

---

---

**Information technology —  
Eiffel: Analysis, Design and Programming  
Language**

*Technologies de l'information — Eiffel: Langage d'analyse, conception  
et programmation*



Reference number  
ISO/IEC 25436:2006(E)

**AVA** Documents  
Document Sharing Hub  
© ISO/IEC 2006

**PDF disclaimer**

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2006

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office  
Case postale 56 • CH-1211 Geneva 20  
Tel. + 41 22 749 01 11  
Fax + 41 22 749 09 47  
E-mail [copyright@iso.org](mailto:copyright@iso.org)  
Web [www.iso.org](http://www.iso.org)

Published in Switzerland

# Contents

Page

|          |                                                  |           |
|----------|--------------------------------------------------|-----------|
| <b>1</b> | <b>Scope .....</b>                               | <b>1</b>  |
| 1.1      | Overview .....                                   | 1         |
| 1.2      | “The Standard” .....                             | 1         |
| 1.3      | Aspects covered .....                            | 1         |
| 1.4      | Aspects not covered .....                        | 1         |
| <b>2</b> | <b>Conformance .....</b>                         | <b>2</b>  |
| 2.1      | Definition .....                                 | 2         |
| 2.2      | Compatibility and non-default options .....      | 2         |
| 2.3      | Departure from the Standard .....                | 2         |
| <b>3</b> | <b>Normative references .....</b>                | <b>2</b>  |
| 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         |
| <b>4</b> | <b>Definitions .....</b>                         | <b>3</b>  |
| <b>5</b> | <b>Notational conventions .....</b>              | <b>3</b>  |
| 5.1      | Standard elements .....                          | 3         |
| 5.2      | Normative elements .....                         | 4         |
| 5.3      | Rules on definitions .....                       | 4         |
| 5.4      | Use of defined terms .....                       | 4         |
| 5.5      | Unfolded forms .....                             | 5         |
| 5.6      | Language description .....                       | 5         |
| 5.7      | Validity: “if and only if” rules .....           | 5         |
| <b>6</b> | <b>Acronyms and abbreviations .....</b>          | <b>5</b>  |
| 6.1      | Name of the language .....                       | 5         |
| 6.2      | Pronunciation .....                              | 5         |
| <b>7</b> | <b>General description .....</b>                 | <b>6</b>  |
| 7.1      | Design principles .....                          | 6         |
| 7.2      | Object-oriented design .....                     | 6         |
| 7.3      | Classes .....                                    | 7         |
| 7.4      | Types .....                                      | 10        |
| 7.5      | Assertions .....                                 | 11        |
| 7.6      | Exceptions .....                                 | 13        |
| 7.7      | Genericity .....                                 | 14        |
| 7.8      | Inheritance .....                                | 15        |
| 7.9      | Polymorphism and dynamic binding .....           | 17        |
| 7.10     | Combining genericity and inheritance .....       | 18        |
| 7.11     | Deferred classes .....                           | 19        |
| 7.12     | Tuples and agents .....                          | 21        |
| 7.13     | Type- and void-safety .....                      | 22        |
| 7.14     | Putting a system together .....                  | 22        |
| <b>8</b> | <b>Language specification .....</b>              | <b>23</b> |
| 8.1      | General organization .....                       | 23        |
| 8.2      | Syntax, validity and semantics .....             | 23        |
| 8.2.1    | Definition: Syntax, BNF-E .....                  | 23        |
| 8.2.2    | Definition: Component, construct, specimen ..... | 24        |
| 8.2.3    | Construct Specimen convention .....              | 24        |
| 8.2.4    | Construct Name convention .....                  | 24        |

|            |                                                                          |           |
|------------|--------------------------------------------------------------------------|-----------|
| 8.2.5      | Definition: Terminal, non-terminal, token .....                          | 24        |
| 8.2.6      | Definition: Production .....                                             | 24        |
| 8.2.7      | Kinds of production .....                                                | 25        |
| 8.2.8      | Definition: Aggregate production .....                                   | 25        |
| 8.2.9      | Definition: Choice production .....                                      | 25        |
| 8.2.10     | Definition: Repetition production, separator .....                       | 25        |
| 8.2.11     | Basic syntax description rule .....                                      | 25        |
| 8.2.12     | Definition: Non-production syntax rule .....                             | 26        |
| 8.2.13     | Textual conventions .....                                                | 26        |
| 8.2.14     | Definition: Validity constraint .....                                    | 27        |
| 8.2.15     | Definition: Valid .....                                                  | 27        |
| 8.2.16     | Validity: General Validity rule .....                                    | 27        |
| 8.2.17     | Definition: Semantics .....                                              | 27        |
| 8.2.18     | Definition: Execution terminology .....                                  | 27        |
| 8.2.19     | Semantics: Case Insensitivity principle .....                            | 27        |
| 8.2.20     | Definition: Upper name, lower name .....                                 | 28        |
| 8.2.21     | Syntax (non-production): Semicolon Optionality rule .....                | 28        |
| <b>8.3</b> | <b>The architecture of Eiffel software .....</b>                         | <b>29</b> |
| 8.3.1      | Definition: Cluster, subcluster, contains directly, contains .....       | 29        |
| 8.3.2      | Definition: Terminal cluster, internal cluster .....                     | 29        |
| 8.3.3      | Definition: Universe .....                                               | 29        |
| 8.3.4      | Syntax: : Class names .....                                              | 30        |
| 8.3.5      | Validity: Class Name rule .....                                          | 30        |
| 8.3.6      | Semantics: Class name semantics .....                                    | 30        |
| 8.3.7      | Definition: System, root type name, root procedure name .....            | 30        |
| 8.3.8      | Definition: Type dependency .....                                        | 30        |
| 8.3.9      | Validity: Root Type rule .....                                           | 31        |
| 8.3.10     | Validity: Root Procedure rule .....                                      | 31        |
| 8.3.11     | Definition: Root type, root procedure, root class .....                  | 32        |
| 8.3.12     | Semantics: System execution .....                                        | 32        |
| <b>8.4</b> | <b>Classes .....</b>                                                     | <b>32</b> |
| 8.4.1      | Definition: Current class .....                                          | 32        |
| 8.4.2      | Syntax: : Class declarations .....                                       | 32        |
| 8.4.3      | Syntax: : Notes .....                                                    | 32        |
| 8.4.4      | Semantics: Notes semantics .....                                         | 33        |
| 8.4.5      | Syntax: : Class headers .....                                            | 33        |
| 8.4.6      | Validity: Class Header rule .....                                        | 33        |
| 8.4.7      | Definition: Expanded, frozen, deferred, effective class .....            | 33        |
| 8.4.8      | Syntax: : Obsolete marks .....                                           | 34        |
| 8.4.9      | Semantics: Obsolete semantics .....                                      | 34        |
| <b>8.5</b> | <b>Features .....</b>                                                    | <b>34</b> |
| 8.5.1      | Definition: Inherited, immediate; origin; redeclaration; introduce ..... | 34        |
| 8.5.2      | Syntax: : Feature parts .....                                            | 35        |
| 8.5.3      | Feature categories: overview .....                                       | 35        |
| 8.5.4      | Syntax: : Feature declarations .....                                     | 35        |
| 8.5.5      | Syntax: : New feature lists .....                                        | 36        |
| 8.5.6      | Syntax: : Feature bodies .....                                           | 36        |
| 8.5.7      | Validity: Feature Body rule .....                                        | 36        |
| 8.5.8      | Validity: Definition: Variable attribute .....                           | 37        |
| 8.5.10     | Definition: Routine, function, procedure .....                           | 37        |
| 8.5.11     | Definition: Command, query .....                                         | 37        |
| 8.5.12     | Definition: Signature, argument signature of a feature .....             | 27        |
| 8.5.13     | Feature principle .....                                                  | 38        |
| 8.5.14     | Syntax: : Feature names .....                                            | 38        |
| 8.5.15     | Syntax (non-production): Alias Syntax rule .....                         | 38        |

|            |                                                                      |           |
|------------|----------------------------------------------------------------------|-----------|
| 8.5.16     | Definition: Operator feature, bracket feature, identifier-only ..... | 39        |
| 8.5.17     | Definition: Identifier of a feature name .....                       | 39        |
| 8.5.18     | Validity: Feature Identifier principle .....                         | 39        |
| 8.5.19     | Definition: Same feature name, same operator, same alias .....       | 39        |
| 8.5.20     | Syntax: : Operators .....                                            | 40        |
| 8.5.21     | Syntax: : Assigner marks .....                                       | 40        |
| 8.5.22     | Validity: Assigner Command rule .....                                | 40        |
| 8.5.23     | Definition: Synonym .....                                            | 41        |
| 8.5.24     | Definition: Unfolded form of a possibly multiple declaration .....   | 41        |
| 8.5.25     | Validity: Feature Declaration rule .....                             | 42        |
| 8.5.26     | Validity: Alias Validity rule .....                                  | 43        |
| <b>8.6</b> | <b>The inheritance relation .....</b>                                | <b>43</b> |
| 8.6.1      | Syntax: : Inheritance parts .....                                    | 43        |
| 8.6.2      | Syntax (non-production): Feature adaptation rule .....               | 44        |
| 8.6.3      | Definition: Parent part for a type, for a class .....                | 44        |
| 8.6.4      | Validity: Class <b>ANY</b> rule .....                                | 44        |
| 8.6.5      | Validity: Universal Conformance principle .....                      | 44        |
| 8.6.6      | Definition: Unfolded inheritance part of a class .....               | 45        |
| 8.6.7      | Definition: Multiple, single inheritance .....                       | 45        |
| 8.6.8      | Definition: Inherit, heir, parent .....                              | 45        |
| 8.6.9      | Definition: Conforming, non-conforming parent .....                  | 45        |
| 8.6.10     | Definition: Ancestor types of a type, of a class .....               | 45        |
| 8.6.11     | Definition: Ancestor, descendant .....                               | 46        |
| 8.6.12     | Definition: Proper ancestor, proper descendant .....                 | 46        |
| 8.6.13     | Validity: Parent rule .....                                          | 46        |
| 8.6.14     | Syntax: : Rename clauses .....                                       | 46        |
| 8.6.15     | Validity: Rename Clause rule .....                                   | 47        |
| 8.6.16     | Semantics: Renaming principle .....                                  | 47        |
| 8.6.17     | Definition: Final name, extended final name, final name set .....    | 47        |
| 8.6.18     | Definition: Inherited name .....                                     | 48        |
| 8.6.19     | Definition: Declaration for a feature .....                          | 48        |
| <b>8.7</b> | <b>Clients and exports .....</b>                                     | <b>48</b> |
| 8.7.1      | Definition: Client relation between classes and types .....          | 48        |
| 8.7.2      | Definition: Client relation between classes .....                    | 49        |
| 8.7.3      | Definition: Supplier .....                                           | 49        |
| 8.7.4      | Definition: Simple client .....                                      | 49        |
| 8.7.5      | Definition: Expanded client .....                                    | 49        |
| 8.7.6      | Definition: Generic client, generic supplier .....                   | 49        |
| 8.7.7      | Definition: Indirect client .....                                    | 49        |
| 8.7.8      | Definition: Client set of a <b>Clients</b> part .....                | 49        |
| 8.7.9      | Syntax: : Clients .....                                              | 50        |
| 8.7.10     | Syntax: : Export adaptation .....                                    | 50        |
| 8.7.11     | Validity: Export List rule .....                                     | 50        |
| 8.7.12     | Definition: Client set of a feature .....                            | 51        |
| 8.7.13     | Definition: Available for call, available .....                      | 51        |
| 8.7.14     | Definition: Exported, selectively available, secret .....            | 51        |
| 8.7.15     | Definition: Secret, public property .....                            | 52        |
| 8.7.16     | Definition: Incremental contract view, short form .....              | 52        |
| 8.7.17     | Definition: Contract view, flat-short form .....                     | 53        |
| <b>8.8</b> | <b>Routines .....</b>                                                | <b>53</b> |
| 8.8.1      | Definition: Formal argument, actual argument .....                   | 53        |
| 8.8.2      | Syntax: : Formal argument and entity declarations .....              | 55        |
| 8.8.3      | Validity: Formal Argument rule .....                                 | 55        |
| 8.8.4      | Validity: Entity Declaration rule .....                              | 54        |
| 8.8.5      | Syntax: : Routine bodies .....                                       | 54        |

|             |                                                                                             |           |
|-------------|---------------------------------------------------------------------------------------------|-----------|
| 8.8.6       | Definition: Once routine, once procedure, once function .....                               | 54        |
| 8.8.7       | Syntax: : Local variable declarations .....                                                 | 55        |
| 8.8.8       | Validity: Local Variable rule .....                                                         | 55        |
| 8.8.9       | Definition: Local variable .....                                                            | 55        |
| 8.8.10      | Syntax: : Instructions .....                                                                | 55        |
| <b>8.9</b>  | <b>Correctness and contracts .....</b>                                                      | <b>56</b> |
| 8.9.1       | Syntax: : Assertions .....                                                                  | 56        |
| 8.9.2       | Syntax (non-production): Assertion Syntax rule .....                                        | 56        |
| 8.9.3       | Definition: Precondition, postcondition, invariant .....                                    | 57        |
| 8.9.4       | Definition: Contract, subcontract .....                                                     | 57        |
| 8.9.5       | Validity: Precondition Export rule .....                                                    | 57        |
| 8.9.6       | Validity: Definition: Availability of an assertion clause .....                             | 57        |
| 8.9.7       | Syntax: : “Old” postcondition expressions .....                                             | 58        |
| 8.9.8       | Validity: Old Expression rule .....                                                         | 58        |
| 8.9.9       | Semantics: Old Expression Semantics, associated variable, associated exception marker ..... | 58        |
| 8.9.10      | Semantics: Associated Variable Semantics .....                                              | 59        |
| 8.9.11      | Syntax: : “Only” postcondition clauses .....                                                | 59        |
| 8.9.12      | Validity: Only Clause rule .....                                                            | 59        |
| 8.9.13      | Definition: Unfolded feature list of an <b>Only</b> clause .....                            | 60        |
| 8.9.14      | Definition: Unfolded <b>Only</b> clause .....                                               | 60        |
| 8.9.15      | Definition: Hoare triple notation (total correctness) .....                                 | 60        |
| 8.9.16      | Semantics: Class consistency .....                                                          | 60        |
| 8.9.17      | Syntax: : Check instructions .....                                                          | 61        |
| 8.9.18      | Definition: Check-correct .....                                                             | 61        |
| 8.9.19      | Syntax: : Variants .....                                                                    | 61        |
| 8.9.20      | Validity: Variant Expression rule .....                                                     | 61        |
| 8.9.21      | Definition: Loop invariant and variant .....                                                | 61        |
| 8.9.22      | Definition: Loop-correct .....                                                              | 61        |
| 8.9.23      | Definition: Correctness (class) .....                                                       | 62        |
| 8.9.24      | Definition: Local unfolded form of an assertion .....                                       | 62        |
| 8.9.25      | Semantics: Evaluation of an assertion .....                                                 | 62        |
| 8.9.26      | Semantics: Assertion monitoring .....                                                       | 62        |
| 8.9.27      | Semantics: Assertion violation .....                                                        | 63        |
| 8.9.28      | Semantics: Assertion semantics .....                                                        | 63        |
| 8.9.29      | Semantics: Assertion monitoring levels .....                                                | 63        |
| <b>8.10</b> | <b>Feature adaptation .....</b>                                                             | <b>63</b> |
| 8.10.1      | Definition: Redeclare, redeclaration .....                                                  | 64        |
| 8.10.2      | Definition: Unfolded form of an assertion .....                                             | 64        |
| 8.10.3      | Definition: Assertion extensions .....                                                      | 64        |
| 8.10.4      | Definition: Covariance-aware form of an inherited assertion .....                           | 64        |
| 8.10.5      | Definition: Combined precondition, postcondition .....                                      | 65        |
| 8.10.6      | Definition: Inherited as effective, inherited as deferred .....                             | 65        |
| 8.10.7      | Definition: Effect, effecting .....                                                         | 65        |
| 8.10.8      | Definition: Redefine, redefinition .....                                                    | 65        |
| 8.10.9      | Definition: Name clash .....                                                                | 66        |
| 8.10.10     | Syntax: : Precursor .....                                                                   | 66        |
| 8.10.11     | Definition: Relative unfolded form of a <b>Precursor</b> .....                              | 66        |
| 8.10.12     | Validity: Precursor rule .....                                                              | 66        |
| 8.10.13     | Definition: Unfolded form of a <b>Precursor</b> .....                                       | 67        |
| 8.10.14     | Semantics: Precursor semantics .....                                                        | 67        |
| 8.10.15     | Syntax: : Redefinition .....                                                                | 67        |
| 8.10.16     | Validity: Redefine Subclause rule .....                                                     | 67        |
| 8.10.17     | Semantics: Redefinition semantics .....                                                     | 67        |
| 8.10.18     | Syntax: : Undefine clauses .....                                                            | 68        |

|             |                                                                             |           |
|-------------|-----------------------------------------------------------------------------|-----------|
| 8.10.19     | Validity: Undefine Subclause rule .....                                     | 68        |
| 8.10.20     | Semantics: Undefinition semantics .....                                     | 68        |
| 8.10.21     | Definition: Effective, deferred feature .....                               | 68        |
| 8.10.22     | Definition: Effecting .....                                                 | 68        |
| 8.10.23     | Deferred Class property .....                                               | 68        |
| 8.10.24     | Effective Class property .....                                              | 68        |
| 8.10.25     | Definition: Origin, seed .....                                              | 69        |
| 8.10.26     | Validity: Redeclaration rule .....                                          | 69        |
| 8.10.27     | Definition: Precursor (joined features) .....                               | 70        |
| 8.10.28     | Validity: Join rule .....                                                   | 70        |
| 8.10.29     | Semantics: Join Semantics rule .....                                        | 71        |
| <b>8.11</b> | <b>Types .....</b>                                                          | <b>71</b> |
| 8.11.1      | Syntax: : Types .....                                                       | 71        |
| 8.11.2      | Semantics: Direct instances and values of a type .....                      | 72        |
| 8.11.3      | Semantics: Instances of a type .....                                        | 72        |
| 8.11.4      | Semantics: Instance principle .....                                         | 72        |
| 8.11.5      | Definition: Instances, direct instances of a class .....                    | 72        |
| 8.11.6      | Base principle; base class, base type, based .....                          | 72        |
| 8.11.7      | Base rule .....                                                             | 73        |
| 8.11.8      | Validity: Class Type rule .....                                             | 73        |
| 8.11.9      | Semantics: Type Semantics rule .....                                        | 73        |
| 8.11.10     | Definition: Base class and base type of an expression .....                 | 73        |
| 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 .....                                                | 74        |
| 8.11.14     | Definition: Anchor, anchored type, anchored entity .....                    | 74        |
| 8.11.15     | Definition: Anchor set; cyclic anchor .....                                 | 74        |
| 8.11.16     | Definition: Types and classes involved in a type .....                      | 75        |
| 8.11.17     | Definition: Deanchored form of a type .....                                 | 75        |
| 8.11.18     | Validity: Anchored Type rule .....                                          | 75        |
| 8.11.19     | Definition: Attached, detachable .....                                      | 76        |
| 8.11.20     | Semantics: Attached type semantics .....                                    | 76        |
| 8.11.21     | Definition: Stand-alone type .....                                          | 77        |
| <b>8.12</b> | <b>Genericity .....</b>                                                     | <b>77</b> |
| 8.12.1      | Syntax: : Actual generic parameters .....                                   | 77        |
| 8.12.2      | Syntax: : Formal generic parameters .....                                   | 77        |
| 8.12.3      | Validity: Formal Generic rule .....                                         | 77        |
| 8.12.4      | Definition: Generic class; constrained, unconstrained .....                 | 78        |
| 8.12.5      | Definition: Generic derivation, generic type, non-generic type .....        | 78        |
| 8.12.6      | Definition: Self-initializing formal parameter .....                        | 78        |
| 8.12.7      | Definition: Constraint, constraining types of a <b>Formal_generic</b> ..... | 78        |
| 8.12.8      | Syntax: : Generic constraints .....                                         | 78        |
| 8.12.9      | Validity: Generic Constraint rule .....                                     | 79        |
| 8.12.10     | Definition: Constraining creation features .....                            | 79        |
| 8.12.11     | Validity: Generic Derivation rule .....                                     | 79        |
| 8.12.12     | Definition: Generic-creation-ready type .....                               | 80        |
| 8.12.13     | Semantics: Generically derived class type semantics .....                   | 80        |
| 8.12.14     | Definition: Base type of a single-constrained formal generic .....          | 80        |
| 8.12.15     | Definition: Base type of an unconstrained formal generic .....              | 81        |
| 8.12.16     | Definition: Reference or expanded status of a formal generic .....          | 81        |
| 8.12.17     | Definition: Current type .....                                              | 81        |
| 8.12.18     | Definition: Features of a type .....                                        | 81        |
| 8.12.19     | Definition: Generic substitution .....                                      | 81        |
| 8.12.20     | Generic Type Adaptation rule .....                                          | 81        |
| 8.12.21     | Definition: Generically constrained feature name .....                      | 82        |

|             |                                                                       |           |
|-------------|-----------------------------------------------------------------------|-----------|
| 8.12.22     | Validity: Multiple Constraints rule .....                             | 82        |
| 8.12.23     | Definition: Base type of a multi-constraint formal generic type ..... | 82        |
| <b>8.13</b> | <b>Tuples .....</b>                                                   | <b>82</b> |
| 8.13.1      | Syntax: : Tuple types .....                                           | 82        |
| 8.13.2      | Syntax: : Manifest tuples .....                                       | 82        |
| 8.13.3      | Definition: Type sequence of a tuple type .....                       | 83        |
| 8.13.4      | Definition: Value sequences associated with a tuple type .....        | 83        |
| <b>8.14</b> | <b>Conformance .....</b>                                              | <b>83</b> |
| 8.14.1      | Definition: Compatibility between types .....                         | 83        |
| 8.14.2      | Definition: Compatibility between expressions .....                   | 84        |
| 8.14.3      | Definition: Expression conformance .....                              | 84        |
| 8.14.4      | Validity: Signature conformance .....                                 | 84        |
| 8.14.5      | Definition: Covariant argument .....                                  | 84        |
| 8.14.6      | Validity: General conformance .....                                   | 85        |
| 8.14.7      | Definition: Conformance path .....                                    | 85        |
| 8.14.8      | Validity: Direct conformance: reference types .....                   | 85        |
| 8.14.9      | Validity: Direct conformance: formal generic .....                    | 86        |
| 8.14.10     | Validity: Direct conformance: expanded types .....                    | 86        |
| 8.14.11     | Validity: Direct conformance: tuple types .....                       | 86        |
| <b>8.15</b> | <b>Convertibility .....</b>                                           | <b>87</b> |
| 8.15.1      | Definition: Conversion procedure, conversion type .....               | 87        |
| 8.15.2      | Definition: Conversion query, conversion feature .....                | 87        |
| 8.15.3      | Validity: Conversion principle .....                                  | 87        |
| 8.15.4      | Validity: Conversion Asymmetry principle .....                        | 87        |
| 8.15.5      | Validity: Conversion Non-Transitivity principle .....                 | 87        |
| 8.15.6      | Syntax: : Converter clauses .....                                     | 87        |
| 8.15.7      | Validity: Conversion Procedure rule .....                             | 87        |
| 8.15.8      | Validity: Conversion Query rule .....                                 | 88        |
| 8.15.9      | Definition: Converting to a class .....                               | 88        |
| 8.15.10     | Definition: Converting to and from a type .....                       | 88        |
| 8.15.11     | Definition: Converting “through” .....                                | 88        |
| 8.15.12     | Semantics: Conversion semantics .....                                 | 89        |
| 8.15.13     | Definition: Explicit conversion .....                                 | 89        |
| 8.15.14     | Validity: Expression convertibility .....                             | 89        |
| 8.15.15     | Definition: Statically satisfied precondition .....                   | 89        |
| 8.15.16     | Validity: Precondition-free routine .....                             | 90        |
| <b>8.16</b> | <b>Repeated inheritance .....</b>                                     | <b>91</b> |
| 8.16.1      | Definition: Repeated inheritance, ancestor, descendant .....          | 91        |
| 8.16.2      | Semantics: Repeated Inheritance rule .....                            | 91        |
| 8.16.3      | Definition: Sharing, replication .....                                | 92        |
| 8.16.4      | Validity: Call Sharing rule .....                                     | 92        |
| 8.16.5      | Semantics: Replication Semantics rule .....                           | 92        |
| 8.16.6      | Syntax: : Select clauses .....                                        | 92        |
| 8.16.7      | Validity: Select Subclause rule .....                                 | 92        |
| 8.16.8      | Definition: Version .....                                             | 92        |
| 8.16.9      | Definition: Multiple versions .....                                   | 93        |
| 8.16.10     | Validity: Repeated Inheritance Consistency constraint .....           | 93        |
| 8.16.11     | Definition: Dynamic binding version .....                             | 93        |
| 8.16.12     | Definition: Inherited features .....                                  | 93        |
| 8.16.13     | Semantics: Join-Sharing Reconciliation rule .....                     | 93        |
| 8.16.14     | Definition: Precursor .....                                           | 93        |
| 8.16.15     | Validity: Feature Name rule .....                                     | 94        |
| 8.16.16     | Validity: Name Clash rule .....                                       | 94        |
| <b>8.17</b> | <b>Control structures .....</b>                                       | <b>95</b> |

|             |                                                                      |            |
|-------------|----------------------------------------------------------------------|------------|
| 8.17.1      | Semantics: Compound (non-exception) semantics .....                  | 95         |
| 8.17.2      | Syntax: : Conditionals .....                                         | 95         |
| 8.17.3      | Definition: Secondary part .....                                     | 95         |
| 8.17.4      | Definition: Prevailing immediately .....                             | 95         |
| 8.17.5      | Semantics: Conditional semantics .....                               | 95         |
| 8.17.6      | Definition: Inspect expression .....                                 | 96         |
| 8.17.7      | Syntax: : Multi-branch instructions .....                            | 96         |
| 8.17.8      | Definition: Interval .....                                           | 96         |
| 8.17.9      | Definition: Unfolded form of a multi-branch .....                    | 96         |
| 8.17.10     | Definition: Unfolded form of an interval .....                       | 96         |
| 8.17.11     | Validity: Interval rule .....                                        | 97         |
| 8.17.12     | Definition: Inspect values of a multi-branch .....                   | 97         |
| 8.17.13     | Validity: Multi-branch rule .....                                    | 97         |
| 8.17.14     | Semantics: Matching branch .....                                     | 98         |
| 8.17.15     | Semantics: Multi-Branch semantics .....                              | 98         |
| 8.17.16     | Syntax: : Loops .....                                                | 98         |
| 8.17.17     | Semantics: Loop semantics .....                                      | 98         |
| 8.17.18     | Syntax: : Debug instructions .....                                   | 98         |
| 8.17.19     | Semantics: Debug semantics .....                                     | 98         |
| <b>8.18</b> | <b>Attributes .....</b>                                              | <b>99</b>  |
| 8.18.1      | Syntax: : Attribute bodies .....                                     | 99         |
| 8.18.2      | Validity: Manifest Constant rule .....                               | 99         |
| <b>8.19</b> | <b>Objects, values and entities .....</b>                            | <b>100</b> |
| 8.19.1      | Semantics: Type, generating type of an object; generator .....       | 100        |
| 8.19.2      | Definition: Reference, void, attached, attached to .....             | 100        |
| 8.19.3      | Semantics: Object principle .....                                    | 100        |
| 8.19.4      | Definition: Object semantics .....                                   | 100        |
| 8.19.5      | Definition: Non-basic class, non-basic type, field .....             | 100        |
| 8.19.6      | Definition: Subobject, composite object .....                        | 101        |
| 8.19.7      | Definition: Entity, variable, read-only .....                        | 101        |
| 8.19.8      | Syntax: : Entities and variables .....                               | 101        |
| 8.19.9      | Validity: Entity rule .....                                          | 101        |
| 8.19.10     | Validity: Variable rule .....                                        | 102        |
| 8.19.11     | Definition: Self-initializing type .....                             | 102        |
| 8.19.12     | Semantics: Default Initialization rule .....                         | 102        |
| 8.19.13     | Definition: Self-initializing variable .....                         | 103        |
| 8.19.14     | Definition: Evaluation position, precedes .....                      | 103        |
| 8.19.15     | Definition: Setter instruction .....                                 | 104        |
| 8.19.16     | Definition: Properly set variable .....                              | 104        |
| 8.19.17     | Validity: Variable Initialization rule .....                         | 104        |
| 8.19.18     | Definition: Variable setting and its value .....                     | 105        |
| 8.19.19     | Definition: Execution context .....                                  | 105        |
| 8.19.20     | Semantics: Variable Semantics .....                                  | 105        |
| 8.19.21     | Semantics: Entity Semantics rule .....                               | 106        |
| <b>8.20</b> | <b>Creating objects .....</b>                                        | <b>106</b> |
| 8.20.1      | Semantics: Creation principle .....                                  | 106        |
| 8.20.2      | Definition: Creation operation .....                                 | 107        |
| 8.20.3      | Validity: Creation Precondition rule .....                           | 107        |
| 8.20.4      | Syntax: : Creators parts .....                                       | 107        |
| 8.20.5      | Definition: Unfolded creators part of a class .....                  | 107        |
| 8.20.6      | Validity: Creation Clause rule .....                                 | 108        |
| 8.20.7      | Definition: Creation procedures of a class .....                     | 108        |
| 8.20.8      | Creation procedure property .....                                    | 108        |
| 8.20.9      | Definition: Creation procedures of a type .....                      | 108        |
| 8.20.10     | Definition: Available for creation; general creation procedure ..... | 109        |

|             |                                                                           |            |
|-------------|---------------------------------------------------------------------------|------------|
| 8.20.11     | Syntax: : Creation instructions .....                                     | 109        |
| 8.20.12     | Definition: Creation target, creation type .....                          | 109        |
| 8.20.13     | Semantics: Creation Type theorem .....                                    | 109        |
| 8.20.14     | Definition: Unfolded form of a creation instruction .....                 | 109        |
| 8.20.15     | Validity: Creation Instruction rule .....                                 | 109        |
| 8.20.16     | Validity: Creation Instruction properties .....                           | 110        |
| 8.20.17     | Semantics: Creation Instruction Semantics .....                           | 110        |
| 8.20.18     | Syntax: : Creation expressions .....                                      | 111        |
| 8.20.19     | Definition: Properties of a creation expression .....                     | 111        |
| 8.20.20     | Validity: Creation Expression rule .....                                  | 111        |
| 8.20.21     | Validity: Creation Expression Properties .....                            | 111        |
| 8.20.22     | Semantics: Creation Expression Semantics .....                            | 112        |
| 8.20.23     | Definition: Garbage Collection, not enough memory available .....         | 112        |
| <b>8.21</b> | <b>Comparing and duplicating objects .....</b>                            | <b>112</b> |
| 8.21.1      | Object comparison features from <i>ANY</i> .....                          | 113        |
| 8.21.2      | Syntax: : Equality expressions .....                                      | 113        |
| 8.21.3      | Semantics: Equality Expression Semantics .....                            | 113        |
| 8.21.4      | Semantics: Inequality Expression Semantics .....                          | 114        |
| 8.21.5      | Copying and cloning features from <i>ANY</i> .....                        | 114        |
| 8.21.6      | Deep equality, copying and cloning .....                                  | 115        |
| <b>8.22</b> | <b>Attaching values to entities .....</b>                                 | <b>115</b> |
| 8.22.1      | Definition: Reattachment, source, target .....                            | 115        |
| 8.22.2      | Syntax: : Assignments .....                                               | 116        |
| 8.22.3      | Validity: Assignment rule .....                                           | 116        |
| 8.22.4      | Semantics: Reattachment principle .....                                   | 116        |
| 8.22.5      | Semantics: Attaching an entity, attached entity .....                     | 116        |
| 8.22.6      | Semantics: Reattachment Semantics .....                                   | 116        |
| 8.22.7      | Semantics: Assignment Semantics .....                                     | 117        |
| 8.22.8      | Definition: Dynamic type .....                                            | 117        |
| 8.22.9      | Definition: Polymorphic expression; dynamic type and class sets .....     | 117        |
| 8.22.10     | Syntax: : Assigner calls .....                                            | 117        |
| 8.22.11     | Validity: Assigner Call rule .....                                        | 118        |
| 8.22.12     | Semantics: Assigner Call semantics .....                                  | 118        |
| <b>8.23</b> | <b>Feature call .....</b>                                                 | <b>118</b> |
| 8.23.1      | Validity: Call Use rule .....                                             | 118        |
| 8.23.2      | Syntax: : Feature calls .....                                             | 118        |
| 8.23.3      | Syntax: : Actual arguments .....                                          | 119        |
| 8.23.4      | Definition: Unqualified, qualified call .....                             | 119        |
| 8.23.5      | Definition: Target of a call .....                                        | 119        |
| 8.23.6      | Definition: Target type of a call .....                                   | 119        |
| 8.23.7      | Definition: Feature of a call .....                                       | 119        |
| 8.23.8      | Definition: Imported form of a <i>Non_object_call</i> .....               | 120        |
| 8.23.9      | Validity: Non-Object Call rule .....                                      | 120        |
| 8.23.10     | Semantics: Non-Object Call Semantics .....                                | 120        |
| 8.23.11     | Validity: Export rule .....                                               | 120        |
| 8.23.12     | Validity: Export Status principle .....                                   | 121        |
| 8.23.13     | Validity: Argument rule .....                                             | 121        |
| 8.23.14     | Validity: Target rule .....                                               | 121        |
| 8.23.15     | Validity: Class-Level Call rule .....                                     | 122        |
| 8.23.16     | Definition: Void-unsafe .....                                             | 122        |
| 8.23.17     | Definition: Target Object .....                                           | 122        |
| 8.23.18     | Semantics: Failed target evaluation of a void-unsafe system .....         | 122        |
| 8.23.19     | Definition: Dynamic feature of a call .....                               | 122        |
| 8.23.20     | Definition: Freshness of a once routine call .....                        | 122        |
| 8.23.21     | Definition: Latest applicable target and result of a non-fresh call ..... | 123        |

|             |                                                                                   |            |
|-------------|-----------------------------------------------------------------------------------|------------|
| 8.23.22     | Semantics: Once Routine Execution Semantics .....                                 | 123        |
| 8.23.23     | Semantics: Current object, current routine .....                                  | 123        |
| 8.23.24     | Semantics: Current Semantics .....                                                | 124        |
| 8.23.25     | Semantics: Non-Once Routine Execution Semantics .....                             | 124        |
| 8.23.26     | Semantics: General Call Semantics .....                                           | 124        |
| 8.23.27     | Definition: Type of a <b>Call</b> used as expression .....                        | 124        |
| 8.23.28     | Semantics: Call Result .....                                                      | 125        |
| 8.23.29     | Semantics: Value of a call expression .....                                       | 125        |
| <b>8.24</b> | <b>Eradicating void calls .....</b>                                               | <b>125</b> |
| 8.24.1      | Syntax: : Object test .....                                                       | 125        |
| 8.24.2      | Definition: Object-Test Local .....                                               | 126        |
| 8.24.3      | Validity: Object Test rule .....                                                  | 126        |
| 8.24.4      | Definition: Conjunctive, disjunctive, implicative;<br>Term, semistrict term ..... | 126        |
| 8.24.5      | Definition: Scope of an Object-Test Local .....                                   | 126        |
| 8.24.6      | Semantics: Object Test semantics .....                                            | 127        |
| 8.24.7      | Semantics: Object-Test Local semantics .....                                      | 127        |
| 8.24.8      | Definition: Read-only void test .....                                             | 127        |
| 8.24.9      | Definition: Scope of a read-only void test .....                                  | 127        |
| 8.24.10     | Definition: Certified Attachment Pattern .....                                    | 127        |
| 8.24.11     | Definition: Attached expression .....                                             | 128        |
| <b>8.25</b> | <b>Typing-related properties .....</b>                                            | <b>128</b> |
| 8.25.1      | Definition: Catcall .....                                                         | 128        |
| 8.25.2      | Validity: Descendant Argument rule .....                                          | 129        |
| 8.25.3      | Validity: Single-level Call rule .....                                            | 129        |
| <b>8.26</b> | <b>Exception handling .....</b>                                                   | <b>129</b> |
| 8.26.1      | Definition: Failure, exception, trigger .....                                     | 129        |
| 8.26.2      | Syntax: : Rescue clauses .....                                                    | 130        |
| 8.26.3      | Validity: Rescue clause rule .....                                                | 130        |
| 8.26.4      | Validity: Retry rule .....                                                        | 130        |
| 8.26.5      | Definition: Exception-correct .....                                               | 130        |
| 8.26.6      | Semantics: Default Rescue Original Semantics .....                                | 130        |
| 8.26.7      | Definition: Rescue block .....                                                    | 130        |
| 8.26.8      | Semantics: Exception Semantics .....                                              | 131        |
| 8.26.9      | Definition: Type of an exception .....                                            | 131        |
| 8.26.10     | Semantics: Exception Cases .....                                                  | 131        |
| 8.26.11     | Semantics: Exception Properties .....                                             | 132        |
| 8.26.12     | Definition: Ignoring, continuing an exception .....                               | 132        |
| <b>8.27</b> | <b>Agents .....</b>                                                               | <b>133</b> |
| 8.27.1      | Definition: Operands of a call .....                                              | 133        |
| 8.27.2      | Definition: Operand position .....                                                | 133        |
| 8.27.3      | Definition: Construction time, call time .....                                    | 133        |
| 8.27.4      | Syntactical forms for a call agent .....                                          | 133        |
| 8.27.5      | Syntax: : Agents .....                                                            | 134        |
| 8.27.6      | Syntax: : Call agent bodies .....                                                 | 134        |
| 8.27.7      | Definition: Target type of a call agent .....                                     | 134        |
| 8.27.8      | Validity: Call Agent rule .....                                                   | 134        |
| 8.27.9      | Definition: Associated feature of an inline agent .....                           | 134        |
| 8.27.10     | Validity: Inline Agent rule .....                                                 | 134        |
| 8.27.11     | Validity: Inline Agent Requirements .....                                         | 135        |
| 8.27.12     | Definition: Call-agent equivalent of an inline agent .....                        | 135        |
| 8.27.13     | Semantics: Semantics of an inline agent .....                                     | 135        |
| 8.27.14     | Semantics: Use of <b>Result</b> in an inline function agent .....                 | 135        |
| 8.27.15     | Definition: Open and closed operands .....                                        | 135        |

|             |                                                                   |            |
|-------------|-------------------------------------------------------------------|------------|
| 8.27.16     | Definition: Open and closed operand positions .....               | 135        |
| 8.27.17     | Definition: Type of an agent expression .....                     | 135        |
| 8.27.18     | Semantics: Agent Expression semantics .....                       | 136        |
| 8.27.19     | Semantics: Effect of executing <i>call</i> on an agent .....      | 136        |
| <b>8.28</b> | <b>Expressions .....</b>                                          | <b>136</b> |
| 8.28.1      | Syntax: : Expressions .....                                       | 136        |
| 8.28.2      | Definition: Subexpression, operand .....                          | 136        |
| 8.28.3      | Semantics: Parenthesized Expression Semantics .....               | 137        |
| 8.28.4      | Syntax: : Operator expressions .....                              | 137        |
| 8.28.5      | Operator precedence levels .....                                  | 137        |
| 8.28.6      | Definition: Parenthesized Form of an expression .....             | 137        |
| 8.28.7      | Definition: Target-converted form of a binary expression .....    | 138        |
| 8.28.8      | Validity: Operator Expression rule .....                          | 138        |
| 8.28.9      | Semantics: Expression Semantics (strict case) .....               | 138        |
| 8.28.10     | Definition: Semistrict operators .....                            | 139        |
| 8.28.11     | Semantics: Operator Expression Semantics (semistrict case) .....  | 139        |
| 8.28.12     | Syntax: : Bracket expressions .....                               | 139        |
| 8.28.13     | Validity: Bracket Expression rule .....                           | 139        |
| 8.28.14     | Definition: Equivalent Dot Form of an expression .....            | 139        |
| 8.28.15     | Validity: Boolean Expression rule .....                           | 140        |
| 8.28.16     | Validity: Identifier rule .....                                   | 140        |
| 8.28.17     | Definition: Type of an expression .....                           | 140        |
| <b>8.29</b> | <b>Constants .....</b>                                            | <b>141</b> |
| 8.29.1      | Syntax: : Constants .....                                         | 141        |
| 8.29.2      | Validity: Constant Attribute rule .....                           | 141        |
| 8.29.3      | Syntax: : Manifest constants .....                                | 141        |
| 8.29.4      | Syntax (non-production): Sign Syntax rule .....                   | 141        |
| 8.29.5      | Syntax (non-production): Character Syntax rule .....              | 141        |
| 8.29.6      | Definition: Type of a manifest constant .....                     | 142        |
| 8.29.7      | Validity: Manifest-Type Qualifier rule .....                      | 142        |
| 8.29.8      | Semantics: Manifest Constant Semantics .....                      | 142        |
| 8.29.9      | Definition: Manifest value of a constant .....                    | 143        |
| 8.29.10     | Syntax: : Manifest strings .....                                  | 143        |
| 8.29.11     | Syntax (non-production): Line sequence .....                      | 143        |
| 8.29.12     | Syntax (non-production): Manifest String rule .....               | 144        |
| 8.29.13     | Definition: <i>Line_wrapping_part</i> .....                       | 144        |
| 8.29.14     | Semantics: Manifest string semantics .....                        | 144        |
| 8.29.15     | Validity: Verbatim String rule .....                              | 144        |
| 8.29.16     | Semantics: Verbatim string semantics .....                        | 144        |
| 8.29.17     | Definition: Prefix, longest break prefix, left-aligned form ..... | 145        |
| <b>8.30</b> | <b>Basic types .....</b>                                          | <b>145</b> |
| 8.30.1      | Definition: Basic types and their sized variants .....            | 145        |
| 8.30.2      | Definition: Sized variants of <i>STRING</i> .....                 | 145        |
| 8.30.3      | Semantics: Boolean value semantics .....                          | 145        |
| 8.30.4      | Semantics: Character types .....                                  | 145        |
| 8.30.5      | Semantics: Integer types .....                                    | 146        |
| 8.30.6      | Semantics: Floating-point types .....                             | 146        |
| 8.30.7      | Semantics: Address semantics .....                                | 146        |
| <b>8.31</b> | <b>Interfacing with C, C++ and other environments .....</b>       | <b>146</b> |
| 8.31.1      | Syntax: : External routines .....                                 | 146        |
| 8.31.2      | Semantics: Address semantics .....                                | 147        |
| 8.31.3      | Syntax: : Registered languages .....                              | 147        |
| 8.31.4      | Syntax: : External signatures .....                               | 147        |
| 8.31.5      | Validity: External Signature rule .....                           | 147        |

|              |                                                                          |            |
|--------------|--------------------------------------------------------------------------|------------|
| 8.31.6       | Semantics: External signature semantics .....                            | 148        |
| 8.31.7       | Syntax: : External file use .....                                        | 148        |
| 8.31.8       | Validity: External File rule .....                                       | 148        |
| 8.31.9       | Semantics: External file semantics .....                                 | 149        |
| 8.31.10      | Syntax: : C externals .....                                              | 149        |
| 8.31.11      | Validity: C External rule .....                                          | 149        |
| 8.31.12      | Semantics: C Inline semantics .....                                      | 150        |
| 8.31.13      | Syntax: : C++ externals .....                                            | 150        |
| 8.31.14      | Validity: C++ External rule .....                                        | 150        |
| 8.31.15      | Semantics: C++ Inline semantics .....                                    | 150        |
| 8.31.16      | Syntax: : DLL externals .....                                            | 150        |
| 8.31.17      | Validity: External DLL rule .....                                        | 151        |
| 8.31.18      | Semantics: External DLL semantics .....                                  | 151        |
| <b>8.32</b>  | <b>Lexical components .....</b>                                          | <b>151</b> |
| 8.32.1       | Syntax (non-production): Character, character set .....                  | 151        |
| 8.32.2       | Definition: Letter, alpha_betic, numeric, alpha_numeric, printable ..... | 151        |
| 8.32.3       | Definition: Break character, break .....                                 | 152        |
| 8.32.4       | Semantics: Break semantics .....                                         | 152        |
| 8.32.5       | Definition: Expected, free comment .....                                 | 152        |
| 8.32.6       | Syntax (non-production): "Blanks or tabs", new line .....                | 152        |
| 8.32.7       | Syntax: : Comments .....                                                 | 153        |
| 8.32.8       | Syntax (non-production): Free Comment rule .....                         | 153        |
| 8.32.9       | Semantics: Header Comment rule .....                                     | 153        |
| 8.32.10      | Definition: Symbol, word .....                                           | 153        |
| 8.32.11      | Syntax (non-production): Break rule .....                                | 153        |
| 8.32.12      | Semantics: Letter Case rule .....                                        | 154        |
| 8.32.13      | Definition: Reserved word, keyword .....                                 | 154        |
| 8.32.14      | Syntax (non-production): Double Reserved Word rule .....                 | 154        |
| 8.32.15      | Definition: Special symbol .....                                         | 154        |
| 8.32.16      | Syntax (non-production): Identifier .....                                | 154        |
| 8.32.17      | Validity: Identifier rule .....                                          | 155        |
| 8.32.18      | Definition: Predefined operator .....                                    | 155        |
| 8.32.19      | Definition: Standard operator .....                                      | 155        |
| 8.32.20      | Definition: Operator symbol .....                                        | 155        |
| 8.32.21      | Definition: Free operator .....                                          | 155        |
| 8.32.22      | Syntax (non-production): Manifest character .....                        | 156        |
| 8.32.23      | Special characters and their codes .....                                 | 156        |
| 8.32.24      | Syntax (non-production): Percent variants .....                          | 157        |
| 8.32.25      | Semantics: Manifest character semantics .....                            | 157        |
| 8.32.26      | Syntax (non-production): String, simple string .....                     | 157        |
| 8.32.27      | Semantics: String semantics .....                                        | 157        |
| 8.32.28      | Syntax: : Integers .....                                                 | 157        |
| 8.32.29      | Validity: Integer rule .....                                             | 158        |
| 8.32.30      | Semantics: Integer semantics .....                                       | 158        |
| 8.32.31      | Syntax (non-production): Real number .....                               | 158        |
| 8.32.32      | Semantics: Real semantics .....                                          | 159        |
| <b>Index</b> | <b>.....</b>                                                             | <b>161</b> |

## Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 25436 was prepared by Ecma International (as ECMA-367) and was adopted, under a special "fast-track procedure", by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

## Introduction

Eiffel was originally designed in 1985 as a method of software construction and a notation to support that method. 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 Clause 3) 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 benefit is 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). Karine Arnout is the recording secretary of TG4.

The final version of the document was prepared by Éric Bezault, Mark Howard, Alexander Kogtenkov, Bertrand Meyer and Emmanuel Stapf.



# Information technology — Eiffel: Analysis, Design and Programming Language

## 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 International 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 Clause 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; and
- 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; and
- other mechanisms for handling Eiffel texts and interacting with users of the language.

The specification of Clause 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**); and
- 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

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

### 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 — ASCII encoded English*.

### 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 direst 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, extendibility 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.

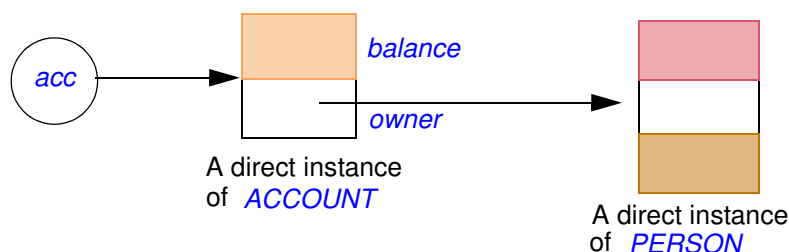


Figure 1: References and objects

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.

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.

```

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

```

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 "["]* 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:

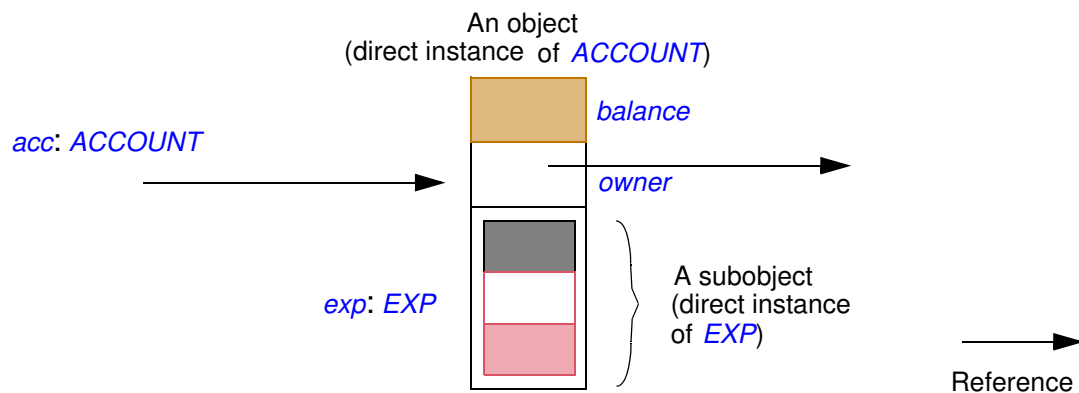


Figure 2: Object and subobject

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:

```

il: 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:

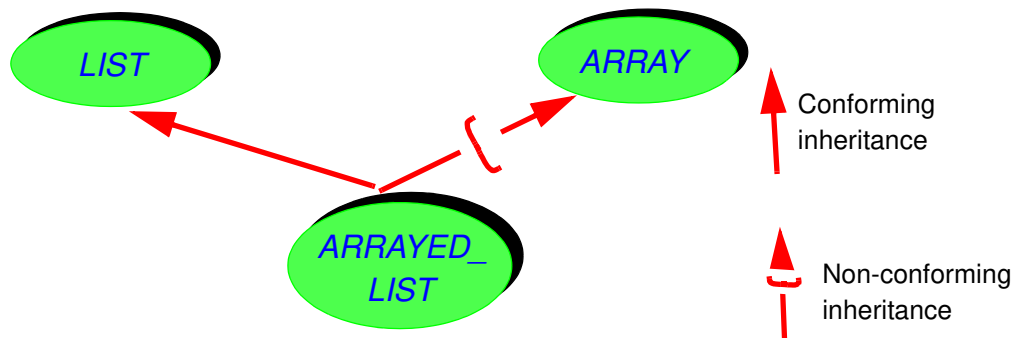


Figure 3: Inheritance

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 polygon 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 **reddeclaration** covers both redefinition and effecting.

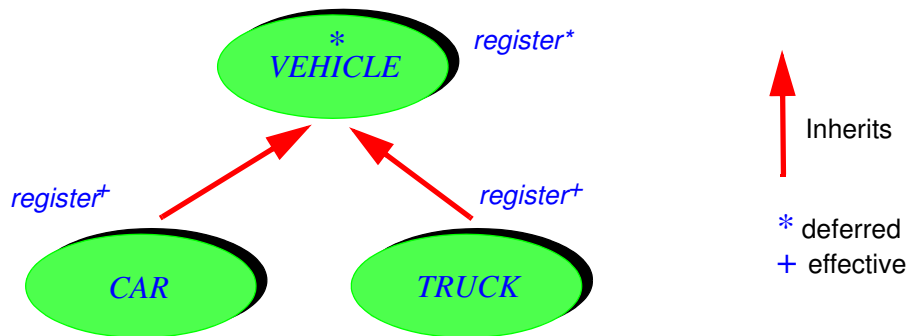


Figure 4: Abstracting variants into a deferred parent

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

$$\{C \ \$ \dots\}^*$$

$$\{C \ \$ \dots\}^+$$

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:

$$C^*$$

$$C^+$$

#### 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**  $\triangleq$  **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 extendibility.

*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: *VS RP*

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 "..."<sup>+</sup>}

**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**.