5 Features

5.1 OVERVIEW

A class is characterized by its features. Every feature describes an operation for accessing or modifying instances of the class.

A feature is either an *attribute*, describing information stored with each instance, or a *routine*, describing an algorithm. Clients of a class *C* may apply *C*’s features to instances of *C* through **call** instructions or expressions.

Every feature has an identifier, which identifies it uniquely in its class. In addition, a feature may have an *alias* to permit calls using operator or bracket syntax.

The following discussion introduces the various categories of feature, explains how to write feature declarations, and describes the form of feature names.

5.2 THE ROLE OF FEATURES

A feature of a class describes an operation which is applicable to the instances of the class. For example:

- A class *SIGNAL* might have such features as *amplitude* (amplitude of a signal) or *modulate* (modulate a signal with another).
- A class *DOCUMENT* might have such features as *character_count* or *print.*
- A class *ELECTRON* might have such features as *spin* or *valence*.
- A class *ABSTRACT_NODE* might have such features as *arity*, *is_leaf*, *is_root*, *add_child* or *remove_child*.

As these examples indicate, the operations represented by features may be of two kinds:

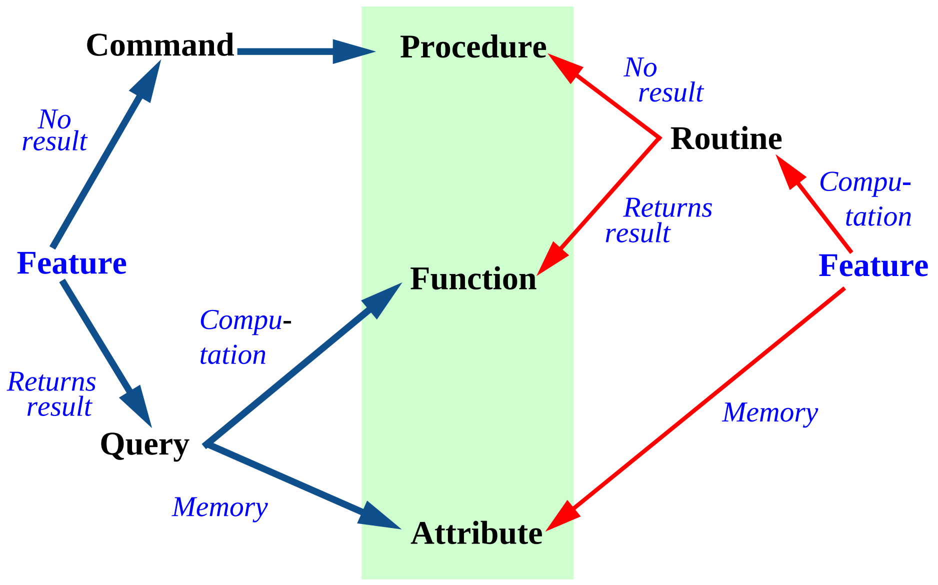
Queries are implemented as attributes or functions, commands as procedures.

- Some are **query** operations, used to find out properties of objects (“What is the amplitude of this signal? How many characters does this document contain? Is this tree node a leaf?”).
- Others are **commands**, used to change objects or apply actions to them (“Print this document! Add a new child to this node!”).

A query will be implemented as an attribute or a routine. By nature, a command will always be a routine.

5.3 FEATURE CATEGORIES

The following diagram shows the variants of the notion of feature and the associated terminology:



From the right, we have a classification based on the implementation of features:

- A feature implemented by storing information in every instance of the class (or, in the case of a constant, common to all instances) is an **attribute**.
- A feature implemented by an algorithm (a computation) applicable to all instances of the class is a **routine**. A routine that returns a result is a **function**; one that doesn't is a **procedure**.

From the left, we have a classification based initially on more abstract properties of features:

- A feature that does not return a result — but may modify its target object — is a **command**. Commands can only be implemented by *procedures* as just defined.
- A feature that provides a result — some information about its target object — is a **query**. A query may be implemented either by storing that information, giving an *attribute*, or by computing that information when requested, giving a *function*.

This book is precise and careful in its use of the terminology. Please make sure (possibly by reading this section once again) that you are familiar with the exact meanings of all the terms: *feature*, *command*, *query*, *routine*, *function* and *attribute*.

The word “method”, sometimes used in the object-oriented literature, may be viewed as a synonym for “routine”, i.e. a feature implemented by an algorithm rather than stored. Although this is a well-accepted term, it is redundant (there were already several words for this notion before O-O came about: routine, subroutine, subprogram…) and leads to confusion with the ordinary sense of the word “method”.

“Feature” is at a higher level, since it covers all categories. The closest word in the C++/UML/Java literature is “member”. Many presentations treat attributes as a data structure implementation mechanism, unrelated to routines; this loses the notion that at the highest level of abstraction we only have a notion of feature, with no commitment to any particular implementation choice. It’s OK to export an attribute (there is no need to encapsulate it in a function!) as long as, to the client, it appears only as a query, with an interface that doesn’t betray whether the query is implemented as an attribute or a function. This is Eiffel’s approach.

5.4 IMMEDIATE AND INHERITED FEATURES

The rest of this chapter will describe the properties of Features parts of a class, which introduces zero or more “features of the class”.

When thinking about features, we must be careful not to confuse two notions:

- The features **introduced in** a class.
- The features **of** that class.

The reason for this distinction is inheritance, which enables a class, in addition to the features declared in its own text, to obtain features declared in other classes — its parents.

The notion of parent is studied in chapters 6, 10 and 16.

Here is the precise terminology.

Inherited, immediate; origin; redeclaration; introduce

Any feature *f* of a class *C* is of one of the following two kinds:

1 • If *C* obtains *f* from one of its parents, *f* is an **inherited** feature of *C*. In this case any declaration of *f* in *C* (adapting the original properties of *f* for *C*) is a **redeclaration**.

2 • If a declaration appearing in *C* applies to a feature that is not inherited, the feature is said to be **immediate** in *C*. Then *C* is the **origin** (short for “class of origin”) of *f*, and is said to **introduce** *f*.

This defines the origin of immediate features only. The full definition, also covering inherited features, appears on page 305.

A feature redeclaration is a declaration that locally changes an inherited feature. The details of redeclaration appear in the study of inheritance; what is important here is that a declaration in the Features part only introduces a new feature (called "immediate" in *C*, or "introduced" by *C*) if it is not a redeclaration of some feature obtained from a parent.

→ Redeclaration is studied in chapter 10, especially 10.28, starting on page 306.

Every feature of a class is immediate either in the class or in one of its proper ancestors (parents, grandparents and so on).

The rest of this chapter only discusses immediate and redeclared features, by describing the Features part of a class declaration.

→ Inherited features are studied in chapters 6, 10 and 16, with the full definition on page 462.

5.5 FEATURES PART: EXAMPLE

A Features part is a sequence of one or more Feature_clause, as in the following sketch of a class from the EiffelBase Library:

```
note
    ... Notes clause omitted ...
class LINKED_LIST [T] inherit
    LIST [T]
        redefine
            first, start, return
        end
feature -- Access
    first: T
            -- Item at first position
        require
            not_empty: not empty
        do
            Result := first_element.item
        end
feature -- Measurement
    count: INTEGER
        -- Number of items in the list
    ... Other feature declarations and Feature_clause omitted ...
feature {LINKED_LIST} -- Implementation
    previous, next: like first_element
    merge_left (other: like Current)
        ... Rest of procedure omitted ...
    ...Other feature declarations omitted...
```

```
feature {NONE} -- Implementation
        first_element: LINKABLE [like first]
                -- First linkable element
        ...Other feature declarations omitted...
invariant
        empty = (first_element = Void)
        ...Other invariant clauses omitted...
end
```

A Features part contains one or more Feature_clause. Each Feature_clause is introduced by the keyword **feature**, which may be followed, as in the last two cases above, by a Clients subclause, which is a list of class names in braces, as in *{A, B, C, …}*.

All the features of a Feature_clause have the same **export status**. If the beginning of the Feature_clause gives a list of clients in braces, the clause's features are available for calls to those clients and their proper descendants only; otherwise they are available to all clients. Here, for example:

→ Chapter 7 explains the details of the export policy and of Clients *clauses.*

- *first* and *count* are available for calls to all clients.
- *previous*, *next* and *merge_left* are available only to *LINKED_LIST* itself, when used as its own client.
- The remaining features are available only to *NONE*; this means that they are secret (accessible within class *LINKED_LIST* only, without use of dot notation).

For a class including many features, you may want to use more than one Feature_clause even for features which all have the same export status. This separates features into feature categories. In this case every Feature_clause should begin (after the keyword **feature** and the Clients list, if any) with a Header_comment indicating the feature category. Here, the comments indicating the various categories are

```
-- Access
-- Measurement
-- Cursor movement
-- Implementation
```

Because the inclusion of such a Header_comment is part of the recommended style, it appears as an optional component in the syntax for Feature_clause given below. Eiffel tools — such as documentation tools, or tools for archiving and retrieving classes — may treat it specially; for eon it for contents. In particular, it appears in the "contract view" serving as the basic documentation for a class.

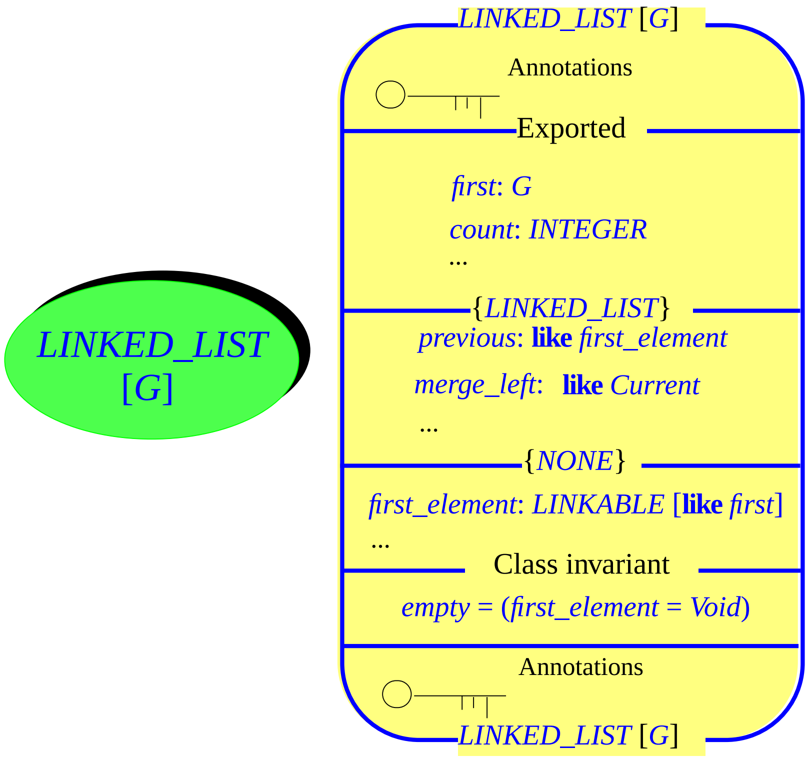
→ "-DOCUMENTING THE CLIENT INTERFACE OF A CLASS", 7.8, page 207 .

Although you may choose any text for header comments, the texts used here — Access and others — are among a dozen or so standard ones used, always in the same order, throughout classes of EiffelBase and other libraries. This yields a consistent style, greatly facilitating software understanding and maintenance. It's a good idea to use such a standard set of headers; start from the one in EiffelBase and extend it if necessary.

5.6 GRAPHICAL REPRESENTATION

In BON (Business Object Notation), the suggested graphical representation for classes and system structures, the features introduced in a class should appear next to the ellipse representing that class.

If enough display space is available and you want a full representation of the features, the format is that of a **feature box** appearing next to the class ellipse, and shown on the next figure for part of the class sketched in the previous section.

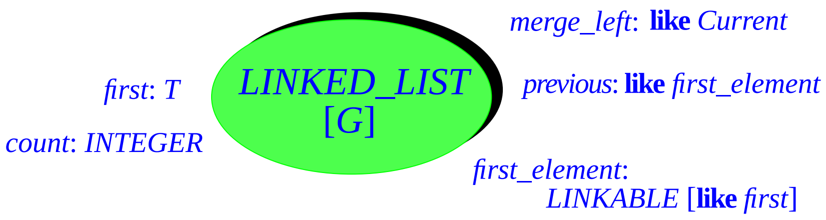


A class and its feature box

As in the textual form of the class, the features are grouped into successive divisions according to their export status.

Each feature includes type information as needed: argument types for routines; for a query (attribute or function), result type.

This graphical notation takes up a large amount of space and is mostly suitable for examining and designing classes in an interactive graphical environment, where you can see the various properties displayed on demand: the ellipses representing classes, the arrows representing the client and inheritance relations, the feature boxes. For printing on paper, or a whiteboard discussion, a more concise representation — frequently used in the present book — is appropriate:


merge_left: like Current
first: T
LINKED_LIST [G]
previous: like first_element
count: INTEGER
first_element: LINKABLE [like first]

A class with some features

An additional convention will be seen in the discussion of attributes: if you know that a feature is an attribute, you may highlight this property by enclosing the feature's name in a rectangle.

→ *Page 491* .

5.7 FEATURES PART: SYNTAX

Here is the precise format of the Features part of a class text, illustrated by the above example.

Feature parts

Features ≜ Feature_clause$^+$

Feature_clause ≜ **feature** [Clients] [Header_comment] Feature_declaration_list

Feature_declaration_list ≜ {Feature_declaration "**;**" …}*

Header_comment ≜ Comment

→ *The syntax for the* Clients *part appears on page 204*.

As part of a general syntactical convention, semicolons are **optional** between a Feature_declaration and the next. The recommended style rule suggests omitting them except in the infrequent case of two successive declarations on a single line.

← *"Syntax (non-production): Semicolon Optionality rule", page 102*

The rest of this chapter concentrates on the Feature_declaration construct, explaining what kinds of feature a class may declare.

5.8 FORMS OF FEATURE

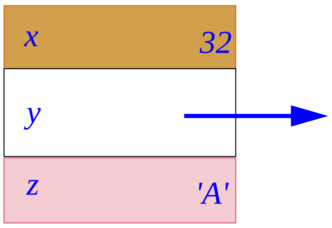
Feature categories: overview

Every feature of a class is either an *attribute* or a *routine*.
An attribute is either *constant* or *variable*.
A routine is either a *procedure* or a *function*.

A set of definitions in the discussion that follows introduces each of these notions precisely, making it possible to recognize, from a valid feature declaration, which kind of feature it introduces.

By introducing an *attribute* in a class, you specify that at run-time every instance of the class will possess a certain value, or field, corresponding to the attribute.

So you may picture any instance of the class as an object made of a number of fields, each giving the value defined by the object for one of the attributes of the class. The figure illustrates a direct instance of a class *C* with three attributes, *x*, *y* and *z*. (To picture a non-direct instance, we would also need to consider attributes introduced in proper descendants.)



A class instance with its fields

x is of type INTEGER, z of type CHARACTER, and y of some reference type. The field for y is attached to an object, which the figure does not show.

An attribute is either **variable** or **constant**:

- If an attribute is variable, the corresponding field may be different for various instances of the class and may change at run-time. As a consequence, the actual values must be stored for each instance.
- If an attribute is constant, the corresponding field is the same value for all instances, and may not change at run-time. This value appears in the class as part of the attribute declaration.

By introducing a *routine* in a class, you specify that a certain computation (an algorithm) must be applicable to every instance of the class. A routine, as we have seen, is either a **procedure** or a **function**:

- A procedure does not return a result; it may perform a number of operations, which may modify the instance.
- A function returns a result and may also perform operations.

A function should not change any object, except if the change only affects an object's representation, not its abstract properties. Because language processing tools cannot easily know which properties are abstract, the ban on object-modifying functions is a methodological guideline — the *Command-Query Separation principle* — and not a language rule.

See "“Object-Oriented Software Construction”.

5.9 FEATURE DECLARATIONS: EXAMPLES

To help you become familiar with the syntax of a Feature_declaration, here are a few artificial examples illustrating the various possibilities. The next sections give the precise syntax and detailed comments; for the most part, however, the examples should suffice as a guide for declaring features. The name of each example feature (such as *function_without_arguments*) suggests its nature.

```
variable_attribute: INTEGER
            -- Some field of integer type
other_variable_attribute: SOME_TYPE
            -- Some other field, of another type
Constant_attribute: REAL = 3.141592
            -- A constant real value used by the class
procedure (argument1: INTEGER; argument2: SOME_TYPE)
            -- (Here should come the description
            -- of the procedure's intended effect.)
      do
            some_attribute.some_procedure
            other_attribute.other_procedure
      end
deferred_procedure (argument1: SOME_TYPE )
            -- (Here should come the description
            -- of the procedure's intended effect.)
      deferred
      end
function_with_arguments (arg1, arg2: SOME_TYPE): OTHER_TYPE
            -- (Here should come the description
            -- of the result computed by the function.)
      do
            create Result
            Result.some_procedure (arg2)
      end
```

```
function_without_arguments: INTEGER
        -- (Here should come the description
        -- of the result computed by the function.)
    do
        Result := some_value
    end
plus alias "+" (some_matrix: like Current): like Current
        -- Matrix sum of Current and some_matrix
    do
        ...(Computation of the sum into Result)...
    end
attribute_with_contract: SOME_TYPE
        -- (Here should come the description of its role.)
    require
        some_property
    attribute
    ensure
        other_property
    end
self_initializing_attribute: SOME_TYPE
        -- (Here should come the description of its role.)
    attribute
        initialization_instructions
    end
```

5.10 FEATURE DECLARATIONS: SYNTAX

With the preceding examples in mind, we can now look at the exact ingredients that make up a Feature_declaration_list.

Such a list introduces immediate features of a class. It is a sequence of individual Feature_declaration clauses. In general each Feature_declaration introduces one feature, although it is possible to use a single declaration to introduce two or more "synonym" features. Each Feature_declaration includes the following pieces of information:

- The feature’s original name (or names in the case of synonyms).
- The type of the feature, if it is an attribute or a function.
- The formal arguments, if the feature is a routine (procedure or function) with arguments.
- The actual value of the feature if it is a constant attribute.

- The contract and computation associated with the feature if applicable; a routine in particular must have an associated algorithm, but an attribute may also have a precondition and postcondition, as with *attribute_with_contract*, and a self-initialization algorithm, as with *self_initializing_attribute*.

→ *“OBSOLETE FEATURES”, 5.21, page 163*.

- Possibly an Obsolete clause for a routine whose use is no longer recommended.

- Possibly the keyword **frozen**, appearing before the feature name to express that the declaration is final (not subject to redefinition in descendants).

→ *More on frozen declarations in discussion of the Feature Declaration rule page 160 below (condition 4), and of the Redefine Subclause rule, page 301 (condition 2).*

The precise syntax is:

Feature declarations

```
  Feature_declaration ≜ New_feature_list Declaration_body
     Declaration_body ≜ [Formal_arguments] [Query_mark]
                        [Feature_value]
           Query_mark ≜ Type_mark [Assigner_mark]
            Type_mark ≜ ":" Type
        Assigner_mark ≜ assign Feature_name
        Feature_value ≜ [Explicit_value]
                        [Obsolete]
                        [Header_comment]
                        [Attribute_or_routine]
       Explicit_value ≜ "=" Manifest_constant
```

Not all combinations of Formal_arguments, Query_mark and Feature_value are possible; the Feature Body rule and Feature Declaration rule will give the exact constraints. For example it appears from the above syntax that both a Declaration_body and a Feature_value can be empty, since their right-side components are all optional, but the validity constraints rule this out.

→ *Pages 144 and 160.*

The above examples illustrate some of the most important valid combinations.

→ *The form of an Extended_feature_name and the rules on multiple feature declarations appear later in this chapter (5.18).*

What appears before the Declaration_body is not just a feature name but a New_feature_list, with the syntax

New feature lists

New_feature_list ≜ {New_feature "," …}$^+$

New_feature ≜ [**frozen**] Extended_feature_name

where an Extended_feature_name is a feature identifier possibly complemented by an Alias (for operator features).

Having a list of features, rather than just one, makes it possible for example to declare together several attributes of the same type or, in the case of routines, to introduce several "synonym" routines, with the same body.

A Formal_arguments part, possible only for a routine, describes the arguments to a routine and their types. An example is

→ *The syntax of* Formal_arguments *appears in 8.3, page 215.*

```
(arg1, arg2: TYPE1; arg3: TYPE2; arg4, arg5, arg6: TYPE3)
```

A Query_mark is present to mark that the feature is a query (attribute or function). It has a Type_mark specifying the type of the information returned by the query: for an attribute that's the type of the field in instances of the class, for a function, it's the type of the result computed by an execution of the function.. Examples of Type_mark are

```
: INTEGER
: SOME_TYPE
```

A Query_mark may also include an optional Assigner_mark. This lets you associate with the query a command of the class (a procedure), which can then be used to change the value of the query for the target object. A typical Assigner_mark is:

```
assign put
```

This may appear in a declaration of a function *item*: *T* for some type *T*, as in

```
item: T assign put …
```

where *put* is a procedure of the same class, taking an argument of type *T*. This allows clients to use assignment syntax, *x*.*item* := *a* (for *a* of type *T*), as an abbreviation for the feature call *x*.*put* (*a*). The mechanism also works for queries with arguments, as in *your_array*.*item* (*i*) := 5 for a feature *item* taking an integer argument, as it does in class *ARRAY* [*G*], where *item* has an integer argument; the associated assigner procedure correspondingly takes two two arguments, of types *G* and *INTEGER*. (Thanks to bracket syntax, you may also write this last example as *your_array* [*i*] := 5.)

→ *"BRACKET FEATURE", 5.17, page 157.*

The procedure which an Assigner_mark associates with a query, such as *put* in these examples, is called an *assigner procedure*. The assignment-like instructions which this makes possible, such as *x*.*item* := *a* — with assignment-like syntax but the semantics of a call — is an *assigner call*.

→ *"ASSIGNER PROCEDURES", 5.16, page 155.*

The Feature_value part, if present, gives the "value" of the feature, required in two cases:

- For a constant attribute,it introduces a literal value (integer, string etc.) with by an "equal: sign, as in *One*: *INTEGER* = 1.
- For a routine, it introduces the routine text.

For an attribute you can use a full form similar to that of routines, as in *x*: *A* ... **attribute** ... **end**, but for the most common case there's an abbreviated form of the declaration: just *x*: *A*.

In the above example, the Feature_value for *constant_attribute* defines the constant's value to be the real number 3.141592; the Feature_value for *procedure* is

```
          -- (Here should come the description
          -- of the procedure's intended effect.)
     do
          some_attribute.some_procedure
          other_attribute.other_procedure
     end
```

A Feature_value may, according to the syntax, introduce some or all of the following components (the validity rules define which combinations are possible):

- An Explicit_value to specify the value of a constant attribute.
- An Obsolete mark to signal that the feature is obsolete.
- A Header_comment to explain the purpose of the feature.
- An Attribute_or_routine part to give the detailed specification of a routine or attribute, with clauses such as a precondition, a postcondition or, for a routine, the associated algorithm, as detailed next.

→ *See 29.2, page 777 and subsequent sections about constants.*

→ *"OBSOLETE FEATURES", 5.21, page 163.*

5.11 FEATURE BODIES

Here indeed is the syntax of Attribute_or_routine:

Subsequent chapters detail various elements of an Attribute_or_routine:

- A Precondition and Postcondition express the contract of a feature.
- Local_declarations introduce local variables needed by the feature's algorithm if any.

→ *Chapter 9.*
→ *"LOCAL VARIABLES AND RESULT", 8.6, page 221.*

Feature bodies

```
Attribute_or_routine ≜ [Precondition]
                       [Local_declarations]
                       Feature_body
                       [Postcondition]
                       [Rescue]
                       end
        Feature_body ≜ Deferred | Effective_routine | Attribute
```

- A Feature_body gives details of its implementation as an Effective_routine with an associated algorithm, or an attribute, or states that it is deferred routine, implemented only in proper descendants.

→ Chapter 8.

- A Rescue clause takes over if a run-time exception arises during the execution of the feature.

→ Chapter 26.

Only some combinations of these various clauses are meaningful. It is convenient to state the corresponding validity rule at the level of a Feature_value as a whole rather than just Attribute_or_routine:

Feature Body rule *VFFB*

A Feature_value is valid if and only if it satisfies one of the following conditions:

1 • It has an Explicit_value and no Attribute_or_routine.

2 • It has an Attribute_or_routine with a Feature_body of the Attribute kind.

3 • It has no Explicit_value and has an Attribute_or_routine with a Feature_body of the Effective_routine kind, itself of the Internal kind (beginning with **do** or **once**).

4 • It has no Explicit_value and has an Attribute_or_routine with neither a Local_declarations nor a Rescue part, and a Feature_body that is either Deferred or an Effective_routine of the External kind.

The variants of Feature_body appear on page 218 as part of the discussion of routines.

The Explicit_value only makes sense for an attribute — either declared explicitly with Attribute or simply given a type and a value — so cases 3 and 4 exclude this possibility.

The Local_declarations and Rescue parts only make sense (case 4) for a feature with an associated algorithm present in the class text itself; this means a routine that is neither deferred nor external, or an attribute with explicit initialization.

In both cases 1 and 2 the feature will be an attribute. Case 1 is the implicit form where we don't take the trouble to write the keyword **attribute**, writing for example just *your_attribute*: *SOME_TYPE*. Case 2 is the long form, explicitly using the keyword **attribute** and making it possible, as with routines, to have a Precondition, a Postcondition, and even an implementation (including a Rescue clause if desired) which will be used, for "self-initializing" types, on first use of an uninitialized field.

The Feature Body rule is the basic validity condition on feature declarations. But for a full view of the constraints we must take into account a set of definitions appearing next, which say what it takes for a feature declaration — already satisfying the Feature Body rule — to belong to one of the relevant categories: *variable attribute*, *constant attribute*, *function*, *procedure*. Another fundamental constraint, the Feature Declaration rule (*VFFD*), will then require that the feature described by any declaration match one of these categories. So the definitions below are a little more than definitions: they collectively yield a validity requirement complementing the Feature Body rule.

→ *Page 160*.

5.12 HOW TO RECOGNIZE FEATURES

The precise form and properties of attributes and routines, as described by the syntax given above for Feature_declaration, are studied in later chapters. You should, however, learn right away how to recognize attributes (constant or variable) and routines (procedures or functions). This is not difficult and the above examples illustrate the most common cases. First, variable attributes:

Variable attribute

A Feature_declaration is a **variable attribute** declaration if and only if it satisfies the following conditions:

1 • There is no Formal_arguments part.

2 • There is a Query_mark part.

3 • There is no Explicit_value part.

4 • If there is a Feature_value part, it has an Attribute_or_routine with a Feature_body of the Attribute kind.

The first two features in the earlier example, *variable_attribute* and *other_variable_attribute*, were in this category. Here is an extract from a Feature_clause with two declarations introducing three variable attributes:

← *Page 139*.

```
count, capacity: INTEGER
backup: LINKED_LIST [INVESTMENT]
```

These examples use the abbreviated form without a keyword. You would obtain the same semantics by specifying **attribute** explicitly:

And similarly for backup.

```
count, capacity: INTEGER
        attribute
        end
```

The keyword is required if you need to include a precondition, a postcondition, or a for self-initialization algorithm, as in *attribute_with_contracts* and *self_initializing_attribute* above, or

```
bounding_rectangle: RECTANGLE
                -- Smallest rectangle including this figure
        require
                non_empty: not empty
        attribute
                create Result.make (lower, leftmost, height, width)
        ensure
                Result.lower = lower ; Result.higher = higher
                Result.leftmost = leftmost ; Result.rightmost = rightmost
                Result.height = height ; Result.width = width
        end
```

The next kind of feature is the constant attribute:

Constant attribute

A Feature_declaration is a **constant attribute** declaration if and only if it satisfies the following conditions:

1 • It has no Formal_arguments part.

2 • It has a Query_mark part.

3 • There is a Feature_value part including an Explicit_value.

Two examples, introducing an Integer_constant and a Manifest_string are:

```
maximum_discount: INTEGER = 25
message: STRING = "No such site"
```

Even though the value is known from the declaration, it may still be useful to associate a contract (precondition, postcondition or both) to emphasize its more fundamental properties, which presumably would survive a change of the value in a revision of the software:

```
message: STRING = "No such site"
    ensure
        Result /= Void
        Result.count <= Max_message_length
```

Finally, the case of a routine, with two variants:

Routine, function, procedure

A Feature_declaration is a **routine** declaration if and only if it satisfies the following condition:

- There is a Feature_value including an Attribute_or_routine, whose Feature_body is of the Deferred or Effective_routine kind.

If a Query_mark is present, the routine is a **function**; otherwise it is a **procedure**.

For a routine the Formal_arguments (like the Query_mark) may or may not be present.

By convention this definition treats a deferred feature as a routine, even though its effectings in proper descendants may be, in the case of a query, attributes as well as functions.

In the example illustrating the various categories of feature, the features with the following names are routines:

← *Page 139 on.*

```
procedure
deferred_procedure
function_with_arguments
function_without_arguments
plus alias "+"
```

plus, with its infix alias "+", is a function; the others are procedures or functions as indicated by their names.

Why do we need such rules for recognizing various kinds of feature? To put it more critically, why doesn't the language distinguish more clearly between them — for example by requiring specific keywords such as **procedure** at the beginning of each declaration?

The reason is methodological. As seen by clients, a feature is an abstract property of the instances of the class. Its particular choice of implementation within the class is a subordinate concern. As a consequence, the syntax downplays the differences between these forms of features instead of emphasizing them.

More precisely, what matters for clients is whether a feature returns a result, or just affects the state of objects without directly producing a result. This distinction is reflected in the following notions:

DEFINITION

Command, query

A **command** is a procedure. A **query** is an attribute or function.

These notions underlie two important principles of the Eiffel method:

- The Command-Query separation principle, which suggests that queries should not change objects.
- The Uniform Access principle, which enjoins, whenever possible, to make no distinction between attributes and argumentless functions.

See the discussion of these principles in "Object-Oriented Software Construction" and "UNIFORM ACCESS", 23.4, page 616 in the present book. These ideas also lead to the notion of contract view, studied in 7.8, page 207.

Of course, it is sometimes necessary to check what category a feature really belongs to. As the above examples indicate, you should quickly become familiar with the various forms.

5.13 THE SIGNATURE OF A FEATURE

PURPOSE

As defined above, a query feature — attribute or function — has a result type. But for any feature, query or command, we often need a more complete characterization of the feature's type properties, involving both the type of its result (in the query case) and the number and type of its arguments if any.

The notion of **signature** provides this. The signature of a feature is made of two lists of types:

- The list of argument types (empty for an attribute, or a routine without arguments).
- The list of result types (empty for a procedure).

To represent lists of types, we can borrow Eiffel's tuple notation, as in *[TYPE1, TYPE2, TYPE3]*. (Such a list is not an Eiffel component, simply a notation to talk about properties of Eiffel features.) So a function whose declaration begins with

Chapter 13 discusses tuples.

```
f (a: INTEGER; b: X): LINKED_LIST [STOCK] is...
```

has the signature

```
[INTEGER, X], [LINKED_LIST [STOCK]]
```

We do not really need a sequence for the second component of a signature, since it will have zero or one element (zero ifor a procedure, one for a query). Using a list on both sides provides more symmetry and generality. (The discussion of tuples will illustrate that symmetry.)

→ "Emulating multiple results", page 371

Here are the signatures of the features in the example at the beginning of this chapter. An empty sequence is shown as [].

variable_attribute:	[ ], [*INTEGER*]
other_variable_attribute:	[ ], [*SOME_TYPE*]
constant_attribute:	[ ], [*REAL*]
procedure:	[*INTEGER*, *SOME_TYPE*], []
deferred_procedure:	[*SOME_TYPE*], []
function_with_arguments:	[*SOME_TYPE*, *SOME_TYPE*], [*OTHER_TYPE*]
function_without_arguments:	[ ], [*INTEGER*]
plus **alias** "+":	[**like** *Current*], [**like** *Current*]
attribute_with_contract:	[ ], [*SOME_TYPE*]
self_initializing_attribute:	[ ], [*SOME_TYPE*]

The signatures of *variable_attribute* and *function_without_arguments* are identical, even though one is an attribute and the other a function. For clients, as noted, the difference won't be visible.

See above, page 147, and "UNIFORMACCESS", 23.4, page 616

The notion of signature deserves a precise definition:

Signature, argument signature of a feature

The **signature** of a feature *f* is a pair *argument_types*, *result_type* where *argument_types* and *result_type* are the following sequences of types:

- For *argument_types*: if *f* is a routine, the possibly empty sequence of its formal argument types, in the order of the arguments; if *f* is an attribute, an empty sequence.
- For *result_type*: if *f* is a query, a one-element sequence, whose element is the type of *f*; if *f* is a procedure, an empty sequence.

The *argument_types* part is the feature's **argument signature**.

The argument signature is an empty sequence for attributes and for routines without arguments.

In the above examples, the argument signature of *variable_attribute* is [ ] (empty sequence); the argument signature of *procedure* is [*INTEGER*, *SOME_TYPE*].

5.14 FEATURE NAME

Feature names serve to identify features.

A feature name always involves an identifier; this means that it is always possible to write a valid call in ordinary object-oriented dot notation, as in

→ As detailed in the chapter on calls: chapter 23.

```
x.f (a)            -- Qualified: from a client, applied to x

f (a)              -- Unqualified: from within the class,
                   -- applied to the current object
```

For some features syntactic variants are available, but the availability of these basic forms is guaranteed:

Feature principle

Every feature has an associated identifier.

Any valid call (qualified or unqualified) to the feature can be expressed through this identifier.

The syntactic variants, available through **alias** clauses, offer other ways to express calls, reconciling object-oriented structure with earlier notations:

- You may qualify the name with **alias** "§" where § is some operator. For example if a feature is named *plus*, clients must call it as *a*.*plus* (*b*); by naming it *plus* **alias** "+" you still allow this form of calls — per the Feature principle — but you also permit *a* + *b* in accordance with traditional syntax for arithmetic expressions. The details of alias operators, as well as the associated conversion mechanism, appear next.
- You may also use a "bracket alias", written simply **alias** "[ ]" (with an opening bracket immediately followed by a closing bracket). This allows access through bracket syntax *x* [*index*], For example if a class describing some table structure has a feature *item* **alias** "[ ]" (*index*: *H*): *G* where *H* is some index type, items can be accessed through *your_table*.*item* (*i*) but also through the more concise *your_table* [*i*]. Again this is just a syntactic facility: the second form is a synonym for the first, which remains available.

  Such bracket aliases appear for example in the *ARRAY*, *LIST* and *HASH_TABLE* classes of EiffelBase, so that you can access an element of one of these structures as *t* [*i*], where *i* is an integer in the first two cases and a key in the last one.

You may combine such **alias** clauses in the extended feature name with assigner procedures. For example the EiffelBase class *HASH_TABLE* defines the access feature

Introduced in

> *item* **alias** "[ ]" (*key*: *H*): *G* **assign** *put* …

where *put* is a procedure to insert an element with a given key. This means that you can not only use bracket syntax for accessing items, as in

> *your_element* := *your_hash_table* [*your_key*]

(an abbreviation for *your_element* := *your_hash_table*.*item* (*your_key*)), but also in an assigner call

> *your_hash_table* [*your_key*] := *your_element*

an abbreviation for *your_hash_table*.*put* (*your_element*, *your_key*).

We have now seen all kinds of feature name. Here is the syntax:

Feature names

Extended_feature_name ≜ Feature_name [Alias]

Feature_name ≜ Identifier

Alias ≜ **alias** '"' Alias_name '"' [**convert**]

Alias_name ≜ Operator | Bracket

Bracket ≜ "[ ]"

The optional **convert** mark, for an operator feature, supports mixed-type expressions causing a conversion of the target, as in the expression *your_integer* + *your_real*, which should use the "+" operation from *REAL*, not *INTEGER*, for compatibility with ordinary arithmetic practice. See the presentation of conversions.

→ "MIXED-TYPE EXPRESSIONS: TARGET CONVERSION", 15.12, page 419.

For the record we must clarify the use of quotes and spaces in an Alias:

→ *Similar rules apply to signed constants, in "Syntax (non-production): Sign Syntax rule", page 778, and, further in that chapter, to character and string constants.*

Syntax (non-production): Alias Syntax rule

The Alias_name of an Alias must immediately follow and precede the enclosing double quote symbols, with no intervening characters (in particular no breaks).

When appearing in such an Alias_name, the two-word operators **and then** and **or else** must be written with one or more spaces (but no other characters) between the two words.

In general, breaks or comment lines may appear between components prescribed by a BNF-E production, making this rule necessary to complement the grammar: you must write **alias** "+", not **alias** " + ".

It is useful to give official names to the aliased forms:

Operator feature, bracket feature, identifier-only

A feature is an **operator feature** if its Extended_feature_name *fn* includes an Operator alias, a **bracket feature** if *fn* includes a Bracket alias. It is **identifier-only** if neither of these cases applies.

The most common case is identifier-only. The other two kinds provide convenient modes of expression ("*syntactic sugar*") for some cases where a shorter form, compatible with traditional mathematical conventions, is desirable for **calling** the feature.

→ Calls are studied in chapter 23. See also chapter 28 about expressions.

When referring to feature names, some syntax rules use the Extended_feature_name, and some use the Feature_name, which is just the identifier, dropping the Alias if any. The criterion is simple: when a class text needs to refer to one of its own features, the Feature_name is sufficient since (from the Feature Identifier principle below) it uniquely identifies the feature. So the Extended_feature_name is used in only two cases: when you first introduce a feature, in a Feature_declaration as discussed above, and when you change its name for a descendant, in a Rename clause (for both inheritance and constrained genericity).

→ "RENAMING", 6.9, page 180.

This also means that in descendants of its original class a feature will retain its Alias, if any, unless a descendant explicitly renames it to a name that may drop the Alias, or provide a new one. In particular, redeclaring a feature does not affect its Alias.

There are indeed three forms of call:

- For the standard case, identifier features, calls use **dot notation**, as in

```
a.variable_attribute
b.procedure (b, c)
a.plus (b)
your_array.item (some_index)
```

- Giving a feature an operator alias, such as *plus* **alias** "+", allows calls to take the form of ordinary arithmetic expressions, such as *a* + *b*, rather than the more "obviously O-O" but heavier *a*.*plus* (*b*).

→ Studied in more detail in the next section.

- A class may have one (and only one) feature with a bracket alias, such as *item* **alias** "[ ]". The purpose, for a class representing container data structures such as arrays or tables, is to let clients access the structures using the traditional syntax of array and function access, for example *your_array* [*x*] as a synonym for *your_array*.*item* (*some_index*).

→ Studied in more detail below.

Remember that the operator and bracket aliases are only there to allow a form of feature call with a syntax other than dot notation, conforming to widely accepted notations (operator expressions, bracket access for arrays). Per the Feature principle, every feature has a Feature_name (which you can use to call the feature, although most people find the operator or bracket form clearer when available). If we need to clarify, we talk of "the identifier" of the feature:

← Page 150.

Identifier of a feature name

The Identifier that starts a Extended_feature_name is called the **identifier of** that Extended_feature_name and, by extension, of the associated feature.

This notion is closely related to one of the language's design principles:

Feature Identifier principle

Given a class *C* and an identifier *f*, *C* contains at most one feature of identifier *f*.

This principle reflects a critical property of object-oriented programming in general and Eiffel in particular: no "*overloading*" of feature names within a class. It is marked as "validity" but has no code of its own since it is just a consequence of other validity rules.

← Discussed at the end of this chapter: "NO IN-CLASS OVERLOAD-ING", 5.22, page 166.

Another general notion that we need to define for feature name is when two feature names or operators are "the same". The definition ensures that we ignore letter case:

Same feature name, same operator, same alias

Two feature names are considered to be "**the same feature name**" if and only if their identifiers have identical lower names.

Two operators are "**the same operator**" if they have identical lower names.

An Alias in an Extended_feature_name is "**the same alias**" as another if and only if they satisfy the following conditions:

- They are either the same Operator or both Bracket.
- If either has a **convert** mark, so does the other.

← The lower name is (page 102) the name all in lower case.

So *my_name*, *MY_NAME* and *mY_nAMe* are considered to be the same feature name. The recommended style uses a name with an initial capital and the rest in lower case (as in *My_name*) for constant attributes, and the lower name, all in lower case (as in *my_name*) for all other features. If letters appear in operator feature names, letter case is also irrelevant when it comes to deciding which feature names are the same and which different.

This notion is useful in particular to enforce the rule that, in any class, there can be only one feature of a given name (no “overloading”), and to determine what constitutes a “*name clash*” under multiple inheritance. In such cases the language rules simply ignore letter case.

→ *“NO IN-CLASS OVERLOADING”, 5.22, page 166*.

5.15 OPERATOR FEATURES

Operator features — those declared with an Alias listing a binary or unary operator — allow class authors, as previewed above, to provide their clients with a form of call based on the time-honored conventions of arithmetic expressions, using infix and prefix operators.

A matrix class can use *plus* **alias** "+" as the name of an addition function, enabling users of this feature to write additions in the usual mathematical form

```
matrix1 + matrix2
```

rather than in dot notation, which in this case might come out as *matrix1*.*plus* (*matrix2*).

Similarly, naming a negation function *negated* **alias** "–" allows calls written in the form – *matrix1* as well as *matrix1*.*negated*.

The following syntax shows that an operator is either a free operator (Free_unary or Free_binary) or a **standard operator**. The standard operators, listed explicitly below, use special symbols, except for boolean operators which, following tradition, use keywords, simple (as **and**) or double (as **and then**).

Operators

```
Operator ≜ Unary | Binary
   Unary ≜ not | "+" | "–" | Free_unary
  Binary ≜ "+" | "–" | "*" | "/" | "//" | "\\" | "^" | ".." |
           "<" | ">" | "<=" | ">=" |
           and | or | xor | and then | or else | implies |
           Free_binary
```

Free operators enable developers to define their own operators with considerable latitude. This is particularly useful in scientific applications where it is common to define special notations, which Eiffel will render as unary or infix operators. You may for example define operators such as **, |–| (maybe as an infix alias for a *distance* function), or various forms of arrow such as <–>, –|–>, =>.

The rule, given formally in the lexical analysis chapter, lets you use the symbols appearing in standard operators and any others non-alphabetic symbols as long as the result does not create any ambiguity with standard operators, special symbols, and predefined operators used for equality and inequality (=, /=, ~, /~).

→ *“OPERATORS”, 32.13, page 882*

To avoid spurious parentheses in the writing of expressions, each of the standard operators, Unary and Binary, has a precedence level, according to a table appearing in the discussion of expressions. All free operators have the same precedence, higher than for standardoperators.

→ *“Operator precedence levels”, page 759*.

Remember that an operator is not by itself a feature name but only appears in the **alias** of an Extended_feature_name, which also lists an identifier. This means that you never *have* to use operator notation to call a feature, as in *matrix1* + *matrix2*: dot notation, using the feature's identifier as in *matrix1*.*plus* (*matrix2*), is always available, with the same semantics.

5.16 ASSIGNER PROCEDURES

In an assignment *x* := *v* the target *x* must be a variable. If *item* is an attribute of the type *T* of *a*, programmers used to other languages may be tempted to write an assignment such as *a*.*item* := *v* to assign directly to the corresponding object field, but this is not permitted as it goes against all the rules of data abstraction and object technology. The normal mechanism is for the author of the base class of *T* to provide a "setter" command (procedure), say *put*, enabling the clients to use *a*.*put* (*v*).

The class author may, for convenience, permit *a*.*item* := *v* as a shorthand for this call *a*.*put* (*v*), by specifying *put* as an **assigner command** associated with *item*. An instruction such as *a*.*item* := *v* is not an assignment, but simply a different notation for a procedure call; it is known as an **assigner call**. This scheme, a notational simplification only, is also convenient for features that have a Bracket alias, allowing for example, with *a* an array, an assigner call *a* [*i*] := *v* as shorthand for a call *a*.*put* (*v*, *i*).

The mechanism is applicable not just to attributes but (in line with the Uniform Access principle) to all queries, including functions with arguments.

The following rules defines under what conditions you may, as author of a class, permit such assigner calls from your clients by associating an assigner command with a query.

VALIDITY

Assigner Command rule VFAC

An Assigner_mark appearing in the declaration of a query *q* with *n* arguments (*n* ≥ 0) and listing a Feature_name *fn*, called the **assigner command** for *q*, is valid if and only if it satisfies the following conditions:

1 • *fn* is the identifier of a command *c* of the class.

2 • *c* has *n* + 1 arguments.

3 • The type of *c*'s first argument is the result type of *q*.

4 • For every *i* in 1..*n*, the type of the *i*+1-st argument of *c* is the type of the *i*-th argument of *q*.

PURPOSE

The feature *q* can only be a query since, from the syntax of Declaration_body, an Assigner_mark can only appear as part of a Query_mark, whose presence makes the feature a query.

In cases 3 and 4, we require the types to be identical, not just compatible (converting or conforming). To understand why, recall that the assignment *a.item* := *y* is only a shorthand for a call *a.put* (*x*) with, as a typical implementation:

```
item: T assign put do … end
put (b: U) do … item := b … end
```

WARNING: not valid for different T and U; see text.

Now assume that *U* is not identical to *T* but only compatible with it, and consider the procedure call

```
a.put (a.item)
```

or the equivalent assignment form

```
a.item := a.item
```

which are in principle useless — they reassign to a field its own value — but should certainly permitted. They become invalid, however, because the source *a.item* (actual argument of the call or right side of the assignment) is of type *T*, the target (the formal argument) of type *U*, and it's generally impossible for two different types to be each compatible with the other.

In the conformance case, two-way compatibility would mean that the base classes are proper descendants of each other, causing an inheritance cycle.

→ Prohibited by clause 1 of the Parent rule, page 176.

In the convertibility case, it would violate the Conversion Asymmetry principle.

→ Page 400.

It is in fact possible to have conformance one way and convertibility the other, but this case is not useful enough to justify a special rule.

This explains clause 3: the first argument of the assigner procedure must have *exactly* the same type as the result of the query. Similar reasoning applied to other arguments (if any) leads to clause 4.

5.17 BRACKET FEATURE

Besides operator aliases, the syntax of Alias offers the Bracket variant, allowing you for example to declare, in a class *HASH_TABLE* [*G*, *H*] describing tables of elements of type *G* with keys of type *H*, a feature

← Bracket, *page* 151.

```
item alias "[]" (key: K): G
        -- Item having the given key
    require
        present: has (key)
    do
        … "Appropriate implementation" …
    ensure
        …
    end
```

This is a normal feature, here a function, distinguished only by a new form of call. Although you may still use the standard dot-notation form

```
your_table.item ("ABC")
```

the bracket alias allows another phrasing for exactly the same semantics:

```
your_table ["ABC"]
```

To avoid ambiguity, at most one feature of a class may have a bracket alias; it must be a function with at least one argument. These requirements appear in the general constraint on aliases: the Alias validity rule.

→ *Page* 162.

If the function has more than one argument, the bracket notation will use commas, as in *matrix3* [*i*, *j*, *k*].

It is often convenient, as already noted, to use the assigner procedure mechanism in connection with a bracket alias. If the declaration of *item* reads

```
item alias "[]"  (key: K): G assign put
        … The rest as above …
```

referring to a procedure *put* of the same class, with compatible signature:

```
put (value: G; key: K) … end
```

then instead of

```
your_table.put (v, "ABC")
```

you may write, with identical semantics:

```
your_table ["ABC"] := v
```

5.18 SYNONYMS AND MULTIPLE DECLARATION

Because the first part of a Feature_declaration is a New_feature_list, not just one Extended_feature_name, each feature declaration may introduce more than one feature, as in

← Syntax on pages 141 and 142.

```
a, b, c : INTEGER                          -- Attributes
f, g  require … do … ensure … end          -- Routines
```

Such features introduced together are known as synonyms:

Synonym

A **synonym** of a feature of a class *C* is a feature with a different Extended_feature_name such that both names appear in the same New_feature_list of a Feature_declaration of *C*.

Synonym declarations should be viewed as an abbreviation, according to the following rule:

Unfolded form of a possibly multiple declaration

The **unfolded form** of a Feature_declaration listing one or more feature names, as in:

$$f_1, f_2, \dots, f_n \; declaration\_body \qquad (n \geq 1)$$

where each f_i is a New_feature, is the corresponding sequence of declarations naming only one feature each, and with identical declaration bodies, as in:

f_1 *declaration_body*
f_2 *declaration_body*
…
f_n *declaration_body*

Thanks to the unfolded form, we may always assume, when studying the validity and semantics of feature declarations, that each declaration applies to only one feature name. This convention is used throughout the language description; to define both the validity and the semantics, it simply refers to the unfolded form, which may give several declarations even if they are all grouped in the class text.

→ "Feature Declaration rule", page 160.

A multiple declaration introduces the feature names as synonyms. But the synonymy only applies to the enclosing class; there is no permanent binding between the corresponding features. Their only relationship is to have the same Declaration_body at the point of introduction.

This means in particular that a proper descendant of the class may rename or redeclare one without affecting the other.

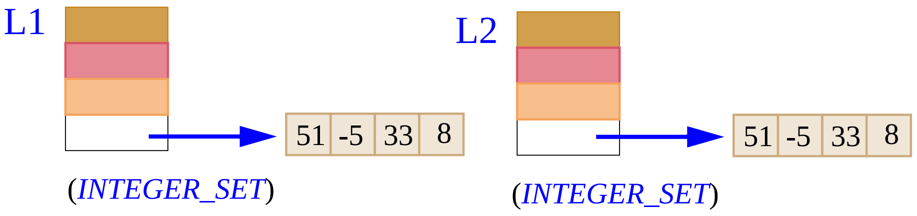
→ Renaming: chapter 6 Redeclaration: chapter 10.

Each f_i, being a New_feature, may include a **frozen** mark. In the unfolded form this mark only applies to the *i*-th declaration.

When should we use multiple declarations? The last observations provide a clue. If you anticipate that a feature may have different variants in descendant classes, it may be better to introduce it as two features, initially identical, in its class of origin. This is in particular the case when you expect descendants to redefine the feature, but want to guarantee them access to the original — for themselves and, if appropriate, their clients. Then you should declare one of the two features as frozen.

ANY, the Kernel Library class serving as ancestor of all developer-defined classes, provides several examples of this technique. *ANY* offers a general comparison function, *is_equal*, originally comparing two objects for field-by-field equality. The semantics of the object comparison operator ~ is defined in terms of *is_equal*. Any class may redefine *is_equal* (and hence the meaning of ~ for operands of the corresponding types) to account for the specific semantics of equality desired for the class. For example, if objects L1 and L2 below are instances of a class *INTEGER_SET*, they are not field-by-field equal (since they contain references to different objects), but the author of *INTEGER_SET* may decide that *is_equal* and ~ must return true on these objects as they represent the same set. The class will redefine *is_equal* to test for the desired notion of equality.

See 21.6, page 572 about the respective roles of functions is_equal and equal.



Equivalent objects not field-by-field equal

Along with such redefinitions of *is_equal*, it is useful to keep the default version (performing field-by-field comparison) for all classes. This is why *ANY* introduces two equality functions, originally as synonyms:

```
is_equal, frozen default_is_equal
            (x: like Current): BOOLEAN is...
```

with the consequence (enforced through the **frozen** mark) that the second function may not be redefined, so that developers can trust it to retain a fixed, universal semantics — indeed, a fixed implementation.

It is also important to understand when multiple declarations are **not** appropriate. This includes the following two situations:

- If you devise a better name for an existing feature, but wish to provide upward compatibility for existing clients and descendants, then a better mechanism, described below, is available: “obsoleting” the feature. This has the advantage of facilitating the eventual phasing out of the obsolete version, whereas there is no incentive to remove a synonym.

"Reusable Software" discusses how to choose the names of library features; see also Appendix 34 of the present book.

- The availability of a synonym mechanism is usually not a good excuse for refusing to choose between possible names. Class designers, especially designers of reusable library classes, should not be fickle; even if two sets of names appear equally good, it is generally better to choose just one than to provide both. By passing on the choice to client developers, the latter solution would only confuse them, and make the class appear more complex than it is.

These observations suggest that multiple declarations, although an important facility for cases such as the one mentioned above, should remain a relatively infrequent occurrence in normal Eiffel development.

The example also suggests what kinds of use are proper for frozen features. The very idea of "freezing" a feature is, in general, contrary to the fundamental Eiffel concepts of software extendibility and adaptability, which the feature adaptation mechanisms (in particular redeclaration) support directly. When you inherit from a class, you should be able to adapt its features to the new context; you may use the assertion mechanism to guarantee that the specification remains compatible with the framework defined in the original, although the implementation may be different.

This mechanism is so central in the Eiffel method that it leaves only a limited role for frozen features: taking care of predefined, system-level operations such as *is_identical*, for which we require not only the specification but the implementation to be determined once and for all.

5.19 VALIDITY OF FEATURE DECLARATIONS

To be valid, a Feature_declaration must satisfy a constraint, known as the Feature Declaration rule. Here is the rule in full, followed by a detailed explanation of its clauses.

← The syntax of Feature_declaration *was given on page 141*

Feature Declaration rule *VFFD*

A Feature_declaration appearing in a class *C* is valid if and only if it satisfies all of the following conditions for every declaration of a feature *f* in its unfolded form:

1 • The Declaration_body describes a feature which, according to the rules given earlier, is one of: variable attribute, constant attribute, procedure, function.

← The rules for determining the kind of feature are those of 5.12.

2 • *f* does not have the same feature name as any other feature introduced in *C* (in particular, any other feature of the unfolded form).

3 • If *f* has the same feature name as the final name of any inherited feature, the Declaration_body satisfies the Redeclaration rule.

→ The full Redeclaration rule is on page 307.

4 • If the Declaration_body describes a deferred feature, then the Extended_feature_name of *f* is not preceded by **frozen**.

5 • If the Declaration_body describes a once function, the result type is stand-alone.

6 • Any anchored type for an argument is detachable.

7 • The Alias clause, if present, is alias-valid for *f*.

→ *As defined on page 162 below.*

Additional obligations apply if there is an Assigner_mark; they are covered by the Assigner Procedure rule (automatically included thanks to the General Validity rule).

← *Assigner Procedure rule: page 156; General Validity rule: page 98.*

As stated at the beginning of the rule, the conditions apply to the *unfolded form* of the declaration; this means that the rule treats a multiple declaration f_1, f_2, ... , f_n *declaration_body* as a succession of *n* separate declarations with different feature names but the same *declaration_body*.

← *"Unfolded form of a possibly multiple declaration", page 158.*

Conditions 1 and 2 are straightforward: the Declaration_body must make sense, and the name or names of the feature being introduced must not conflict with those of any other feature introduced in the class.

A redeclaration is either a **redefinition** of an inherited feature (changing its specification, signature or implementation) or an **effecting** (an effective implementation of a feature inherited in deferred form). The exact requirements in this case are captured by the Redeclaration rule, which will be given when we complete the study of inheritance, redefinition and deferred features.

In applying conditions 2 and 3, remember that two feature names are "the same" not just if they are written identically, but also if they only differ by letter case. Only the identifiers (Feature_name) of the features play a role in this notion, not any Alias they may have.

← *Definition of "same feature name", page 153.*

The Feature Name rule will state a consequence of conditions 2 and 3 that may be more appropriate for error messages in some cases of violation.

→ *Page 466.*

Condition 4 prohibits a frozen feature from being declared as deferred. The two properties are conceptually incompatible since frozen features, by definition, may not be redeclared, whereas the purpose of deferred features is precisely to force redeclaration in proper descendants.

← *Frozen features were introduced on page 142.*

A companion constraint, seen as part of the Redefine Subclause rule in a later chapter, will prohibit the *redefinition* of a frozen feature.

→ *Page 301.*

Condition 5 applies to once functions. A once routine only executes its body on its first call. Further calls have no effect; for a function, they yield the result computed by the first call. This puts a special requirement on the result type *T* of such a function: if the class is generic, *T* should not depend on any formal generic parameter, since successive calls could then apply to instances obtained from different generic derivations; and *T* must not be anchored, as in the context of dynamic binding it could yield incompatible types depending on the type of the target of each particular call. The notion of *stand-alone type* captures these constraints on *T*.

→ *"ONCE ROUTINES", 23.14, page 633.*

→ *See, in the chapter on types: "STAND-ALONE TYPES", 11.12, page 340.*

Condition 6 addresses delicate cases of polymorphism and dynamic binding, where anchored arguments and their implicit form of "covariance" may cause run-time errors known as "catcalls". It follows from the general rule for signature conformance and is discussed with it.

→ "Signature conformance", page 378.

The last condition, 7, is the consistency requirement on features with an operator or bracket alias. It relies on the following definition (which has a validity code enabling compilers to give more precise error messages).

VALIDITY

Alias Validity rule *VFAV*

An Alias clause is **alias-valid** for a feature *f* of a class *C* if and only if it satisfies the following conditions:

1 • If it lists an Operator *op*: *f* is a query; no other query of *C* has an Operator alias using the same operator and the same number of arguments; and either: *op* is a Unary and *f* has no argument, or *op* is a Binary and *f* has one argument.

2 • If it lists a Bracket alias: *f* is a query with at least one argument, and no other feature of *C* has a Bracket alias.

3 • If it includes a **convert** mark: it lists an Operator and *f* has one argument.

4 • If it lists one of the "semistrict" operators **and then**, **or else** and **implies**: *C* is the Kernel Library class *BOOLEAN*.

PURPOSE

The first two conditions express the uniqueness and signature requirements on operator and bracket aliases:

• An operator feature *plus* **alias** "§" can be either unary (called as § *a*) or binary (called as *a* § *b*), and so must take either zero or one argument. Two features may indeed share the same alias— like *identity* **alias** "+" and *plus* **alias** "+", respectively unary and binary addition in class *INTEGER* and others from the Kernel Library — as long as they have different identifiers (here *identity* and *plus*) and different signatures, one unary and the other binary.

Such a feature must return a result and hence be a query — attribute or function. An attribute is only possible in the unary case, and is indeed permitted in line with the Uniform Access principle, although in most practical cases you'll need a function.

→ "UNIFORM ACCESS", 23.4, page 616.

• A bracket feature, of which there may be at most one in a class, will be called under the form *x* [a_1, … a_n] with $n \geq 1$, and so must be a query with at least one argument (and hence a function). Condition 2 tells us that there may be at most one bracket feature per class.

Condition 3 indicates that a **convert** mark, specifying "target conversion" as in *your_integer + your_real*, makes sense only for features with one argument, with an Operator which (from condition 1) must be a Binary.

Condition 4 addresses the special case of the three semistrict boolean operators, which follow unique semantic rules enabling them not to evaluate their second operand in some cases: *a* **and then** *b* is guaranteed not to evaluate *b* if *a* evaluates to false, the result then being necessarily false. Although the language definition almost always provides generality — based on the principle that if a technique is useful to the language designer it must also be useful to the language user — it makes an exception here, because there is no simple way for programmers to write a semistrict function definition. So **and then**, **or else** and **implies** are for the private use of class *BOOLEAN*. As condition 4 indicates, even its own proper descendants cannot redefine these features.

5.20 SCOPE OF NAMES

Any feature of a class is accessible for use in any Feature_declaration of the class, and in its Invariant clause. Examples of the first use include unqualified (direct) calls in the Feature_body, Precondition, Postcondition and Rescue clauses of a routine, and use as Anchor for an Anchored type in a declaration of a feature or of a routine argument.

→See chapters 8 about Feature_body, *9 about* Precondition *and* Postcondition, *26 about* Rescue.

To avoid any ambiguity, constraints will prohibit reusing the name of a feature of the class for any other entity appearing in the class: formal argument or local variable of a routine, Object-Test Local of an Object_test.

→Formal argument rule, page 216; Local variable rule, page 222; Object Test rule, page 651.

The Feature Declaration rule does **not**, however, prohibit conflicts between feature names and names of **classes**. It is possible for a feature to bear the same lower name as a class of the universe. You may sometimes find it convenient to write a feature declaration such as

```
error_window: ERROR_WINDOW
```

in a class text which only needs one feature of a certain type (here given by class *ERROR_WINDOW*) if you consider that the type name provides enough information to describe the role of the feature.

5.21 OBSOLETE FEATURES

As classes evolve in the constantly changing world of software development, you may find that a feature is no longer satisfactory.

If all you need is to change its implementation, then you should be able to update the feature without affecting its dependent classes (clients and proper descendants). For example, you may change a Feature_body, even if this causes replacing an attribute by a function or conversely, with minimum impact on dependents.

Unfortunately, this is not always the case. You may become unhappy with a feature's name, its signature or its specification — all of which are part of the interface and known to the clients.

The specification defines the routine's semantics. It is normally expressed by assertions; see chapter 9.

In such a situation, if you are certain that you have found a better replacement for the feature, you should perform the change without delay, for fear of prolonging the life of inferior software versions. But you must also take into consideration any existing dependent classes that relied on the feature. Clearly, you should avoid any change that would suddenly prevent such classes from functioning; but you may want to encourage their authors to adapt them to the new version within a reasonable time.

The preceding chapter showed how to declare an entire **class** as obsolete. This is a rather drastic decision; more often, the class as a whole remains adequate, but you want to update a few features.

← See 4.11, page 128, about obsoleting an entire class.

The feature obsolescence mechanism supports this need. By declaring a feature as **obsolete**, you keep it usable exactly as it was, while alerting its users to the existence of a better version. This provides a graceful way to phase out a feature while remaining friends with the developers of its clients.

Both routines and attributes may become obsolete. To mark a routine obsolete, give it an Obsolete clause, of the form

```
obsolete Message
```

where *Message* is a Manifest_string. This serves to warn authors of dependent classes that the routine should no longer be used. The *Message* should direct readers to alternate features.

Here is an obsolete routine which once figured in class *ARRAY* of the Kernel Library:

```
enter (i: INTEGER; new_value: T)
                obsolete "Use 'put (new_value, i)'"
            -- Replace by new_value the element at index i
        require
            i >= lower; i <= upper
        do
            ...(Appropriate algorithm)...
        ensure
            set: item (i) = new_value
        end
```

In older versions of the library, *enter* was the routine used to replace by *new_value* the value of the element at index *i* in an array. An examination of the consistency of names and conventions in the library resulted in a decision to update the routine; both the name (*put* rather than *enter*) and the order of arguments were changed. The Message explains this change.

To avoid cluttering library classes with features that are no longer relevant, library maintainers should not allow obsolete routines to loiter forever. After a suitable grace period — time for one or two new releases of the software to displace the older generations — they will have fulfilled their duties as Client-Friendly Transition Facilitators and should be retired with honors. This indeed happened to the above version of *enter*, which (although fondly remembered) disappeared long ago from class *ARRAY*, allowing — by an unforeseen twist of fate — *enter* to reappear as a synonym of *put*.

The syntax of class-level Obsolete clauses also applies — through the production for Feature_value — to routine-level clauses; here it is again:

← Page 141.

← This syntax appeared originally on page 129, followed by the semantics, applicable to obsolete features as well as obsolete classes.

Obsolete $\triangleq$ **obsolete** Message

Message $\triangleq$ Manifest_string

As we saw there, marking a feature as Obsolete does not affect its semantics. But language processing tools may produce a warning when they process a client or descendant class that uses the feature. The warning should include the Message.

The contract view of a class does not retain any feature marked obsolete.

→ "-DOCUMENTING THE CLIENT INTERFACE OF A CLASS", 7.8, page 207.

A compiler or other language processing tool may also go further and provide an option that, under some conditions, will automatically update the text of client classes, replacing all calls to an obsolete routine by the body of the routine with appropriate argument substitution.

As noted in the discussion of obsolete classes, the availability of a feature obsolescence mechanism is not an excuse to grant a reprieve to software components that are buggy or otherwise deficient. If you discover a specification, design or implementation flaw, only one reaction is reasonable: correcting the mistake. A routine is a candidate for obsolescence only when, as originally written, it adequately covered a certain need, but is not in a manner satisfying your current standards. You prefer the new version, but the obsolete version is not *wrong*; it's just not what you wish to keep for the future.

5.22 NO IN-CLASS OVERLOADING

A consequence of the various validity rules on features and their names — to be expressed fully by the Feature Name rule — is that Eiffel never permits the same name to denote different features within the scope of a given class. This is expressed by the Feature Name rule.

→ *Page 466.*

When you see a feature name *f* in a class you immediately know what feature it refers to; and when you see a feature call *a*.*f* (…) with *a* of type *T* you can immediately find the feature *f* to which it refers in *T*. (The actual feature to be called may of course, as a result of dynamic binding, be a redeclared version from a descendant.)

The full Feature Name rule appears only in a later chapter because it must take into account, along with the features introduced in a class, those inherited from its parents, and possibly renamed in the process: no name conflicts must arise between any of these. The rule must handle the effect of all inheritance mechanisms, including renaming, redefinition, and sharing under repeated inheritance.

The result, however, is simple: no name conflicts, period. The "overloading" mechanisms permitted by some languages is a confusing facility with no known benefit. It contradicts the principles of object technology and creates difficulties for both language users and compiler writers. The idea of using the same name to denote *different things* within a given scope can at best be described as rather puzzling.

Eiffel enforces instead the clear rule that in one class one feature name means one thing. (Nothing prevents you, as noted, from using the same name for feature names and *class* names, which can cause no ambiguity.)

← *"SCOPE OF NAMES", 5.20, page 163.*

The only form of overloading permitted by Eiffel is the reuse of a feature name across *different* classes. The systematic naming conventions of the recommended style actually encourage you in this direction; the idea is to use a single name for features that correspond to the *same* basic semantics adapted to different contexts — the reverse of in-class overloading. Inter-class overloading takes its full power through dynamic binding, which allows dynamic (run-time) selection of the proper semantic variant, where intra-class overloading is static (compile-time). The dynamic form of inter-class overloading can also be called *semantic* overloading, in contrast with the *syntactic* nature of in-class overloading.

What is commonly known as "operator overloading", the possibility of using the same operators, arithmetic in particular, for operations on different data types, is provided in a more general and flexible way by the combination of Alias clauses, permitting operator syntax for calls, and the conversion mechanism.