8 Routines

8.1 OVERVIEW

Routines describe computations.

Syntactically, routines are one of the two kinds of feature of a class; the other kind is attributes, which describe data fields associated with instances of the class. Since every Eiffel operation applies to a specific object, a routine of a class describes a computation applicable to instances of that class. When applied to an instance, a routine may query or update some or all fields of the instance, corresponding to attributes of the class.

→ Chapter 18 explores attributes.

A routine is either a procedure, which does not return a result, or a function, which does. A routine may further be declared as **deferred**, meaning that the class introducing it only gives its specification, leaving it for descendants to provide implementations. A routine that is not deferred is said to be **effective**.

An effective routine has a **body**, which describes the computation to be performed by the routine. A body is a Compound, or sequence of instructions; each instruction is a step of the computation.

The present discussion explores the structure of routine declarations, ending with the list of possible various forms of instructions.

8.2 ROUTINE DECLARATION

A routine declaration describes the interface of a routine and, unless the routine is deferred, its implementation.

Here are two routine declarations; *total* is a function, *move* a procedure.

```
total: INTEGER
        -- Sum of attributes a, b and c
    deferred
    ensure
        summed: Result = a + b + c
    end
```

```
move (mice: MOUSE; men: MENU)
        -- Move mouse cursor to first item in menu.
    require
        men_exist: men /= Void
    do
        mice.move (men)
    end
```

It is not necessary to repeat the name of the routine as an ending comment, writing for example **end** -- *move*, as an earlier convention suggested. Most routine texts in well-written Eiffel texts are short, so the ending comment tends to obscure, not help. You may still use an ending comment for the occasional long routine.

A Feature_declaration, as you will remember, declares a routine if and only if it satisfies the following condition:

← *"HOW TO RECOGNIZE FEATURES", 5.12, page 145.*

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

The Formal_arguments and Type_mark parts may or may not be present. If the Query_mark is present, the declaration describes a function; otherwise it describes a procedure.

As with any other feature, a routine declaration may include more than one routine name, as in the following declaration of three procedures:

```
proc2, proc3, proc4 (x, y: REAL)
    require
        x > y
    do
        print (sqrt (x – y))
    end
```

The meaning, as in the general case of "synonym" features, is the same as that of three separate declarations with identical Declaration_body.

← *"SYNONYMS AND MULTIPLE DECLARATION", 5.18, page 158.*

The routines remain otherwise independent; in particular, redefining or renaming one in a descendant does not affect the others.

→ *Renaming: chapter 6 ;redefinition: chapter 10.*

8.3 FORMAL ARGUMENTS

A routine may have arguments, corresponding to information that callers will pass to every execution of the routine.

Formal argument, actual argument

Entities declared in a routine to represent information passed by callers are the routine's **formal arguments**.

The corresponding expressions in a particular call to the routine are the call's **actual arguments**.

Rules on Call require the number of actual arguments to be the same as the number of formal arguments, and the type of each actual argument to be compatible with (conform or convert to) the type of the formal argument at the same position in the list.

→ Chapter 25. See 8.4, page 217 below about achieving the effect of a variable number of arguments.

A note on terminology: Eiffel always uses the term **argument** to refer to the arguments of a routine. The word "parameter" is never used in this context, because it could create confusion with the types that can parameterize *classes*, called **generic parameters**.

← About genericity see chapter 12.

Function *total* seen earlier has no arguments. Procedure *move* has two formal arguments called *mice* and *men*. Assuming both *move* and *total* appear in a class *C*, instructions using typical calls to these routines, appearing in some routine of a class *B*, might be

```
c1. move (mo, me)
n := c1.total
```

with *c1* of type *C*, *mo* of type *MOUSE*, *me* of type *MENU*, *n* of type *INTEGER*. Expressions *mo* and *me* are the actual arguments of the first call.

The formal arguments of *move* were all of different types. As with feature names in a Feature_declaration, you may group two or more formal arguments of the same type into an Entity_declaration_group. The comma serves as separator, as in this routine from class *TWO_WAY_LIST* in EiffelBase:

```
update_after_deletion
                    (one, other: like first_element; index: INTEGER)
        … Rest of routine omitted …
```

This declares both *one* and *other* as being of type **like** *first_element*. The effect would have been identical with a routine header of the form

```
update_after_deletion
      (one: like first_element;
      other: like first_element;
      index: INTEGER)
```

The preceding examples illustrate the general form of the Formal_arguments part of a routine declaration.

Formal argument and entity declarations

```
         Formal_arguments ≜ "(" Entity_declaration_list ")"
  Entity_declaration_list ≜ {Entity_declaration_group ";" …}+
 Entity_declaration_group ≜ Identifier_list Type_mark
          Identifier_list ≜ {Identifier "," …}+
```

As with other semicolons, those separating an Entity_declaration_group from the next are optional. The style guidelines suggest including them for successive declarations on a line, as with short formal argument lists, but omitting them between successive lines, as with local variable declarations (also covered by Entity_declaration_group).

A validity constraint mandates a choice of name avoiding any ambiguity:

Formal Argument rule *VRFA*

Let *fa* be the Formal_arguments part of a routine *r* in a class *C*. Let *formals* be the concatenation of every Identifier_list of every Entity_declaration_group in *fa*. Then *fa* is valid if and only if no Identifier *e* appearing in *formals* is the final name of a feature of *C*.

Another rule, given later, applies the same conditions to names of local variables. Permitting a formal argument or local variable to bear the same name as a feature could only cause confusion (even if we had a scoping rule removing any ambiguity by specifying that the local name overrides the feature name) and serves no useful purpose.

→*"LOCAL VARIABLES AND RESULT", page 221.*

The standard Eiffel style suggests different conventions anyway for features and formal arguments. Features, which have a wide scope (meaning that they can be used throughout a class and all its descendants), must have clear, meaningful names, typically made of one or more full words, separated, if more than one, by underscores, as in *spouse_name* (but not overqualified by the class name: if this feature appears in a class *EMPLOYEE*, do not call it *employee_spouse_name* as this would be redundant). For a formal argument, which has a small scope — most routines in Eiffel are short — the declaration of the argument and its uses will seldom be more than a few lines apart; you should choose short, simple names. Abbreviations are perfectly all right, as in *update_price* (*r*: *RATE*; *pc*: *PROMOTION_CODE*).

Being too pompous about names of formal arguments may *decrease* readability by giving arguments more attention that they deserve. Features are the aristocracy of a class and deserve full glory; formal arguments (and local variables) are their servants and should not try to shine above their rank.

Beginners sometimes use names of the form *a_TYPE_NAME*, as in *raise_salary* (*a_rate*: *RATE*; *a_promotion_code*: *PROMOTION_CODE*). Seasoned Eiffel developers consider this revolting kitsch.

See "Further reports of abominable taste in the provinces", in Proc. BISTOORI (45th Intl. Conf. on Biedermeier Influences on the Style of Typical Object-Oriented Retrograde Implementations), Sotchi, 2004, pp. 2045-3497.

Complementing the Formal Argument rule is a general rule — also applicable to local variables, studied later in this chapter — that precludes using the same identifier twice in an Entity_declaration_list. Clearly, in

```
x: T1
x, y, y: T2
```

***WARNING**: not valid!*

the type of *x* would be ambiguous. The type of *y* would not be ambiguous since the two occurrences are part of the same Entity_declaration_group, but the duplicate listing of *y* is invalid all the same; it can serve no useful purpose. This is the only condition on an Entity_declaration_list:

Entity Declaration rule *VRED*

Let *el* be an Entity_declaration_list. Let *identifiers* be the concatenation of every Identifier_list of every Entity_declaration_group in *el*. Then *el* is valid if and only if no Identifier appears more than once in the list *identifiers*.

8.4 USING A VARIABLE NUMBER OF ARGUMENTS

From the above syntax, and the previewed constraint on valid calls, it follows that every routine has a fixed number of arguments, which is the number of entities appearing in the Entity_declaration_list of its Formal_arguments part.

These rules do not prevent you from obtaining the effect of routines with variable numbers of arguments if you so desire. If the arguments are of arbitrary types, you may replace by a single argument of type *TUPLE*, corresponding to a sequence of arbitrary values, as in

→ See chapter 13 about tuples.

```
write_formatted (values: TUPLE; format: STRING)
            -- Print all elements of values, under given format.
      …See below about the procedure body …
```

which you can then call with a "Manifest tuple" consisting of a sequence of values in brackets:

```
write_formatted ([your_integer, your_string, your_real],
                         your_output_format)
```

The procedure body will analyze the successive tuple elements and their types. The discussion of tuples shows how to write it.

→ *"Emulating a variable number of arguments", page 372* .

If all the items are of types conforming to a known *T* you can, as an alternative to tuples, use *ARRAY* [*T*] as the argument type. A routine to print numeric values could read

```
write_numerics ( values: ARRAY [NUMERIC] )
                -- Print all elements of values.
        …Procedure body omitted …
```

where a typical call appears as:

```
write_numerics ({ARRAY [NUMERIC]} [your_integer, your_real])
```

Here we are passing an *INTEGER* and a *REAL*; both of these types conform to *NUMERIC*. The manifest array passed as argument is a tuple converted into an array.

8.5 ROUTINE BODY

A Feature_body had three possible forms:

← *This syntax appeared first on page 144.*

```
Feature_body ≜ Deferred | Effective_routine | Attribute
```

The last case will be studied in the discussion of attributes. Routines correspond to the first two cases:

Routine bodies

```
         Deferred ≜ deferred
Effective_routine ≜ Internal | External
         Internal ≜ Routine_mark Compound
     Routine_mark ≜ do | Once
             Once ≜ once [ "("Key_list ")" ]
         Key_list ≜ {Manifest_string "," …}+
```

A Feature_body of the first possible form, Deferred, consists of the sole keyword **deferred**; this indicates that the routine, and as a consequence the enclosing class, are deferred .

→ *See "DEFERRED FEATURES", 10.11, page 266 and subsequent sections.*

A routine of the other form, Effective_routine, may be External, indicating that it is implemented in another language. In the remaining and by far the most common case, Internal, the routine body is a Compound: a sequence of instructions describing the algorithm to be executed (after initialization of any local variables including *Result* for a function) on a call to the routine.

→Compound *and other control structures are the topic of chapter 17.*

The introductory keywords **do** and **once** of an Internal body correspond to different semantics for calls to the routine:

→*For details: "PRECISE CALL SEMANTICS", 23.17, page 643.*

- With a **do** body the initialization and body are executed anew for each call.

- If routine *o* of class *C* has a **once** body (*o* is then called a ''once routine''), the initialization and body are executed only for the first call to *o* applied to an instance of *C* during any given session. For every subsequent call on an instance of *C* during the same session, the routine call has no effect; if the routine is a function, the value it returns is the same as the value returned by the first call. Once routines are useful for "smart initialization" actions which must be applied the first time a certain structure is accessed, and for shared information. They help avoid the global variables of conventional programming languages.

  →*"ONCE ROUTINES", 23.14, page 633.*

  You can fine-tune the meaning of "once" by including *once keys* after the keyword **once**, as in **once** ("*THREAD*") to specify that the execution will take place once in each thread. Other predefined values include "*PROCESS*" (the default) and "*OBJECT*" (to require computation once for every instance). You can even define your own once keys and then reset the key, through *onces*.*reset* ("*YOUR_KEY*"), ensuring that the next call to any once routine using this key will execute its body again. The mechanism also makes it possible to define variable keys to be set from outside the Eiffel text proper, for example in an Ace file.

It is convenient to introduce precise terms:

Once routine, once procedure, once function

A **once routine** is an Internal routine *r* with a Routine_mark of the Once form.

If *r* is a procedure it is also a **once procedure**; if *r* is a function, it is also a **once function**.

Here is an example procedure (from the EiffelTime library) with all the optional components except Obsolete and Rescue clauses:

```
make_fine (h, m: INTEGER; s: DOUBLE)
            -- Set hour, minute and second to h, m and integer part of s;
            -- Set fractional_second to fractional part of s.
      require
            correct: is_correct_time (h, m, s, False)
      local
            s_trunc: INTEGER
      do
            s_trunc :=  s.truncated_to_integer
            fractional_second := s – s_trunc
            make (h, m, s_trunc)
      ensure
            hour_set: hour = h
            minute_set: minute = m
            fine_second_set: fine_second = s
      end
```

The various components and their respective roles are the following. All components except the Feature_body and the final **end** are optional.

• The text appearing immediately after the routine name and arguments, starting with --, is a Header_comment explaining the purpose of the routine. Other comments may be inserted at the end of any line; but this one has a special role, documenting the routine's interface. The "contract view" of a class retains header comments.

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

• The keyword **require** introduces an Assertion, called the Precondition of the routine. This expresses the conditions under which a call to the routine is correct. Here *is_correct_time* must be true for the arguments given. The Identifier correct is a label identifying that assertion.

→ *Chapter* 9

• The Local_declarations clause, studied below, declares local variables used only within the routine body, and initialized anew on each call. Here *make_fine* uses a local variable *s_trunc* of type *INTEGER*.

• The Feature_body is here of the Effective kind, more specifically Internal, starting with **do** (the other possibility is **once**) and continuing with instructions — zero or more in the general case, here three.

• The keyword **ensure** introduces another Assertion, the Postcondition of the routine. This expresses the conditions that a routine call will ensure on return if called in a state satisfying the precondition. Here it states that a number of queries have been set from the values of the arguments.

→ *Chapter* 9

The example does not include a Rescue clause. If present, this describes what to do if an exception occurs during an execution of the routine. The absence of a Rescue clause has the same effect as the presence of a Rescue clause just consisting of a call to the procedure *default_rescue* of the universal class *ANY*. So the example routine could have been written equivalently as

→ The Rescue clause and procedure default_rescue are discussed in detail in chapter 26.

```
make_fine (h, m: INTEGER; s: DOUBLE)
          ... All other clauses as above ...
     rescue
          default_rescue
     end
```

In its original form, *default_rescue* has a null effect, but a class may redefine it to provide specific exception handling.

8.6 LOCAL VARIABLES AND *RESULT*

If present in a routine, a Local_declarations clause is the declaration of variable entities available only within the Feature_body; they are useful for the computation it describes, but their values do not need to be retained by the current object after a call to the routine.

The last example introduced just one local variable:

```
     local
          s_trunc: INTEGER
```

used in the **do** clause to hold the value of *s*.*truncated_to_integer*, needed by two of the instructions.

In this example there is no need for an automatic initialization of the variable since the first instruction of the routine assigns it a value. Such an explicit assignment is not required; if the routine's execution accesses the value of the variable when it has not been assigned, initialization rules guarantee a well-defined initial value, for example 0 for integers and False for booleans.

→ "Default Initialization rule", page 508.

The general structure of a Local_declarations clause is:

Local variable declarations

Local_declarations $\triangleq$ **local** [Entity_declaration_list]

The Entity_declaration_list may be absent, as we tolerate an empty **local** part — perhaps while you are refactoring your software and moving local variable declarations in and out.

In addition to the earlier constraint requiring all identifiers in an Entity_declaration_list to be different, we must avoid any ambiguity between local variables and features of the class:

← *“VRED”, page 217*

Local Variable rule *VRLV*

Let *ld* be the Local_declarations part of a routine *r* in a class *C*. Let *locals* be the concatenation of every Identifier_list of every Entity_declaration_group in *ld*. Then *ld* is valid if and only if every Identifier *e* in *locals* satisfies the following conditions:

1 • No feature of *C* has *e* as its final name.

2 • No formal argument of *r* has *e* as its Identifier.

Most of the rules governing the validity and semantics of declared local variables also apply to a special predefined entity: **Result**, which may only appear in a function or attribute, and denotes the value to be returned by the function. The following definition of “local variable” reflects this similarity.

Local variable

The local variables of a routine include all entities declared in its Local_declarations part, if any, and, if it is a query, the predefined entity **Result**.

Result can appear not only in the Compound of a function or variable attribute but also in the optional Postcondition of a constant attribute, where it denotes the value of the attribute and allows stating abstract properties of that value, for example after a redefinition. In this case execution cannot change that value, but for simplicity we continue to call **Result** a local “variable” anyway.

When applying validity and semantics rules, you must treat **Result** as an entity of the type declared for the enclosing function’s result. For example, this function from class *CLOSED_FIGURE* in EiffelVision treats *Result* as a local variable of type *INTEGER*:

```
fill_style_count: INTEGER
            -- Number of defined fill styles for this figure
    do
            Result := global_fill_style_count + local_fill_style_count
    end
```

8.7 EXTERNALS

A routine may have a Feature_body of the External form, which means that its implementation is written in another language.

The following examples illustrate the form of an External body:

```
open_file (file_od: INTEGER; mode: CHARACTER)
          -- Open file_od in mode mode.
     require
          file_status (file_od) <= 0
     external
          "C"
     end
```

```
file_status (file_od: INTEGER): INTEGER
          -- Current status of file associated with file_od
     external
          "C"
     alias
          "_fstat"
     end
```

They enable other Eiffel elements to call a C procedure under the Eiffel name *open_file* and a C function under *file_status*.

Such routines are viewed by the rest of a system as normal Eiffel routines; their only special property is that their execution, instead of being under the control of the Eiffel system to which they belong, is a call to some code generated by a compiler for the foreign language.

The second external routine of the example, a function, has a subclause of the form **alias** *external_name*, indicating that this function will be known through an Eiffel name, *file_status*, different from its name in the foreign language; by default the two would be the same. Here an alias is required since the C name, *_fstat*, begins with an underscore and so is not a valid Eiffel identifier.

The External mechanism include a wide set of possibilities; in particular, you may include inline C code directly, through the **alias** clause. A later chapter is entirely devoted to this mechanism.

→ *Chapter* 31.

8.8 TYPES OF INSTRUCTIONS

The Internal body of a non-deferred routine is a Compound, or sequence of instructions. As an introduction to the detailed study of instructions in ter chapters, here is an overview of the available variants. The syntax is:

Instructions

```
Compound    ≜ {Instruction ";" ...}*
Instruction ≜ Creation_instruction | Call |
              Assignment | Assigner_call |
              Conditional | Multi_branch | Loop
              | Debug | Precursor | Check | Retry
```

A Compound is a possibly empty list of instructions, to be executed in the order given. In the various parts of control structures, such as the branches of a Conditional or the body of a Loop, the syntax never specifies Instruction but always Compound, so that you can include zero, one or more instructions.

→ *Chapter 20.*

A Creation_instruction creates a new object, initializes its fields to default values, calls on it one of the creation procedures of the class (if any), and attaches the object to an entity.

Call applies a routine to the object attached to a non-void expression. For the Call to yield an instruction, the routine must be a procedure.

→ *Chapter 23.*

Assignment changes the value attached to a variable.

→ *Chapter 22.*

An Assigner_call is a procedure call written with an assignment-like syntax, as in *x*.*a* := *b*, but with the semantics of a call, as just a notational abbreviation for x.*set_a (b)* where the declaration of *a* specifies an assigner command *set_a*.

→ *"ASSIGNER CALL", 22.12, page 599.*

Conditional, Multi_branch, Loop and Compound describe complex instructions, or control structures, made out of other instructions; to execute a control structure is to execute some or all of its constituent instructions, according to a schedule specified by the control structure.

→ *Control structures and* Debug: *chapter 17.*

Debug, which may also be considered a control structure, is used for instructions that should only be part of the system when you enable the *debug* compilation option.

Precursor enables you, in redefining a routine, to rely on its original implementation.

→ Precursor: *10.24, page 293.*

Check is used to express that certain assertions must hold at certain moments during run time.

→ *Assertions and* Check: *chapter 9.*

Retry is used in conjunction with the exception handling mechanism.

→ *Exceptions and* Retry: *chapter 26.*