REAL, REAL_GENERAL 992

abstract data type 6

actual generic parameter 15

Actual_generics 350

Actual_list 626

Actuals 626

adapt

ROUTINE 1011

adapted

TYPE 976

Address 761

Address rule 834

Agent 751

Agent_actual 752

Agent_actual_list 752

Agent_actuals 752

Agent_qualified 752

Agent_target 752

Agent_unqualified 752

Alias 151

alias 9, 20

alias "-"

INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,

INTEGER_GENERAL 985
NUMERIC 980, 985
REAL, REAL_GENERAL 992

alias ":@"

INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,
INTEGER_GENERAL 985
NUMERIC 980
REAL, REAL_GENERAL 992

alias "+"

INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,
INTEGER_GENERAL 985
NUMERIC 980
POINTER 995
REAL, REAL_GENERAL 992

alias ".."

CHARACTER 983
COMPARABLE 978
INTEGER_GENERAL 984
PART_COMPARABLE 977
REAL_GENERAL 991
STRING 998
TYPE 976

alias "/"

INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,
INTEGER_GENERAL 985
NUMERIC 980
REAL, REAL_GENERAL 992

alias "//"

INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,
INTEGER_GENERAL 985

alias "<"

CHARACTER 983
COMPARABLE 978
INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,

INTEGER_GENERAL 984
INTERVAL 981
PART_COMPARABLE 977
REAL, REAL_GENERAL 991
STRING 998

alias "<="

CHARACTER 983
COMPARABLE 978
INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,
INTEGER_GENERAL 984
PART_COMPARABLE 977
REAL, REAL_GENERAL 991
STRING 999
TYPE 976

alias ">"

CHARACTER 983
COMPARABLE 978
INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,
INTEGER_GENERAL 984
INTERVAL 981
PART_COMPARABLE 977
REAL, REAL_GENERAL 991
STRING 999

alias ">="

CHARACTER 983
COMPARABLE 978
INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,
INTEGER_GENERAL 984
PART_COMPARABLE 977
REAL, REAL_GENERAL 991
STRING 999

alias "[]"

TYPE 976

alias "\"

INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,

- INTEGER_GENERAL 985
- alias** "**^**"
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 - NUMERIC 980
 - REAL, REAL_GENERAL 992
- alias** "**and then**"
 - BOOLEAN 982
- alias** "**and**"
 - BOOLEAN 982
- alias** "**implies**"
 - BOOLEAN 982
- alias** "**not**"
 - BOOLEAN 982
- alias** "**or else**"
 - BOOLEAN 982
- alias** "**or**"
 - BOOLEAN 982
- alias** "**xor**"
 - BOOLEAN 982
- Alias Validity rule 163
- alias** "**+**"
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 - REAL, REAL_GENERAL 992
- Alias_name 151
- all** 17
- America, Pierre 1058
- ancestors of a class 16
- Anchor 328
- Anchored 328
- Anchored Type rule 345
- Anderson, Bruce 1058
- Anderson, John 1058, 1059
- ANONYMOUS** 997
- ANY** 975
- append_boolean*
 - STRING 999
- append_character*
 - STRING 999
- append_integer*
 - STRING 999
- append_real*
 - STRING 999
- append_string*
 - STRING 999
- apply*
 - FUNCTION 1013
 - PREDICATE 1014
- ROUTINE 1011, 1012
 - argument*
 - ARGUMENTS 1008
 - Argument rule 634
 - argument_count*
 - ARGUMENTS 1008
 - ARGUMENTS 1008
 - ARNAUD, FRANCK 1057, 1059
 - Arnout, Karine 1056
 - ARRAY 996
 - Arslan, Volkan 1056
 - as** 17
 - ASCII 880
 - 7-bit 880
 - extended 880
 - Assertion 232
 - assertion 11–14
 - weakening and strengthening in descendants 19
 - Assertion_clause 232
 - assertion_violation*
 - EXCEPTIONS 1007
 - Assigner Call rule 610
 - Assigner Command rule 156
 - Assigner_call 609
 - Assigner_mark 155
 - Assignment 589
 - assignment
 - attempt, see assignment attempt
 - Assignment rule 590
 - attached object 7
 - Attribute 501
 - attribute 8
 - Attribute_or_routine 143
- B**
- basic types 10
- Basic_expression 761
- Basic_manifest_string 795
- basic_store*
 - STORABLE 1005
- Berry, Dave 1058
- Bezault, Éric 1056, 1059, 1060
- Bielak, Richard 1056, 1059
- Binary 154
- Binary_expression 766
- Bishop, Judith 1057
- bit_and*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
- bit_not*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 986
- bit_one*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
- bit_or*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
- bit_shift*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
- bit_shift_left*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
- bit_shift_right*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
- bit_size*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
- bit_xor*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 986
- BON (Business Object Notation) 136
- BOOLEAN** 982
- Boolean_bits*
 - PLATFORM 1009
- Boolean_constant 788
- Boolean_expression 761
- bounds*
 - ARRAY 996
- Boussard, Jean-Claude 1057, 1058
- Bouy, Reynald 1056
- Bracket 151
- Bracket Expression rule 780
- Bracket_expression 778
- Browne, Roger 1059, 1060
- Butler, David 1058

C

- C
 - interface with Eiffel 842-??
- C external rule 846
- C Inline rule 848
- C++_external 848
- C_external 843
- Call 626
- call
 - feature of a call 629
 - target 628
 - target type 629
- call*
 - FUNCTION 1013
 - PREDICATE 1014
 - ROUTINE 1011, 1012
- Call Agent rule 754
- Call rule, see General Call rule, Single-level Call rule, Export Validity rule
- Call Sharing rule 458
- Call Use rule 623
- Call_agent 751
- Call_agent_body 752
- callable*
 - FUNCTION 1013
 - PREDICATE 1014
 - ROUTINE 1011, 1012
- Callegarin, Giuseppe 1057
- ceiling*
 - REAL, REAL_GENERAL 992
- change_name*
 - FILE 1003
- CHARACTER 983
- Character_bits*
 - PLATFORM 1009
- Character_constant 788
- Check_instruction*
 - EXCEPTIONS 1007
- Choice 485
- Choices 485
- class 6, 7–10
 - ancestors 16
 - clients 7
 - descendants 16
 - heirs 16
 - needs another 23
 - parents 16
- class** 119, 124
- class** 8
 - Class ANY rule 173
 - Class Header rule 126, 310, 311
 - class invariant 11
 - Class Name rule 111
 - Class Type rule 333
 - Class_declaration 119
 - Class_header 124
 - Class_invariant*
 - EXCEPTIONS 1007
 - Class_list 208
 - Class_name 110
 - class_name*
 - TYPE 976
 - Class_or_tuple_type 328
 - Class_type 328, 350
 - Class-Level Call rule 636
 - client 7
 - Clients 208
 - Clone* (language primitive in Eiffel 1 and 2, now discarded) 1083
 - close*
 - FILE 1003
 - Close_bracket 795
 - cluster 23
 - code*
 - CHARACTER 983
 - Cohen, Paul 1056
 - collecting*
 - MEMORY 1006
 - collection_off*
 - MEMORY 1006
 - collection_on*
 - MEMORY 1006
 - Colnet, Dominique 1056, 1057
 - command_name*
 - ARGUMENTS 1008
 - COMPARABLE 978
 - Comparison 567
 - compositional 777
 - Compound 96, 228, 478
 - Conditional 89, 481
 - conformance rules
 - VNCC (general) 388
 - VNCE (expanded types) 396
 - VNCF (formal generic) 393
 - VNCN (non-generic) 390
 - VNCS (signature) 386
 - VNCT (tuple types) 397
 - conforms_to*
 - TYPE 976
 - conjoined*
 - BOOLEAN 982
 - conjoined_semistrict*
 - BOOLEAN 982
 - Constant 787
 - Constant Attribute rule 787
 - Constant_attribute 513, 787
 - Constant_interval 485
 - constrained, see genericity
 - Constraining_types 357
 - Constraint 357
 - constraint
 - constraints on language constructs, see under “validity constraint”
 - individual constraints, see under “validity constraints”, and also under each constraint’s name
 - generic constraint, see under “generic constraint”
 - Constraint_creators 357
 - Constraint_list 357
 - Construct 89
 - constructor, synonym for “creation procedure” 526
 - contract 13, 14
 - Conversion Procedure rule 411
 - Conversion Query rule 413
 - Conversion_procedure 410
 - Conversion_query 410
 - Converter 410
 - Converter_list 410
 - Converters 410
 - convertibility rules
 - VYEC (expression) 424
 - VYPF (precondition-free) 426
 - Cook, William 1058
 - copy*
 - ANY 975
 - FUNCTION 1013
 - PREDICATE 1014
 - ROUTINE 1011, 1012
 - STRING 1000
 - count*
 - ANONYMOUS 997
 - ARRAY 996
 - FILE 1002
 - STRING 998

- create_read_write*
 FILE 1002
- creation** 11
- Creation Clause rule 548
- Creation Expression Properties 562
- Creation Expression rule 562
- creation instruction 7
- Creation instruction Properties 555
- Creation Instruction rule 553, 688
- Creation Precondition rule 547
- Creation System-Validity rule 687
- Creation_call* 551
- Creation_clause* 547
- Creation_expression* 561
- Creation_instruction* 551
- Creation_procedure* 547
- Creation_procedure_list* 547
- Creators** 547
- Creel, Christopher 1057
- Crismer, Paul 1056
- Current** (entity denoting the current object) 512, 513, 622, 9
- current object 649, 9
- D**
- Debug** 498
- Declaration_body* 141
- deep_clone*
 ANY 975
- deep_equal*
 ANY 975
- default*
 TYPE 976
- default value for each type 9
- Default_bit_size*
 INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
- default_create*
 INTEGER 987, 988, 989, 990
 INTEGER_GENERAL 984
- default_output*
 STD_FILES 1001
- default_rescue*
 ANY 975
- Deferred** 222
- deferred** 119, 124
- deferred** 20
- deferred class 20–??
- delete*
 FILE 1003
- Deramat, Frédéric 1056
- Dernbach, Frédéric 1055
- Descendant Argument rule 667
- descendants of a class 16
- design 21
- Design by Contract 12– 14, 19
 subcontracting 19
- developer_exception_name*
 EXCEPTIONS 1007
- die*
 EXCEPTIONS 1007
- Digit_sequence* 900
- direct instance 7, 23
- disjuncted*
 BOOLEAN 982
- disjuncted_exclusive*
 BOOLEAN 982
- disjuncted_semistrict*
 BOOLEAN 982
- dispose*
 FILE 1003
 MEMORY 1006
- divided*
 INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 NUMERIC 980
 REAL, REAL_GENERAL 992
- divisible*
 INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 NUMERIC 980
 REAL, REAL_GENERAL 991
- DLL (Dynamically Linked Library)
 856–??, 858–??
 calling a DLL routine determined at run time 858–??
 sharing and freeing ??– 857, 864– 865
- DLL_external* 857
- DLL_identifier* 857
- DLL_index* 857
- dll32** (C interface) 856
- do** 8
- do_nothing*
 ANY 975
- Dubois, Paul 1056, 1058, 1059
- Durchholz, Joachim 1060
- dynamic binding 17– 19
 definition 18
- E**
- ECMA 1056
- effecting a feature 21
- effective 20
- Effective_routine* 222
- efficiency of Eiffel 6
- Eiffel
 organization of Eiffel software 23
- Eiffel language 3–??, 1061– 24
- Eldridge, Geoff 1057
- Elinck, Philippe 1056
- else** 15
- Else_part* 481
- end** 119
- end** 7
- end_of_file*
 FILE 1002
- ensure** 11
- enter*
 ARRAY 996
- Entity** 512
- entity 7
 local, see local variable
- Entity Declaration rule 221
- Entity rule 513
- Entity_declaration_group* 220
- Entity_declaration_list* 220
- Equal* (language primitive in Eiffel 1 and 2, now discarded) 1083, 1084
- Equality** 567
- error*
 STD_FILES 1001
- ETL, abbreviation for the title of the present book (*Eiffel: The Language*)
- exception 14– 15
- exception*
 EXCEPTIONS 1007
 EXCEPTIONS 1007
- exists*
 FILE 1002

- Exit_condition 495
- expanded** 119, 124
- expanded type 10
 - conformance 396
- Explicit_creation_type 551
- exponentiable*
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 - NUMERIC 980
 - REAL, REAL_GENERAL 991
- export** 16
- Export List rule 210
- Export rule 632
- Expression 761
- expression
 - convertibility 424
 - expression convertibility 424
 - Expression rule 781
 - Expression_list 373, 810
 - extended ASCII 880
 - extendibility 5
 - External 829
 - external** ??– 878
 - External DLL rule 857
 - External File rule 841
 - External Signature rule 839
 - external software 823– 878
 - External_argument_types 839
 - External_file_list 840
 - External_file_name 841
 - External_file_use 840
 - External_language 829
 - External_name 829, 837
 - External_signature 839
 - External_system_file 841
 - External_type 839
 - External_type_list 839
 - External_user_file 841
- F**
- failure of a routine 14
- false** 15
- feature 7
 - of a call 629
- feature** 8
- Feature Body rule 144
- Feature Declaration rule 162
- Feature Identifier principle 153
- Feature Name rule 474, 475
- Feature_adaptation 171
- Feature_body 143, 222
- Feature_clause 137
- Feature_declaration 141
- Feature_declaration_list 137
- Feature_list 209
- Feature_name 151
- Feature_set 209
- Feature_value 141
- Features 137
- Fiat, Jocelyn 1056
- FILE** 1002
- fill*
 - STRING 999
- floor*
 - REAL, REAL_GENERAL 992
- force*
 - ARRAY 996
- Ford, Paul 1057
- Forget* (language primitive in Eiffel 1 and 2, now discarded) 1063, 1083
- Formal 513
- Formal Argument rule 220
- formal generic
 - conformance 393
- formal generic parameter 15
- Formal Generic rule 351
- Formal_arguments 220
- Formal_generic 128, 351
- Formal_generic_list 128, 351
- Formal_generic_name 351
- Formal_generics 128, 351
- Franceschi, Fabrice 1056
- fresh*
 - ONCE_MANAGER 1010
- Freund, Pascal 1055
- from_c*
 - STRING 998
- from_integer*
 - INTEGER 987
 - INTEGER_16 989
 - INTEGER_64 990
 - INTEGER_8 988
 - INTEGER_GENERAL 984
- from_string*
 - STRING 998
- full_collect*
 - MEMORY 1006
- function 8
- G**
- Gacsaly, Michael 1059, 1060
- garbage collection 564
- Gautier, Martine 1057
- General Call rule 681
- general conformance 388
- general_store*
 - STORABLE 1005
- generating_type*
 - ANY 975
- generic
 - instantiation, do not use this term for “generic derivation” 352
- Generic Constraint rule 357
- Generic Derivation rule 359
- genericity 15, 19– 20
 - combined with inheritance 19– 20
 - constrained 19
 - unconstrained 15
- Gindre, Cyrille 1058
- Gish, Jim 1058
- Goldsmith, Jacques 1056
- Gore, Jacob 1057
- Granik, Serge 1058
- Gupta, Uday 1059
- H**
- hash_code*
 - ANONYMOUS 997
 - BOOLEAN 982
 - CHARACTER 983
 - FUNCTION 1013
 - HASHABLE 979
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
 - POINTER 995
 - PREDICATE 1014
 - REAL, REAL_GENERAL 991
 - ROUTINE 1011, 1012
 - STRING 998
 - TYPE 976
- HASHABLE** 979
- head*
 - STRING 999
- Header_comment 137

- Header_mark 124
- heir 16
- Henson, Andy 1058
- Hollenberg, David 1056
- Horvilleur, Gerardo 1058
- Howard, Mark 1056, 1059
- Hucklesby, Philip 1056
- I**
- Identifier rule 782, 891
- Identifier_list 220
- identity*
- INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 - NUMERIC 980
 - REAL, REAL_GENERAL 992
- if** 7
- implication*
- BOOLEAN 982
- include** (C interface) 840
- include file for external C routines 840
- Incorrect_inspect_value*
- EXCEPTIONS 1007
- independent_store*
- STORABLE 1005
- index_of*
- STRING 998
- infix operator 9
- inherit** 16
- Inherit_clause 171
- Inheritance 171, 303
- inheritance 16–??
- combined with genericity 19–20
- Initialization 495
- initialization 9
- Inline Agent requirements 756
- Inline Agent rule 755
- Inline_routine 751
- input*
- STD_FILES 1001
- insert*
- STRING 999
- insert_character*
- STRING 999
- instance 6, 7, 8, 9, 10, 11, 21
- current 9
 - see also direct instance
- instantiation
- generic, do not use this term for “generic derivation” 352
- Instruction 228
- INTEGER 987, 988, 989, 990
- Integer 900
- Integer rule 901
- INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
- Integer_base 900
- Integer_base_letter 900
- Integer_bits*
- PLATFORM 1009
- Integer_constant 788
- Internal 222
- INTERVAL 981
- Interval
- rule 487
- Interval rule 487
- Invariant 232
- invariant 11
- invariant** 11
- io*
- ANY 975
- is_closed*
- FILE 1002
- is_comparable*
- COMPARABLE 978
 - INTERVAL 981
 - PART_COMPARABLE 977
- is_developer_exception*
- EXCEPTIONS 1007
- is_empty*
- FILE 1002
 - INTERVAL 981
 - STRING 999
- is_equal*
- ANY 975
 - COMPARABLE 978
 - FUNCTION 1013
 - PART_COMPARABLE 977
 - PREDICATE 1014
 - ROUTINE 1011, 1012
 - STRING 998
 - TYPE 976
- is_greater*
- CHARACTER 983
 - COMPARABLE 978
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
 - PART_COMPARABLE 977
 - REAL, REAL_GENERAL 991
 - STRING 999
- is_greater_equal*
- CHARACTER 983
 - COMPARABLE 978
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
 - REAL, REAL_GENERAL 991
 - STRING 999
- is_less*
- CHARACTER 983
 - COMPARABLE 978
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
 - PART_COMPARABLE 977
 - REAL, REAL_GENERAL 991
 - STRING 998
- is_less_equal*
- CHARACTER 983
 - COMPARABLE 978
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
 - PART_COMPARABLE 977
 - REAL, REAL_GENERAL 991
 - STRING 999
- is_open_read*
- FILE 1002
- is_open_write*
- FILE 1002
- is_plain_text*
- FILE 1002
- is_readable*
- FILE 1002
- is_signal*
- EXCEPTIONS 1007
- is_subinterval*
- INTERVAL 981
- is_superinterval*
- INTERVAL 981
- is_writable*
- FILE 1003
- item*
- ANONYMOUS 997
 - ARRAY 996
 - FUNCTION 1013
 - PREDICATE 1014
 - STRING 998

J

Jézéquel, Jean-Marc 1057
 John, Randy 1056, 1059
 Johnson, Paul 1057
 Join rule 319
 Jones, Rick 1058

K

Kallio, Sami 1056
Key_list 222
 Kogtenkov, Alexander 1056,
 1059
 Kraemer, Vincent 1058

L

Lace 24
 Lahire, Philippe 1056
 Lalanne, Frédéric 1056
 Lancaster, Tal 1058
last_character
 FILE 1003
 STD_FILES 1001
last_integer
 FILE 1003
 STD_FILES 1001
last_real
 FILE 1003
 STD_FILES 1001
last_result
 FUNCTION 1013
 PREDICATE 1014
last_string
 FILE 1003
 STD_FILES 1001
left_adjust
 STRING 999
 Local 513
local 15
 local entity, see local variable
 local variable
 rule 226
 Local Variable rule 226
Local_declarations 225
 Löhr, Peter 1058
 Loop 495
 Loop_body 495
Loop_invariant
 EXCEPTIONS 1007
Loop_variant

EXCEPTIONS 1007

lower

ARRAY 996
 INTERVAL 981
 Ludwig, Stefan 1058

M

Macrakis, Stavros 1058
make
 ARRAY 996
 FILE 1002
 INTEGER_GENERAL 984
 INTERVAL 981
 STRING 998
 Mallet, Olivier 1056
 Manfredi, Raphaël 1055
 Mangseth, Eirik 1060
 Manifest Constant rule 503
Manifest_array 810
Manifest_constant 788, 789
Manifest_string 795
Manifest_tuple 373
Manifest_type 788
Manifest_value 141, 788
 Masini, Gérald 1057

max

CHARACTER 983
 COMPARABLE 978
 INTEGER, INTEGER_8,
 INTEGER_16, INTEGER_32,
 INTEGER_GENERAL 984
 PART_COMPARABLE 977
 REAL, REAL_GENERAL 991
 STRING 999

Maximum_character_code

PLATFORM 1009

Maximum_integer

PLATFORM 1009

McKim, James 1057, 1059

McKim, Jim 1057, 1059

MEMORY 1006

Message 129

Métrás, Pierre 1060

Meyer, Annie 1058

Meyer, Caroline 1058

Meyer, Isabelle 1058

Meyer, Laurent 1058

Meyer, Raphaël 1058

Meyer, Sarah 1058

min

CHARACTER 983

COMPARABLE 978

INTEGER, INTEGER_8,
 INTEGER_16, INTEGER_32,
 INTEGER_GENERAL 984
 PART_COMPARABLE 977
 REAL, REAL_GENERAL 991
 STRING 999

Mingins, Christine 453, 1056,
 1057, 1059

Minimum_character_code

PLATFORM 1009

Minimum_integer

PLATFORM 1009

minus

INTEGER, INTEGER_8,
 INTEGER_16, INTEGER_32,
 INTEGER_GENERAL 985

NUMERIC 980

REAL, REAL_GENERAL 992

Mitchell, Richard 1057

Monninger, Frieder 1057, 1058,
 1059

Multi_branch 485

Multi-Branch

rule 488

Multi-Branch rule 488

Multiple Constraints rule 369

Multiple_constraint 357

N

name

FILE 1002

TYPE 976

needed class 23

needs (relation between classes) 23

negated

BOOLEAN 982

INTEGER, INTEGER_8,
 INTEGER_16, INTEGER_32,
 INTEGER_GENERAL 985

NUMERIC 980

REAL, REAL_GENERAL 992

Nelson, Jim 1057

Nerson, Jean-Marc 1055

New_export_item 209

New_export_list 209

New_exports 209

New_feature 141

New_feature_list 141

No_more_memory

- EXCEPTIONS 1007
 - Non_conformance 171
 - Non_object_call 626
 - NONE 8
 - non-generic
 - conformance 390
 - Non-Object Call rule 631
 - not enough memory available 564
 - Note_entry 123
 - Note_item 123
 - Note_list 123
 - Note_name 123
 - Note_values 123
 - Notes 123
 - NUMERIC 980
- O**
- O'Connor, Sam 1056, 1059, 1060
 - Object Test rule 659
 - Object_call 626
 - Object_test 658
 - object-oriented design 6, 21
 - Obsolete 129
 - occurrences
 - STRING 998
 - Old 239, 242
 - old 11, 12
 - Old Expression rule 239
 - Once 222
 - ONCE_MANAGER 1010
 - Once_string 761
 - onces
 - ANY 975
 - one
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
 - NUMERIC 980
 - REAL, REAL_GENERAL 991
 - Only Clause rule 243
 - open_append
 - FILE 1002
 - Open_bracket 795
 - open_count
 - FUNCTION 1013
 - PREDICATE 1014
 - ROUTINE 1011, 1012
 - open_operand_type
 - FUNCTION 1013
 - PREDICATE 1014
 - ROUTINE 1011, 1012
 - open_read
 - FILE 1002, 1003
 - open_read_append
 - FILE 1003
 - open_read_write
 - FILE 1002, 1003
 - open_write
 - FILE 1002, 1003
 - openness 6
 - operands
 - FUNCTION 1013
 - PREDICATE 1014
 - ROUTINE 1011, 1012
 - operational 777
 - Operator 154
 - Operator Expression rule 772
 - operator, prefix or infix 9
 - Operator_expression 766
 - Osmond, Roger 1056, 1059
 - out
 - ANY 975
 - BOOLEAN 982
 - CHARACTER 983
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 986
 - POINTER 995
 - REAL, REAL_GENERAL 992
 - STRING 1000
 - output
 - STD_FILES 1001
 - overloading (incompatible with object-oriented principles) 153
- P**
- parameter, see formal generic parameter, actual generic parameter (for routines the terminology is "argument")
 - Parent 171
 - parent 16
 - Parent rule 178
 - Parent_list 171
 - Parenthesized 761
 - Parenthesized_target 626
 - Parker, Simon 1057
 - PART_COMPARABLE 977
 - PART_is_greater_equal
 - PART_COMPARABLE 977
 - PART_COMPARABLE 977
 - Piens, Irina 1058
 - Placeholder 752
 - PLATFORM 1009
 - plus
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 - NUMERIC 980
 - POINTER 995
 - REAL, REAL_GENERAL 992
 - POINTER 995
 - Pointer_bits
 - PLATFORM 1009
 - polymorphic data structure 19
 - polymorphism 17– 19
 - portability 6
 - Postcondition
 - EXCEPTIONS 1007
 - Postcondition 232
 - postcondition 11
 - postcondition
 - FUNCTION 1013
 - PREDICATE 1014
 - PROCEDURE 1012
 - ROUTINE 1011
 - Potter, John 1057, 1059
 - power
 - INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 - NUMERIC 980
 - REAL, REAL_GENERAL 992
 - Precondition
 - EXCEPTIONS 1007
 - Precondition 232
 - precondition 11
 - precondition
 - FUNCTION 1013
 - PREDICATE 1014
 - PROCEDURE 1012
 - ROUTINE 1011
 - Precondition Export rule 237
 - precondition-free routine 426
 - Precursor rule 304
 - PREDICATE 1014
 - prefix operator 9
 - principle
 - reattachment 599
 - PROCEDURE 1012
 - procedure 8

- Proch, Karl 1057
- product*
- INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 - NUMERIC 980
 - REAL, REAL_GENERAL 992
- program, see system
- put*
- ANONYMOUS 997
 - ARRAY 996
 - STRING 1000
- put_boolean*
- FILE 1004
 - STD_FILES 1001
- put_character*
- FILE 1004
 - STD_FILES 1001
- put_integer*
- FILE 1004
 - STD_FILES 1001
- put_new_line*
- STD_FILES 1001
- put_real*
- FILE 1004
 - STD_FILES 1001
- put_string*
- FILE 1004
 - STD_FILES 1001
- put_substring*
- STRING 1000
- Q**
- Query_mark 141
- quotient*
- INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
- R**
- raise*
- EXCEPTIONS 1007
- read_character*
- FILE 1003
 - STD_FILES 1001
- read_integer*
- FILE 1003
 - STD_FILES 1001
- read_line*
- FILE 1003
 - STD_FILES 1001
- Read_only 513
- read_real*
- FILE 1004
 - STD_FILES 1001
- read_stream*
- FILE 1004
 - STD_FILES 1001
- read_word*
- FILE 1004
 - REAL 991, 993
- Real_bits*
- PLATFORM 1009
- Real_constant 788
- reattachment
- principle 599
- redeclaration 21
- unfolded 318
- Redeclaration rule 313
- Redefine 307
- redefine** 18
- Redefine Subclause rule 307
- redefinition 17
- reference** 124
- reference type 10
- reflection, see introspection
- reflective facilities, see introspection
- refresh*
- ONCE_MANAGER 1010
- refresh_all*
- ONCE_MANAGER 1010
- refresh_all_except*
- ONCE_MANAGER 1010
- refresh_some*
- ONCE_MANAGER 1010
- Registered_language 837
- reliability 6
- remainder*
- INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
- rename*
- STRING 998
- remove*
- STRING 1000
- Rename 183
- rename** 17
- Rename Clause rule 185
- Rename_list 183
- Rename_pair 183
- Renaming 357
- renaming 17
- Repeated Inheritance Consistency constraint 466
- Repeated Inheritance rule 438
- require** 11
- Rescue 701
- rescue** 14
- rescue clause rule 701
- resize*
- ARRAY 996
 - STRING 1000
- Result** 9
- retrieved*
- STORABLE 1005
- Retry 701
- retry
- rule 701
- retry** 14
- Retry rule 701
- reusability 5
- reverse assignment, see assignment attempt
- Ribet, Philippe 1056
- right_adjust*
- STRING 1000
- Rist, Robert 1057
- Robertson, Keith 1058
- Rochat, Kim 1058
- Rochat, Roxanne 1058
- root class 23
- root creation procedure 23
- root object 23
- Root Procedure rule 113
- Root Type rule 112
- rounded*
- REAL, REAL_GENERAL 992
- Rousseau, Roger 1057, 1058, 1059
- ROUTINE** 1011, 1013
- routine 8
- precondition-free 426
- Routine_failure*
- EXCEPTIONS 1007
- Routine_mark** 222
- rule, see under “validity constraints”, and also under each rule’s name

S

Sada, Frédérique 1058
 Sar, Savrak 1055
 Sarkela, John 1058, 1059
 Sarkis, Jean-Pierre 1056
 Satchell, Marcel 1056, 1060
 Schmidt, Heinz 1057
 Schoeller, Bernd 1056
 Schramm, Andreas 1058
 Schweitzer, Michael 1057, 1059
 Select rule 463
 semantics
 compositional, non-compositional 777
 operational, non-operational 777
set_error_default
 STD_FILES 1001
set_operands
 FUNCTION 1013
 PREDICATE 1014
 ROUTINE 1011, 1012
set_output_default
 STD_FILES 1001
 shared library 858–??
 Shelley, Norman 1058
 short form 14
Sign 788
sign
 INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 984
 REAL, REAL_GENERAL 991
 signature
 conformance 386
 signature conformance 386
 Simic, Zoran 1055, 1056
 Simon, Raphaël 1055, 1060
Single_constraint 357
 Single-level Call rule 668
 SmartEiffel 1056
 Sommarskog, Erland 1058
Special_expression 761
standard_default
 STD_FILES 1001
 Stapf, Emmanuel 1055, 1056, 1058, 1059, 1060
STD_FILES 1001
 Stephan, Philippe 1055
STORABLE 1005

STRING 998
String_content 795
 strong typing 6, 19
 subcontracting 19
substring
 STRING 1000
substring_index
 STRING 998
 Switzer, Robert 1057, 1058
 system 23–24

T

Tag 232
Tag_mark 232
tail
 STRING 1000
 Tanzer, Christian 1058
Target 626
target
 FUNCTION 1013
 PREDICATE 1014
 ROUTINE 1011, 1012
 target of a call 628
 Target rule 635
 target type of a call 629
 Tarvydas, Paul 1058
 Texier, Emmanuel 1055, 1060
then 7
Then_part 481
Then_part_list 481
 Thomas, Pete 1057
three_way_comparison
 CHARACTER 983
 COMPARABLE 978
 INTEGER, INTEGER_8, INTEGER_16, INTEGER_32, INTEGER_GENERAL 985
 PART_COMPARABLE 977
 REAL, REAL_GENERAL 991
 STRING 999
to_boolean
 STRING 1000
to_integer
 STRING 1000
to_lower
 STRING 1000
to_next_line
 FILE 1003
 STD_FILES 1001
to_real

STRING 1000
to_upper
 STRING 1000
true 15
truncated_to_integer
 REAL, REAL_GENERAL 992
 tuple type
 conformance 397
Tuple_parameter_list 372
Tuple_parameters 372
Tuple_type 372
 Tynor, Steve 1058, 1059
TYPE 976
Type 328
 type 10–11
 basic 10
 expanded, see expanded type
 reference, see reference type
Type_interval 485
Type_list 350
Type_mark 141

U

Unary 154
Unary_expression 766
 unconstrained, see genericity
Undefine 308, 463
 Undefine Subclause rule 308
 unfolded form 100–??, 1074
 of a creation expression 561, 562
 of a creation instruction 530, 552, 553, 555, 556, 557, 563
 of a **Creators** part 548, 550, 551, 552
 of a multi-branch 485, 486, 488, 489
 of an anchored type 344, 345
 of an assertion 237, 547
 of an assigner call 610
 of an interval 487
 unfolded feature list of an **Only** clause 244
 unfolded redeclaration 318
 Unicode 880
 Universal Conformance principle 173
 universe 23
Unlabeled_assertion_clause 232
Unqualified_call 626

- Unregistered_language** 829
- up_to*
- CHARACTER** 983
 - COMPARABLE** 978
 - INTEGER_GENERAL** 984
 - PART_COMPARABLE** 977
 - REAL_GENERAL** 991
 - STRING** 998
 - TYPE** 976
- upper*
- ARRAY** 996
 - INTERVAL** 981
- V**
- valid_index*
- ANONYMOUS** 997
 - ARRAY** 996
 - STRING** 999
- valid_operands*
- FUNCTION** 1013
 - PREDICATE** 1014
 - ROUTINE** 1011, 1012
- Validity constraints**
- VAON** (Only Clause) 243
 - VAOX** (Old Expression) 239
 - VAPE** (Precondition Export) 237
 - VAVE** (Variant Expression) 251
 - VBAC** (Assigner Call rule) 610
 - VBAR** (Assignment rule) 590
 - VBGV** (General Validity) 98
 - VCCH** (Class Header) 126
 - VCFG** (Formal Generic) 351
 - VCRN** (ending comments, no longer present) 1072
 - VDJR** (Join Rule) 319
 - VDPR** (Precursor) 304
 - VDRD** (Redeclaration rule) 313
 - VDRS** (Redefine Subclause) 307
 - VDUS** (Undefine Subclause) 308
 - VEEN** (Entity) 513
 - VEVA** (Variable) 514
 - VEVI** (Variable Initialization) 519
 - VFAC** (Assigner Command) 156
 - VFAV** (Alias Validity) 163
 - VFFB** (Feature Body) 144
 - VFFD** (Feature Declaration) 162
 - VGCC** (Creation Clause) 548
 - VGCE** (Creation Expression) 562
 - VGCI** (Creation Instruction) 553, 688
 - VGCP** (Creation instruction Properties) 555
 - VGCP** (Creation Precondition) 547
 - VGCS** (Creation System-Validity) 687
 - VGEX** (Creation Expression Properties) 562
 - VHCA** (Class Any rule) 173
 - VHPR** (Parent rule) 178
 - VHRC** (Rename Clause) 185
 - VHUC** (Universal Conformance rule, theorem rather than separate validity rule) 173
 - VIID** (Identifier) 891
 - VIIN** (Integer) 901
 - VLCP** (Clients part, no longer present) 1073
 - VLEL** (Export List) 210
 - VMCS** (Call Sharing rule) 458
 - VMFN** (Feature Name) 474, 475
 - VMRC** (Repeated Inheritance Consistency constraint) 466
 - VMSS** (Select Subclause rule) 463
 - VMSS** (Select Subclause), no longer present 1074
 - VNCx** (conformance rules):
 - VNCC** (general conformance) 388
 - VNCF** (formal generic) 393
 - VNCF** (non-generic) 390
 - VNCS** (signature) 386, 396
 - VNCT** (tuple types) 397
 - VOIN** (Interval) 487
 - VOMB** (Multi-Branch) 488
 - VPCA** (Call Agent) 754
 - VPIA** (Inline Agent) 755
 - VPIR** (Inline Agent requirements) 756
 - VQMC** (Manifest Constant) 503
 - VRED** (Entity Declaration) 221
 - VRFA** (Formal Argument) 220
 - VRLV** (Local Variable) 226
 - VSCN** (Class Name) 111
 - VSRP** (Root Procedure) 113
 - VSRT** (Root Type) 112
 - VTAT** (Anchored Type) 345
 - VTCT** (Class Type) 333
 - VTGC** (Generic Constraint) 357
 - VTGD** (Constrained Genericity) 359
 - VTMC** (Multiple Constraints) 369
 - VUAR** (Argument) 634
 - VUCC** (Class-Level Call) 636
 - VUCU** (Call Use) 623
 - VUDA** (Descendant Argument) 667
 - VUEX** (Export) 632
 - VUGC** (General Call) 681
 - VUNO** (Non-Object Call) 631
 - VUOT** (Object Test) 659
 - VUSC** (Single-level Call) 668
 - VUTA** (Target) 635
 - VWBR** (Bracket Expression) 780
 - VWCA** (Constant Attribute) 787
 - VWER** (Expression) 781
 - VWID** (Identifier) 782
 - VWMQ** (Manifest-Type Qualifier) 791
 - VWOE** (Operator Expression) 772
 - VWVS** (Verbatim String) 800
 - VXRC** (rescue clause) 701
 - VXRT** (retry) 701
 - VYCP** (Conversion Procedure) 411
 - VYCF** (Conversion Function) 413
 - VZAR** (Address) 834
 - VZCC** (C external) 846
 - VZCI** (C Inline) 848
 - VZDL** (External DLL) 857
 - VZEF** (External File) 841
 - VZES** (External Signature) 839
- Variable** 512
- Variable Initialization rule 519
- Variable rule 514
- Variable_attribute** 512
- Variant Expression rule 251
- Verbatim String rule 800
- Verbatim_string** 795
- Verbatim_string_opener** 795
- Void**
- no longer reserved word (as it was in Eiffel 1 and 2) 1083
- Void_attached_to_expanded EXCEPTIONS** 1007
- Void_call_target EXCEPTIONS** 1007
- W**
- Waldén, Kim 1056, 1059
- Watkins, Dan 1060

Weedon, Ray 1057
Weiner, Bob 1058
Wells, Kevin 1060
When_part 485
When_part_list 485
Wiener, Richard 1057
wipe_out
STRING 1000

Y

Yost, David 1058
Yuksel, Deniz 1056

Z

Zendra, Olivier 1056
zero
INTEGER, INTEGER_8,
INTEGER_16, INTEGER_32,
INTEGER_GENERAL 984
NUMERIC 980
REAL, REAL_GENERAL 991