This draft: Version 4.82-00-00, 21 September 02 12:15 (Zürich time)

Types

11.1 OVERVIEW

Types describe the form and properties of objects that can be created during the execution of a system. The type system lies at the heart of the object-oriented approach; the use of types to declare all entities leads to more clear software texts and permits compilers to detect errors and inconsistencies before they can cause any damage.

This chapter — complemented by the next two, which address generic types and tuple types — presents the type system.

11.2 THE ROLE OF TYPES

Every object is an instance of some type. (More precisely, it is a *direct instance* of exactly one type; thanks to the inheritance mechanism it may also be an instance of other, more general types.) Class texts may refer to eventual run-time objects through the software elements that denote values: constants, attributes, function calls, formal routine arguments, local entities, and expressions built from such elements.

Typing in Eiffel is static. For software developers, this means four practical properties:

- Every element denoting run-time values is *typed*: it has an associated type, limiting the possible types of the attached run-time objects.
- This type is immediately clear — to a human reader or to a language processing tool — from the element itself or the surrounding software text. For a manifest constant, such as the Integer *421*, the type follows from the way the constant is written; in all other cases it is a consequence of a type declaration, made compulsory by the validity rules of the language.
- Non-atomic constructs impose complementary validity constraints, defining admissible type combinations. For example, an assignment requires the type of the source to conform to the type of the target.

- Since the constraints are defined as conditions on the software text, language processing tools such as compilers or static analyzers may check the type consistency of a system **statically**, that is to say, just by examining the system's text, without making any attempt at execution.

This *explicit* and *static* approach to typing has a number of advantages. It makes software texts easier to read and understand, since developers, by declaring the types of entities, reveal how they intend to use them. It enables compilers and other tools to catch many potential errors by detecting inconsistencies between declarations and actual uses. It gives compilers information that helps them generate much more efficient code than would be possible with an untyped (or more weakly typed) language.

Typing in Eiffel is taken seriously. Many languages that claim to be statically (or even "*strongly*") typed allow developers to cheat the type system, enticing them into sordid back-alley deals sometimes known as *casts*. No such cheating exists in Eiffel, where the typing rules suffer no exception. This is essential if we want to have any trust in our software. The only price to pay for this added security is the need to declare entities explicitly and to observe validity constraints — obligations which are even easier to justify if you observe that the type system, far from being a hindrance to the developer's freedom of expression, helps in the production of powerful and readable software systems.

It should be noted, however, that some conceptual issues, having to do with *covariance* and *descendant hiding* can cause type problems in certain borderline cases. The chapter on type checking discusses them.

→ *Chapter 24.*

The present chapter and the next two (on generic types and tuple types) explore the basic forms of types and their properties. This will not exhaust, however, the issue of typing, which pervades most of the discussions of this book. To understand the type system fully, you will need important complements provided by two separate chapters:

- The discussion of **conformance** will explain how a type may be used in lieu of another, and its instances in lieu of that other's instances.
- The presentation of the **type checking** policy will show how the typing policy defines the fundamental validity constraints on the most important computational construct — feature call.

→ *Conformance is the topic of chapter 14. Chapter 23 covers calls; on type checking, see chapter 24.*

11.3 WHERE TO USE TYPES

You will need to write a type — a specimen of the construct Type — in the following contexts:

1 • To declare the result type of an attribute or function: construct Declaration_body. ← *Syntax: page 80.*

2 • To declare the arguments of a routine or inline agent: construct Formal_arguments, defined in terms of Entity_declaration_list. ← *Page 144; see also* Inline_agent, *p. 578.*

3 • To declare a local routine entity: construct Local_declarations (also defined in terms of Entity_declaration_list). ← *Page 150.*

4 • To indicate that a class has a certain parent: construct Parent, as part of Inheritance. ← *Page 103.*

5 • To specify actual generic parameters, as explained in the next chapter: construct Actual_generics. → *Page 256.*

6 • To specify a generic Constraint, also in the next chapter: construct Constraint, part of Formal_generics. → *Page 261.*

7 • To indicate an explicit creation type in a creation instruction or expression: construct Explicit_creation_type. → *Page 435.*

8 • To choose from a set of instructions, based on an expressions's type, in a Multi_branch. → Choice, *page 362*:.

9 • To specify the parameters (component types) of a Tuple_type. → *Page 274.*

10 •To declare the type of an open operand of a Call_agent. → Agent_actual, *page 579*:

As an example of the first three cases, here is the beginning of a possible function declaration:

```
total_occupied_area (wl: LIST [WINDOW]): RECTANGLE is
            -- Smallest rectangle that covers the representations
            -- of all windows in wl
        local
            xmin, ymin, xmax, ymax: REAL
            ... Rest of routine omitted ...
```

In this example and all the others, types are easy to recognize: apart from keywords such as **like**, they use all-upper-case names.

The function has a result (case 1) of type *RECTANGLE*, probably a reference type, and one argument (case 2) of type *LIST* [*WINDOW*], a "generically derived" reference type. It uses four local entities (case 3) of type *REAL*, a basic expanded type. The use of *WINDOW* as actual generic parameter to *LIST* provides an example of case 5. → *See chapter 12 about generically derived types.*

The following class beginning uses types in its two Parent parts (case 4):

```
class DISPLAY_STATE inherit
        LIST [WINDOW]
        INPUT_MODE
        …
```

An example of case 6 is the use of type *ADDABLE* in a class text starting with

```
class MATRIX [G –> ADDABLE] ...
```

which states that any actual generic parameter must conform to *ADDABLE* (which means roughly that it must be based on a descendant of that class).

An example of case 7 is the Creation_instruction

```
create {WINDOW} a.set (x_corner, y_corner)
```

→ *See chapter 20 about creation instructions and expressions.*

which creates a direct instance of *WINDOW*, initializes it using a call to *set* with the given arguments, and attaches it to *a*. If *a* is of type *WINDOW* you may (and usually should) omit the {*WINDOW*} part; but it is useful if *a*'s type is a proper ancestor of *WINDOW* and you expressly want to create *a* s a *WINDOW*. Another example of case 7 s the Creation_expression in

```
screen.display (create {WINDOW}.set (x_corner, y_corner))
```

where we pass as argument to procedure *display* an object of type *WINDOW* created for the occasion. Here specifying the type is not an option but a necessity since, unlike the previous case, we don't have an entity *a* with a type declaration to serve as the default.

An example of case 8 is a multi-branch instruction

```
inspect
        last_exception
when {DEVELOPER_EXCEPTION} then
        fix_context ; retry
when {SIGNAL}, {NO_MORE_MEMORY then
        cleanup
end
```

appearing in this case in a Rescue clause to process exceptions. This states what to do depending on the type of *last_exception*.

An example of case 9 (similar in syntax to case 5, actual generic parameters) is the tuple type

```
TUPLE [REAL, INTEGER, RECTANGLE
```

which describes "tuples" — sequences of values— with at least three elements, the first of type *REAL* and so on. Finally an example of case 10 is

```
{RECTANGLE} ~ rotate (90)
```

an agent expression denoting a partially specified operation, ready to call *rotate* (assumed to be a procedure of *RECTANGLE*, with a single argument representing an angle) to rotate any rectangle by 90 degrees.

An example of case 10 is an agent expression of the form

```
agent {RESULT_TYPE}. your_function
```

denoting an object that represents the function *your_function*, ready to be called on arbitrary targets of type *TARGET_TYPE*.

11.4 HOW TO DECLARE A TYPE

The basis of the type system is the notion of class: every type is, directly or indirectly, based on a class, which provides the principal information for determining how instances of the class will look like. But classes are only the starting point of a whole set of type mechanisms that afford you considerable flexibility:

→ *See "BASE CLASS, BASE TYPE", 11.6, page 236 below.*

- Certain types are **reference** types, meaning that their values are references to objects, as opposed to **expanded** types, whose values are the objects themselves. The reference-expanded distinction is studied in detail in subsequent sections.
- Certain classes, said to be **generic**, do not directly describe a type; instead, they describe a type pattern, with one or more variable parts that must be filled in, through a "generic derivation", to yield an actual type. For example the class *LIST* [*G*] describes lists of elements of an arbitrary type, denoted in the class by *G*.
- Within the text of a generic class such as *LIST*, the **formal generic parameters** such as *G* themselves represent types (the possible actual generic parameters). The class may for example introduce an attribute of type *G*, or a routine with an argument or result of type *G*. Syntactically, then, a formal generic parameter is a type, although the exact nature of that type is not known in the class itself; only when a generic derivation provides the corresponding *actual generic parameter* (such as *WINDOW* above) can we know what *G* represents in that case.
- Finally, you may declare an entity *x* in a class *C* by using an **anchored** type of the form **like** *anchor* for some other entity *anchor*. This mechanism avoids tedious redeclarations since it ties the fate of *x*'s type to that of *anchor*: in *C*, *x* is treated as if you had declared it with the type used for the declaration of *anchor*; if a proper descendant of *C* redeclares *anchor* with a new type, *x*'s type will automatically follow.

Here is the syntactical specification covering all the possibilities.


```
Type ≜ Class_type |
       Class_type_expanded |
       Class_type_reference |
       Formal_generic_name |
       Anchored | Tuple_type
Class_type ≜ Class_name [Actual_generics]
Class_type_expanded ≜ expanded Class_type
Class_type_reference ≜ reference Class_type
Anchored ≜ like Anchor
Anchor ≜ Identifier | Current
```

→ Tuple_type *is defined in the chapter on tuples, page 274.*

→ Actual_generics *describes a list of types. The specification is on page 256 as part of the discussion of genericity in the next chapter.*

The first category, Class_type, covers types described by a class name, followed by actual generic parameters if the class is generic. The class name gives the type's base class. If the base class is expanded, the Class_type itself is an expanded type; if the base class is non-expanded, the Class_type is a reference type.

A class is an "expanded class" if its Class_header *begins with* **expanded class**, *and a non-expanded class otherwise.*

The second variant, Class_type_expanded, is written

```
expanded CT
```

for some Class_type *CT*. The result is an expanded type, even if *CT* was a reference type. The base class is *CT*, so that both types *CT* and **expanded** *CT* have the same *direct instances*, but they may have different *values*: references to such instances in one case, the instances themselves in the other.

The next variant ensures the reverse property: using

```
reference CT
```

guarantees that the corresponding values are references to instances of *CT*, even if *CT* is expanded. Here too the direct instances of both types are the same.

The fourth syntactical form, Formal_generic_name, covers the formal generic parameters of a class. If *C* has been declared as

```
... class [...,G,...] ...
```

then, within the text of *C*, *G* denotes a type. As noted, you cannot know the precise nature of this type just by looking at class *C*; *G* represents whatever actual generic parameter is provided in a particular generic derivation.

The next category, Anchored types of the form **like** *anchor*, accounts for anchored declarations.

Tuple_type, the last category, covers types of the form *TUPLE* [*X*, *Y*, …], whose instances are tuples: finite sequences of values of which the first must be of type *X*, the second of type *Y* and so on.

→ *Chapter 13 discusses tuples.*

The rest of this chapter examines these type categories, except for the generic and tuple mechanisms which have their own chapters.

11.5 TYPE SEMANTICS

The semantics of a type *T* is defined by its *direct instances* and its *values*:

- The **direct instances** of *T* are objects that may be created, at run time, by creation instructions using *T* as "creation type".
- The **values** of *T* are the values that entities declared of type *T* may take during execution.

→ *The* **instances** *of T include its direct instances and those of conforming types. See "Instance, direct instance of a class", page 402.*

For each kind of type introduced in this chapter and the next two, the presentation will specify how to determine the corresponding direct instances and values.

One of the reasons for considering both instances and values in defining type semantics is the reference-expanded distinction:

- The values associated with a **reference type** are references to potential objects — instances of the type — created at run-time through explicit creation instructions.
- The values of an **expanded type** are not references to objects but the objects themselves, which do not require creation instructions.

(*Direct or indirect.*)

This distinction only affects the *values* of a type, not its *direct instances*. We will see for example that if *CT* is a reference type you may define an expanded type as **expanded** *CT*. Both types then have the same direct instances, but the values are different: references to such instances in the first case; the instances themselves in the second.

So if all we had were expanded types, we wouldn't need to distinguish between values and direct instances.

Expanded types include as a special sub-category the basic types: *BOOLEAN*; *CHARACTER*; *INTEGER* and its sized variants *INTEGER_8*, *INTEGER_16*, *INTEGER_32* and *INTEGER_64*; *REAL* and its sized variants *REAL_32* and *REAL_64*; *DOUBLE*; and *POINTER*, covering addresses of features to be passed to external (non-Eiffel) routines.

→ *On POINTER and how to pass feature addresses to external routines see 28.8, page 646.*

How should you choose between reference types and expanded types? Here are a few general guidelines:

- Basic types aside, reference types are the most frequently used in typical Eiffel applications because of their flexibility: dynamic object creation allows developers to produce objects when and only when they need them; and reference semantics supports sophisticated data structures whose elements are chained to each other. In particular, reference types are the only possibility for structures that may be cyclic, such as a circular chain; this is why the Expanded Client rule specifically prohibited any cycles in the expanded client relation.

← *The Expanded Client rule was given on page 125.*

- Expanded types are useful when you want to avoid run-time indirections, since the entities will give you access to objects directly rather than through references. As noted, they also cover basic types such as *INTEGER*; clearly, an entity of integer type should give us an integer value, not a reference to a dynamically allocated cell that contains an integer.

Besides this reference-expanded distinction, the type system offers other variants of the notion of type:

- You may define a type by **anchoring**, as **like** *something*, tying it to the type of an entity, so that it will follow any redefinitions in descendants. Anchoring is covered later in this chapter.
- A type may also be a Formal_generic_name representing a formal generic parameter of the enclosing class; it then serves as a placeholder for any type (reference or expanded) that is used in a generic derivation. The whole generic mechanism will be discussed in the next chapter.

Although anchoring and genericity will add expressive power, the resulting semantics — the types' values and direct instances — will always in the end come down to one of the two fundamental categories: reference or expanded.

11.6 BASE CLASS, BASE TYPE

A type *T* of any category always proceeds, directly or indirectly, from a class, called the **base class** of the type. This rule suffers no exception.

If *T* is a Class_type the connection is direct: *T* is either the name of a non-generic class, such as *PARAGRAPH*, or the name of a generic class followed by Actual_generics, such as *LIST* [*WINDOW*]. In both cases the base class of *T* is the class whose name is used to obtain *T*, with any Actual_generics removed: *PARAGRAPH* and *LIST* in the examples.

For other categories of type the derivation from a class will be indirect, but just as clear. For example if *T* is an Anchored type of the form **like** *anchor*, and *anchor* is of type *LIST* [*WINDOW*], then the base class of that type, *LIST*, is also the base class of *T*.

A general property applies to the base class and base type:

Base rule

The **base type** of any type is a Class_type.

The **base class** of any type which is not a Class_type is (recursively) the base class of its base type.

The **direct instances** of a type are those of its base type.

The **values** of a type are its instances if the type is expanded, references to such instances if it is a reference type.

Why are these notions important? Many of a type's key properties (such as the features applicable to the corresponding entities) are defined by its base class. Furthermore, class texts almost never directly refer to classes: they refer to *types* based on these classes. For example, assuming that *C* is generic:

- If *D* is an heir of *C*, the Inheritance part of *D* will list as Parent not *C*, but a type of the form *C* [*ACTUAL1*, …].
- To describe objects to which *C*'s features are applicable, *D* will declare an entity *e* using not *C* but, again, a type generically derived from *C*.

A class text may refer to a class rather than a type in only three cases: the beginning of the class declaration, as in **class** *YOUR_CLASS_NAME …; a* Clients *part (syntax page 135); and a* Precursor *construct (syntax page 214).*

In such situations (and all other uses of types listed earlier) the base class provides the essential information: what features are associated with *C*. In the first example, they give the list of features that *D* inherits from *C*; in the second, they provide the features which *D* may call on *e*.

As for the base type, besides its role in defining the base class, it appears in many of the conformance rules, and determines what kind of object a creation instruction or expression will produce at run time.

→ *Conformance: chapter 14; creation: chapter 20.*

Clearly, you may only build a class type, generically derived or not, if the base class is a class of the universe:

Class Type rule *CTCT*

An Identifier *CC* is valid as the Class_name part of a Class_type if and only if it is the name of a class in the surrounding universe.

The class of name *CC* will be the type's base class.

The Base rule simplifies the presentation of type semantics. For every kind of type reviewed in this chapter and the next two we must specify the type's semantics, by stating what are the type's direct instances and its values. Thanks to the Base rule the process is straightforward:

As soon as we know *T*'s base type, and its actual generic parameters if any, we will know its **direct instances**: those of its base type, determined by the rules on type instances. If *T* is not a class type, we will know from the Base rule that its **base class** is the base class of *T*'s base type (itself a Class_type). Finally, the **values** of *T* will be its instances if it is an expanded type, otherwise references to such instances.

→ *Chapter 19.*

Type Semantics rule

To define the semantics of a type *T* it suffices to specify:

1 • Whether the type is expanded or reference.

2 • What is *T*'s base type.

3 • If *T* is a Class_type, what is its base class.

→ *For* Formal_generic_name *types the expanded/reference status depends on each generic derivation. See "SEMANTICS OF GENERIC TYPES", 12.7, page 264.*

In application of the Type Semantics rule, every presentation of a new kind of type in this chapter and the next two has a SEMANTICS paragraph that simply defines the base type (item 2 above), the base class in the case of a Class_type (3), and whether it is expanded or reference (1).

11.7 CLASS TYPES WITHOUT GENERICITY

We start our exploration of the type categories with the simplest way of defining a type: using a class without generic parameters.

In this case there is no difference between class and type. Assume for example a class text of the form

```
class PARAGRAPH feature
        first_line_indent: INTEGER;
        other_lines_indent: INTEGER;
        set_first_line_indent (n: INTEGER) is
                ... Procedure body omitted ...
        ... Other features omitted ...
end -- class PARAGRAPH
```

Then a class of the same universe (including *PARAGRAPH* itself) may use *PARAGRAPH* as a type, for example to declare entities.

Here *PARAGRAPH* is declared as a non-expanded class, so the corresponding type is a reference type. At run-time, entities of that type represent references which, if not void, are attached to instances of *PARAGRAPH*, obtained through creation instructions.

If class *PARAGRAPH* had been declared a **expanded class** …, then the resulting type would be expanded. In the general case:

Semantics of a non-generic class type

A non-generic class *C* used as a type (of the Class_type category) has the same expansion status as *C* (i.e. it is expanded if *C* is an expanded class, reference otherwise). It is its own base type and base class.

→ *The generic version will be only slightly different: "Semantics of a generically derived class type", page 264.*

These are not fascinating notions yet, but we must define a base class and base type for every type, and they will get less trivial as we move on.

PARAGRAPH, used as a type, is its own base type and its own base class. Values of type *PARAGRAPH* are references to instances of the class. Clients of the class may call exported features such as *first_line_indent* and others on entities of type *PARAGRAPH*.

→ *See chapter 23 about calling features on entities.*

There is no constraint on a non-generic Class_type other than the Class Type rule: the Identifier used must be the name of a class of the universe.

11.8 USING EXPANDED TYPES

The next few sections describe expanded types (Class_type_expanded and basic types). Before looking at the details, it is useful to recall when expanded types are useful.

In most systems, the vast majority of types used are reference types, based on non-expanded classes similar to the last examples (*LIST, WINDOW, PARAGRAPH*…). This is because reference types offer two major advantages, previewed at the beginning of this chapter:

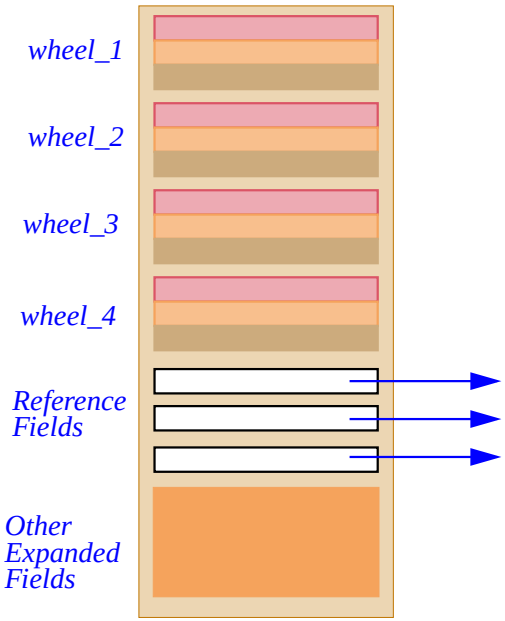
- The dynamic creation of objects provides a high degree of flexibility, a key attraction of the object-oriented method. Your systems will create objects if they need them, when they need them, and as many of them as they need.
- Reference types are indispensable for any data structure that involves chained elements and the possibility of cycles.

Expanded types, for their part, allow you to describe composite objects — objects with sub-objects — as previewed in an earlier chapter. Some of the (non-exclusive) cases justifying the use of composite objects are:

← *A first view of expanded types appeared in 7.5, page 123 which introduced composite objects and sub-objects.*

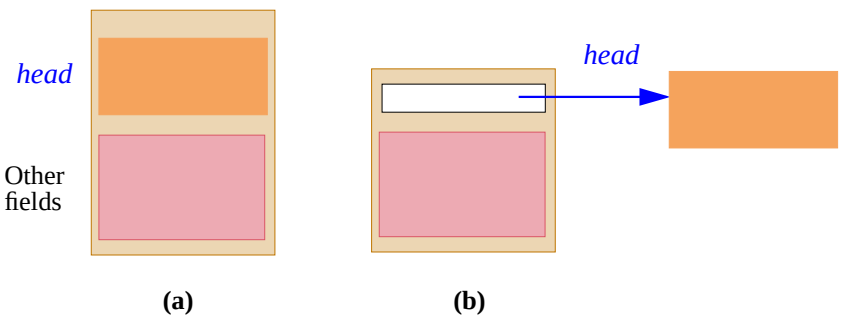
- Realism in modeling external world objects.
- Possible efficiency gain.
- Basic types.
- Interface with other languages.
- Machine-dependent operations.

The first case arises when we use Eiffel objects to model external world objects which are composite, rather than containing references to other objects. For example, in a Computer-Aided Design application, we may view a car as containing, among others, four "wheel" sub-objects, rather than four references to such objects. Such a decision, illustrated on the following figure, is only legitimate for objects which may never share sub-objects: in this example, a wheel may not be part of two different cars.



Sub-object vs. reference to another object

The second reason is, in some circumstances, a gain in efficiency: composite objects save space (by avoiding pointers) and time (by avoiding indirections). For example, if every instance of *PERSON* has a *head*, declaring *head* of an expanded type will give the structure illustrated by **(a)** on the next figure, avoiding the indirection of **(b)**. Here again, this only applies because there is no sharing of sub-objects, at least if we exclude the case of Siamese twins.



Composite car object

You must realize, however, that the possible efficiency gain is not guaranteed. The last two figures, and similar illustrations of expanded attributes and composite objects, are only conceptual descriptions, not implementation diagrams. (Unlike other languages that shall remain nameless here, Eiffel is specified in terms of the abstract properties of software execution, not by prescribing a certain implementation.) The authors of an Eiffel compiler or interpreter may choose any representation they wish as long as they guarantee the *semantics* of expanded values, according to which (as explained in the discussion of reattachment in a later chapter) an assignment *x* := *y* must copy the object attached to *y* onto the object attached to *x*, and an equality test *x* = *y* must compare the objects field by field.

→ *Chapter 22*.

Both the time and space gains are important in the case of basic types such as integers or characters; to manipulate the value *3*, we should not need to allocate an integer object dynamically, or to access it through a reference. For that reason, basic types are described by expanded classes of the Kernel Library, as explained in a later section.

→ *See 11.11, page 244, about basic types.*

Another opportunity for expanded types may be the need to keep data structures produced and handled by software elements written in other languages. An example might be control information associated with a database management system, which Eiffel routines will not manipulate directly, but pass back and forth to foreign (non-Eiffel) routines. As you have no control over the format and size of such data structures, the best way may be simply to keep them as sub-objects within your Eiffel objects.

11.9 CLASS TYPES EXPANDED

How do we obtain an expanded type from a class?

The class types seen so far may or may not be expanded:

- A Class_type whose base class is expanded is itself an expanded type; values of that type are objects (instances of the type).
- A Class_type whose base class is not expanded is a reference type; values are references to potential objects, created dynamically.

In some cases we may also need to produce an expanded type from a non-expanded class. Assume for example that *HEAD* is a non-expanded class. In *PERSON*, you may declare the attribute

```
head: HEAD
```

But this will give you the **(b)** variant of the last figure, not **(a)**.

You could achieve **(a)** by introducing a special expanded class *EXPANDED_HEAD* just for this purpose and declaring *head* to be of type *EXPANDED_HEAD*. The class declaration, using inheritance, is trivial:

```
expanded class EXPANDED_HEAD inherit HEAD end
```

Remember that by default the export policy of *HEAD* will be passed on to *EXPANDED_HEAD*, and that the expansion status of a class (whether it is declared as **expanded class**) does not affect that of its heirs. (It only affects the semantics of entities declared of the corresponding class types.)

← *See 7.11, page 130, about the influence of inheritance on the export status, and 6.12, page 114, on the non-transmission of expansion status to heirs.*

Although it produces the desired result, this technique requires introducing extra classes such as *EXPANDED_HEAD*, whose only role is to provide a base for expanded types. This can become tedious.

The notion of Class_type_expanded solves the problem. If you only use *EXPANDED_HEAD* to declare *head* and a few other entities, you can avoid introducing the class altogether, by declaring the entities under the form

```
head: expanded HEAD
```

The direct instances of a Class_type_expanded **expanded** *CT*, where *CT* is a Class_type, are the same as those of *CT*; its possible values are direct instances of *CT* (rather than references to such objects).

It is not an error to use the type **expanded** *T* if the base class of *T* is already an expanded class; such a type is simply equivalent to *T*. For example, using the Kernel Library expanded class *INTEGER*, the following two declarations are equivalent:

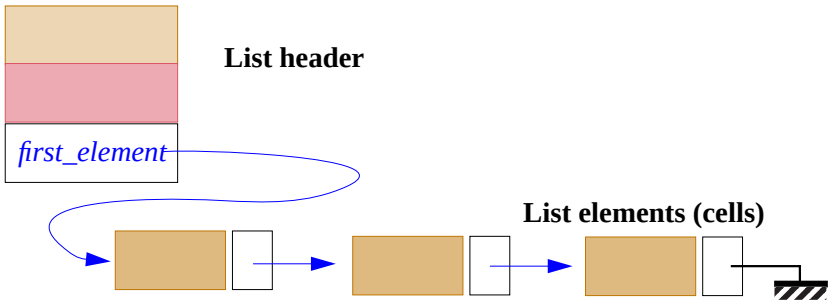
```
n: INTEGER
n: expanded INTEGER
```

The second form is useless in this case; but it is useful to know that **expanded** *T* will always have the intended effect, whether *T* is a reference type or already expanded.

This technique for obtaining expanded types by adding **expanded** to a Class_type also works for generically derived types. So you may declare

```
pls: expanded LINKED_LIST [PARAGRAPH]
```

The expansion only applies to the object corresponding to *pls*; there is no recursive expansion of the data structure. An object of type *LINKED_LIST* (a class from EiffelBase) is not an entire list but a list header, with references to list cells; an example is the object marked **List header** on the following figure. Such an object contains references to actual list cells, which of course will not be expanded. So the only effect of the above declaration is to make the value of *pls* be an object such as the illustrated list header, rather than a reference to such a header.



List header and list cells

This is a repetition of the figure on page 185.

A Class_type_expanded of the form **expanded** *T* is its own base type; its base class is the base class of *T*.

11.10 VALIDITY OF EXPANDED TYPES

Expanded types have specific properties, already previewed. First we must know precisely when a type is “expanded” and when it is “reference”:

Expanded type, reference type

A type *T* is **expanded** if and only if one of the following two conditions holds:

1 • *T* is a Class_type whose base class *C* is an expanded class.

2 • *T* is of the form **expanded** *CT*.

T is a **reference type** if neither of these conditions applies and *T* is not a Formal_generic_name.

In case 2 it is, as noted earlier, redundant but not erroneous for CT itself to be expanded.

This definition characterizes every type as either reference or expanded, except for one case mentioned in the last clause: formal generic parameters. A Formal_generic_name stands for any type to be used as actual generic parameter in a generic derivation; since an actual generic parameter may be expanded as well as reference, it would be premature to classify the Formal_generic_name in either of these two categories.

→ *Genericity is discussed in the next chapter.*

In the case of expanded types, three rules apply:

- One has already been seen: the Expanded Client rule, which prohibits cycles in the client relation between expanded classes.

← *Page 125.*

- Next, the base class of an expanded type may not be *deferred*, since deferred classes are not fully implemented. Assume in the last example that instead of *LINKED_LIST* we had attempted to use *LIST*, a deferred ancestor. *LIST* [*T*] has, among others, a deferred feature *last* of type *T*, which descendants may choose to implement (to *effect*) as either an attribute or a function. Then it would be impossible to compile a client class that uses an attribute such as *pls*, since we do not know how many fields to set aside for the attributes of *pls*.

- Finally, the base class of an expanded type must retain *default_create* — the default initialization procedure coming, after possible renaming or redefinition, from the universal class *ANY* — as one of its creation procedures. The reason is that initializing an object with sub-objects, such as the one illustrated on the figure page 240, requires initializing all its sub-objects, for which all that's available is the standard initialization provided by *default_create*.

  In the simplest case this requirement is automatically met: a class that doesn't have a Creators part (that is to say, doesn't explicitly list creation procedures) is considered to have *default_create* as its sole creation procedure. The details appear in the discussion of creation.

→ *"THE CASE OF EXPANDED TYPES", 20.8, page 428.*

Here is the validity rule formalizing the last two constraints:

Expanded Type rule *CTET*

It is valid to use an expanded type of base class *C* in the text of a class *B* if and only if it satisfies the following two conditions:

1 • *C* is not a deferred class.

2 • *C*'s version of the procedure *default_create*(inherited from *GENERAL*) is one of the creation procedures of *C*, available to *B* for creation.

The reason for the last part of clause 2 is that creation procedures, as discussed in the chapter on creation, might be "available for creation" to some clients only. The one that matters here is *B*.

→ *"RESTRICTING CREATION AVAILABILITY", 20.7, page 425.*

11.11 BASIC TYPES

An important case of expanded types is a collection of **basic types** covering simple values:

→ *Detailed in chapter 37.*

- *BOOLEAN*, describing boolean values (true and false).
- *CHARACTER*, describing single characters.
- *INTEGER* (32-bit integers) and its variants supporting specific sizes: *INTEGER_8*, *INTEGER_16*, *INTEGER_64*.
- *REAL*: 32-bit floating-point numbers.
- *DOUBLE*: 64-bit floating-point numbers.
- *POINTER*, serving to pass addresses of Eiffel features and expressions to non-Eiffel routines.

  Three types also enjoy special properties but are not considered basic types: *ARRAY*, *STRING* and tuple types.

→ *See chapters 33 about ARRAY and STRING and 13 about tuples.*

These types are covered by classes in the Kernel Library, and compilers know about them, permitting extensive optimizations of arithmetic operations. But this is only for purposes of efficient implementation: semantically, the basic types are just like other class types:

Semantics of basic types

The basic types *BOOLEAN*, *CHARACTER*, *INTEGER*, *INTEGER_8*, *INTEGER_16*, *INTEGER_64*, *REAL*, *DOUBLE* and *POINTER* are all specimens of Class_type, defined by expanded classes. As a result, each is expanded, and is its own base type and base class.

The instances — which are the same as the values — are specified in the chapter on basic types. Respectively: the two boolean values True and False; character values; integers represented with the appropriate bit size; 32-bit floating-point numbers; 64-bit floating point numbers; addresses.

→ *Chapter 37*.

The basic types will need some special conformance properties. In general, a type *U* conforms to a type *T* only if *U*'s base class is a descendant of *T*'s base class. But then *INTEGER*, for example, is not a descendant of *REAL*. Since mathematical tradition suggests allowing the assignment *r* := *i* for *r* of type *REAL* and *i* of type *INTEGER*, the definition of conformance will include a small number of special cases for basic types.

→ *"EXPANDED TYPE CONFORMANCE", 14.9, page 294*.

Except for *POINTER* which has no exported feature of its own, each of the basic classes describes the operations applicable to values of the corresponding type (booleans, characters etc.). For compatibility with traditional arithmetic notation, many of the feature identifiers are Prefix or Infix.

11.12 CLASS TYPES EXPLICITLY SPECIFIED AS EXPANDED

The semantics of a Class_type_expanded — of the form **expanded** *CT* for some class type *CT* — follows from previous discussions:

Semantics of a Class_type_expanded

A Class_type_expanded of the form **expanded** *CT* is an expanded type. Its base type is *CT*.

As a consequence of the Base rule, the type's direct instances are the same as those of *CT*; they also are the type's values. Although in most of this chapter we restrict ourselves to the non-generic case, the rule also applies to a generically derived *CT*.

Because such a type is not a Class_type, the Type Semantics rule tells us that we don't need to specify the base class explicitly: it is the base type's own base class. This applies to all the type categories that remain to be seen in this chapter and the next two. Here in **expanded** *PARAGRAPH* the base class is *PARAGRAPH*, and in **expanded** *LINKED_LIST* [*INTEGER*] the base class is *LINKED_LIST*.

←*Page 238*.

11.13 CLASS TYPES EXPLICITLY SPECIFIED AS REFERENCE

For symmetry with Class_type_expanded, of the form **expanded** *CT*, the syntax also permits Class_type_reference, of the form

reference *CT*

This notation is useful in particular to give us reference versions of the basic types. Usually, as noted, we want to manipulate integers, not references to integers. But occasionally we may want to manipulate entities of type **reference** *INTEGER*, or **reference** *REAL* and so on. This is useful in particular if we have a polymorphic constainer of references, such as an *ARRAY* [*NUMERIC*] or a *LIST* [*ANY*], where some of the entries are references to real numbers (or other objects of basic types), and others are references to non-basic objects, such as matrices.

A specific constraint applies to the base type

Class Type Reference rule *CTCR*

A Class_type_reference **reference** *T* if and only if its base Class_type is neither of the Kernel Library classes *INTEGER_GENERAL* and *REAL_GENERAL*.

This is an ad hoc rule justified by the special needs of matching the purely O-O types system with the traditional properties of integer and floating-point arithmetic.

The semantics of a Class_type_reference directly parallels the previous case, Class_type_expanded:

Semantics of a Class_type_reference

A Class_type_reference of the form **reference** *CT* is a reference type. Its base type is *CT*.

As in the Class_type_expanded case it follows from the Base rule that the type's direct instances are the same as those of *CT*, its values are references to instances of *CT*, and the base class is that of *CT*. The rule applies to a generically derived *CT* as it does to the non-generic case.

11.14 ANCHORED TYPES

The originality of an Anchored type, the last category in this chapter, is that it carries a provision for automatic redefinition in descendants of the class where it appears.

An Anchored type is of the form

like *anchor*

where *anchor* is, predictably, called the type's **anchor**. The anchor must be either an entity, or *Current*. If an entity, *anchor* must be the final name of a feature of the enclosing class, or, in the text of a routine, a formal argument.

A declaration using an Anchored type is an **anchored declaration**, and the entities it declares are **anchored entities**.

Anchored types avoid "redefinition avalanche". As long as what you only consider what happens in a class *C*, declaring an entity of type **like** *anchor* in *C* is the same as declaring it of the same type as *anchor*, say *T*. The difference comes from inheritance: if any descendant of *C* redefines the type of *anchor* to a new type (conforming to *T*), it will be considered to have also redefined all the entities anchored to *anchor*.

Since it is quite common to have a group of related entities that must keep the same type throughout their redefinitions, anchored declaration is essential to the smooth functioning of the type system. Without it we would constantly be writing lots of new declarations serving no other purpose than type specialization.

Anchored examples

We already encountered anchored declarations in the discussion of redeclaration; the example was that of a routine in the Data Structure Library class *LINKED_LIST*:

→ *The encounter was towards the end of 10.9, page 185.*

put_element (*lc*: **like** *first_element*; *i*: *INTEGER*)

whose argument *lc* represents a list cell. This declaration "anchors" *lc* to *first_element*, a feature of the class declared of type *LINKABLE* [*G*] (the type representing list cells). As a result, *lc* itself is considered in *LINKED_LIST* to have the same type as *first_element*, *LINKABLE* [*G*]. Because *lc* has been anchored to *first_element*, any descendant of *LINKED_LIST* which redefines *first_element* to a new type, taking into account more specific forms of list cells (such as cells chained both ways, or tree nodes), does not need to redefine *lc* and all similar entities of the class: their types will automatically follow the redeclared type of their anchor, *first_element*.

Anchoring is often useful for arguments of "set" procedures. If class *EMPLOYEE* has an attribute *assignment* of type *EMPLOYEE_ASSIGNMENT*, and an associated procedure

```
set_assignment (a: EMPLOYEE_ASSIGNMENT) is
        -- Make a the employee's current assignment.
    require
        exists: a /= Void
    do
        assignment := a
    ensure
        set: assignment = a
    end
```

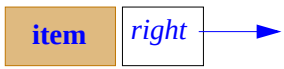
*Warning: this is not the recommended style — see text.*pl

it is usually be preferable to use the type **like** *assignment* to declare the argument *a*. Within the given class, the effect is the same, since *assignment* is of type *EMPLOYEE_ASSIGNMENT*; but if a descendant redefines *assignment* to a more specific type — such as *ENGINEERING_ASSIGNMENT* — the signature of the procedure *set_assignment* will automatically follow.

Anchoring to *Current*

You may use *Current* as anchor. Declaring *x* of type **like** *Current* in a class *C* is equivalent to declaring it of type *C* in *C*, and redeclaring it of type *D* in any proper descendant *D* of *C*.

Among other advantages, this technique avoids lengthy redefinitions. *LINKABLE*, mentioned earlier, relies on it. A list cell has a reference to its right neighbor:



Linkable list cell

This figure and the next appeared previously on page 186.

The attribute *right* denotes that reference in class *LINKABLE*, where it is anchored to *Current*:

```
right: like Current
```

This declaration guarantees that in any more specialized version of *LINKABLE*, described by a proper descendant of class *LINKABLE*, *right* will automatically denote to objects of the descendant type. An example is class *BI_LINKABLE*, representing elements chained both ways:

Bi-linkable list cell

In this case the anchored declaration guarantees that a doubly linked list element is only used in conjunction with other elements of the same (or a more specialized) type. Another descendant of *LINKABLE* is a class describing tree nodes; here too, the anchoring guarantees that tree nodes only refer to other tree nodes, not to simple *LINKABLE* elements.

Anchoring to an argument

The third possibility for an *anchor* besides *Current* and the name of a feature is, in a routine, one of the formal arguments. This is useful in particular for guaranteeing that two arguments of a routine, or an argument and the result (for a function) must have compatible types. The feature *clone* coming from the universal class *ANY* (and hence present in all classes) provides a good example:

→ *See 21.3, page 452 on clone.*

```
frozen clone (other: ANY): like other is
            -- Void if other is void; otherwise, new object
            -- field-by-field identical to object attached to other.
        …
```

This function produces a fresh copy of the object passed as argument. Since *clone* is applicable to objects of arbitrary type, the type declared for the argument *other* may only be *ANY*. Then without anchored types we would be reduced to using *ANY* as result type; but this would make normal uses of *clone* impossible, as in

clone returns void if its argument is void (not attached to an object).

```
x, y: SOME_TYPE
…
        -- Create an object and attach it to y:
create y …
x := clone (y)
```

where the last assignment would violate the type conformance rules since the source is of type *ANY*, which doesn't conform to the target's type, *SOME_TYPE*.

→ *The conformance rules are in chapter 14.*

In principle we could use *clone* not with an assignment as above, but with an Assignment_attempt, which here would succeed. But this would be inappropriate: assignment attempt is for cases in which the author of a class cannot guarantee the type of an object statically (for example with a persistent object retrieved from a file or database), not for cases such as the above which shouldn't present any type difficulty.

→ *"Assignment attempt", , page 488, and subsequent sections.*

Anchored declaration solves the problem; without this mechanism we would have to redefine the function in every class.

If, like *Clone*, a routine has a formal argument or result anchored to another of its formal arguments, the type rules on routine calls will require that, in a call, the type of the associated actual argument or result conform to the type of the actual argument corresponding to the anchor. For example, in

```
u: = clone (v)
```

the type *v* must conform to that of *u*. This also applies to the case of an argument anchored to another: with

```
reset_from_model (model: SOME_TYPE; target: like model)
```

a call of the form *reset_from_model* (*a*, *b*) requires that the type of *b* conform to the type of *a*.

The precise rules will appear in the study of conformance and reattachment.

→ *"EXPRESSION CONFORMANCE", 14.12, page 299*.

Anchoring to an expanded or generic

In **like** *x* where *x* is a query or argument, there is no particular restriction on the type *T* of *x*. In the most common case *T* will be a reference type, but it may also be anything else, such as:

- An anchored type itself — under a no-cycle requirement explained below.
- An expanded type.
- A Formal_generic_name representing a generic parameter of the enclosing class.
- A Tuple_type.

The expanded case is not very exciting because redefinition possiblities are very limited for the anchor. It enables you, however, to emphasize that a group of expanded entities must have the same type, and facilitates switching between reference and expanded status if you don't get the first time around. [NOTE: NEXT TWO SECTIONS WILL PROBABLY BE REMOVED.]

→ *As a consequence of "Redeclaration rule", page 223 and "Direct conformance: expanded types", page 296*.

The formal parameter case is more subtle. If *x* is of type *G* in a class *C* [*G*], **like** *x* denotes the actual generic parameter corresponding to *G*. Declaring *y*: **like** *x* has, within the text of *C*, the same effect as declaring *y* of type *G*. With *z* of type *C* [*T*] for some type *T*, the rules on genericity imply that *z*.*y* has type *T*. If *C* has a feature *f* (*u*: **like** *x*), a call *z*. *f*(*v*) will be valid only if the type of *v* is exactly *T* — not another type conforming to *T*, as would be valid if *u* was declared just with the type *G*.

→ *"Generic Type Adaptation rule", page 268*; *see also "THE TYPE OF AN EXPRESSION", 26.13, page 606*.

The same spirit guides the interpretation of **like** *t*, where *t* is of a tuple type such as *TUPLE* [*A, B, C*]. If *u* is declared as *TUPLE* [*A, B, C*], the conformance rules on tuple types let us assign to *u* not only a tuple such as [*a1, b1, c1*] (with *a1* of type *A* and so on) but also a longer tuple such as [*a1, b1, c1, d1, e1*] as long as the initial items are of the requisite types (*A*, *B* and *C* respectively). But with *u* of type **like** *t*, only a tuple of exactly three elements will be permissible. This means that you can have your choice between a lax interpretation of tuple types (tuples of *n* items or more, for some *n*) and a restrictive one (tuples of exactly *n* items). The strict interpretation will be useful in particular for routine agents.

→ *"TUPLE TYPE CONFORMANCE", 14.10, page 297. On the rules for agents, see*

Avoiding anchor cycles

To go from the preceding informal presentation of anchored types to their precise constraint and semantics requires that we address the issue of anchor chains: do we allow a type **like** *anchor* if *anchor* is itself anchored, of type **like** *other_anchor*?

Although most developments do not need anchor chains, they turn out to be occasionally useful for advanced applications, so the answer will be yes. But then of course we must make sure that an anchor chain is meaningful, by exclusing cycles such as *a* declared as **like** *b*, *b* as **like** *c*, and *c* as **like** *a*.

Because of genericity, the cycles might in fact be not directly through the anchors but through the types they involve, as with *a* of type *LIST* [**like** *b*] where *b* is of type **like** *a*. Here we say that a type "involves" all the types appearing in its definition: *A* [*B, C, LIST* [*ARRAY* [*D*]]] involves *B*, *C*, *D*, *ARRAY* [*D*] and *LIST* [*ARRAY*[*D*]. We can define this notion in full generality:

Types involved in a type

The types **involved** in a type *T* are the following:

- If *T* is a generically derived Class_type: all the types (recursively) involved in any of the actual generic parameters.
- If *T* is a Class_type_reference or a Class_type_expanded: all they types (recursively) involved in its base type.
- In all other cases (including a non-generic Class_type, Formal_generic, Anchored): just *T* itself.

This allows us to state when an anchored declaration involves a cycle:

Anchor set; cyclic anchor

The **anchor set** of a type *T* is the set of entities made of the following elements:

- If *T* is an anchored type **like** *anchor*: its anchor *anchor*.
- (Recursively) the anchor set of the types involved in *T*.

An entity *a* of type *T* is a **cyclic anchor** if the anchor set of *T* includes *a* itself.

The basic rule, stated next, will be that if *a* is a cyclic anchor you may not use it as anchor: the type **like** *a* will be invalid.

Validity and semantics of anchored types

The notions just introduced enable us to define the validity of anchored types:

Anchored Type rule *CTAT*

An anchored type of the form **like** *anchor* appearing in a class *C* is valid if and only if it satisfies the following two conditions:

1 • *anchor* is one of: *Current*; the final name of a query (attribute or function) of *C*; a formal argument of the enclosing routine if any.

2 • *anchor* is not a cyclic anchor.

Other than the no-cycle requirement, the rule on anchors is liberal. In particular **an anchor's type may be expanded**, or a Formal_generic_name. Anchoring is of limited benefit in these cases, since the conformance rules leave little possibility of redeclaration for an entity of expanded or formal generic types. But an anchored declaration can cause no harm, and still has the benefits of clarity and concision.

Now for the semantics. When we declare *a* as being of type **like** *anchor* with *anchor* of type *T* we consider *a*, for all practical purposes — such as deciding what features are applicable to *a* — to be of type *T* too. So the base type of **like** *anchor* will be *T*, or more generally the base type of *T* (since we allow *T* itself to be **like** *other_anchor* or some other non-primitive type). So in

```
frozen clone (other: ANY): like other is … do … end
```

we may consider, within the function's body, that *Result* is of type *ANY*. Similarly, with

```
set_assignment (a: like assignment) is … do … end
```

where *assignment* is an attribute of type *EMPLOYEE_ASSIGNMENT*, we may treat *a*, within *set_assignment*, as being of that same type.

Here is the rule that formalizes these notions:

Semantics of an anchored type

The type **like** *Current* has the same expansion status as the enclosing class. Its base type is the current type.

The type **like** *anchor*, with an *anchor* — other than *Current* — of type *T*, has the same expansion status as (recursively) *T*. Its base type is (recursively) the base type of *anchor*.

As usual for types other than Class_type, *the base class is defined as the base type's own base class.*

The "**current type**", used in the **like** *Current* case, is the class name equipped with its generic parameters if applicable. So for a **like** *Current* declaration in class *PARAGRAPH* the base type is *PARAGRAPH*; in class *HASH_TABLE* [*G*, *KEY* –> *HASHABLE*] it is *HASH_TABLE* [*G*, *KEY*]. This notion will be discussed in the next chapter.

→ *"CURRENT TYPE, FEATURES OF A TYPE", 12.8, page 266*.

"Expansion status" means whether the type is expanded or reference. In the of anchoring to a Formal_generic_name, as with **like** *G* in a class *C* [*G*], we shall see that the expansion status of *G* depends on every particular generic derivation: it is the same as the expansion status of the corresponding actual generic parameter. The status of **like** *G* will follow.

→ *"SEMANTICS OF GENERIC TYPES", 12.7, page 264*.

The Anchored Type rule legitimates the use of a recursive definition of the above semantic rule. To determine the base type of **like** *anchor* we must look at the type of *anchor*, which might itself involve one or more types of the form **like** *other_anchor*, leading us to look at the type of *other_anchor* and so on. Because the Anchored Type rule requires *anchor* to be a non-cyclic anchor, this process will always terminate. This also applies to the process of determining whether the type is reference or expanded.

The rules on expression conformance will complete the definition of the semantics of anchored types.

→ *"EXPRESSION CONFORMANCE", 14.12, page 299*.

Anchored declaration is essentially a syntactical device: you may always replace it by explicit redefinition. But it is extremely useful in practice, avoiding much code duplication when you must deal with a set of entities (attributes, function results, routine arguments) which should all follow suit whenever a proper descendant redefines the type of one of them, to take advantage of the descendant's more specific context.