11 Types

11.1 OVERVIEW

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

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

11.2 THE ROLE OF TYPES

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

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

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

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

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

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

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

→ Chapter 25.

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

- The discussion of **conformance** will explain how a type may be used in lieu of another, and its instances in lieu of that other's instances.

→ Conformance is the topic of chapter14. Chapter 23 covers calls; on type checking, see chapter 25.

- 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.

11.3 WHERE TO USE TYPES

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

1 • To declare the result type of an attribute or function: construct Declaration_body. ← *Syntax: page 141.*

2 • To declare the arguments of a routine or inline agent: construct Formal_arguments, defined in terms of Entity_declaration_list. ← *Page 216; see also* Inline_agent, *p. 743.*

3 • To declare a local routine entity: construct Local_declarations (also defined in terms of Entity_declaration_list). ← *Page 221.*

4 • To indicate that a class has a certain parent: construct Parent, as part of Inheritance. ← *Page 169.*

5 • To specify actual generic parameters, as explained in the next chapter: construct Actual_generics. → *Page 342.*

6 • To specify a generic Constraint, also in the next chapter: construct Constraint, part of Formal_generics. → *Page 349.*

7 • To indicate an explicit creation type in a creation instruction or expression: construct Explicit_creation_type. → *Page 543.*

8 • To choose from a set of instructions, based on an expressions's type, in a Multi_branch. → Choice, *page 477:.*

9 • To specify the parameters (component types) of a Tuple_type. → *Page 364.*

10 •To declare the type of a target of a Call_agent. → Agent_actual, *page 744:*

11 •To specify target conversion for infix operators. → *Page 762:*

12 •To call a feature without a target, in a Non_object_call. → *Page 618.*

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

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

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

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

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

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

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

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

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

An example of case 7 is the Creation_instruction

→See chapter 20 about creation instructions and expressions.

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

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.type
when { DEVELOPER_EXCEPTION } then
      fix_context ; retry
when { SIGNAL }, { NO_MORE_MEMORY } then
      cleanup
end
```

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

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

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

which describes "tuples" — sequences of values— with at least three elements, the first of type *REAL* and so on.

An example of case 10 is

```
agent { RECTANGLE }.rotate (90)
```

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

An example of case ---- ADAPT --- is an instruction

```
if {x: EXPECTED_TYPE } retrieved_from_network then
      x.f
      …
else
      …
end
```

which determines whether a run-time object obtained from an outside source is of a certain predicted type.

Case 11 covers the ability to convert the result of an arithmetic operator to the type of the second operand, as in the following in class *INTEGER*

```
plus alias "+" convert { COMPLEX  }  (other: REAL): REAL
            … Definition of integer addition …
```

which dispatches *your_real + your_complex* to the feature with the same name in class *COMPLEX* (rather than the one specified here).

Finally, case 12 allows calls of the form { *T* } *some_feature* where *some_feature* is a feature of type *T* that doesn't need a target, for example a constant attribute.

11.4 HOW TO DECLARE A TYPE

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

→ See "BASE CLASS, BASE TYPE AND TYPE SEMANTICS", 11.7, page 324 below.

• 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.

Types

```
                 Type ≜ Class_or_tuple_type |
                        Formal_generic_name |
                        Anchored
  Class_or_tuple_type ≜ Class_type | Tuple_type
           Class_type ≜ [Attachment_mark]
                        Class_name
                        [Actual_generics]
      Attachment_mark ≜ "?" | "!"
             Anchored ≜ [Attachment_mark] like Anchor
               Anchor ≜ Feature_name | Current
```

→ Tuple_type *is defined in the chapter on tuples, page 364.*

→ Actual_generics *describes a list of types. The specification is on page 342 as part of the discussion of genericity in the next chapter.*

The most common and versatile kind is Class_type, covering types described by a class name, followed by actual generic parameters if the class is generic. The class name gives the type's base class. If the base class is expanded, the Class_type itself is an expanded type; if the base class is non-expanded, the Class_type is a reference type.

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

Class_or_tuple_type covers tuple types as well as class types. Tuple types, studied in their own chapter, are a kind of trimmed-down class type; *TUPLE* [*a*: *X*; *b*: *Y*; *c*: *Z*] acts like a class with three features *a*, *b* and *c* of the types given. You can omit the labels: *TUPLE* [*X*, *Y*, …] describes finite sequences of values of which the first must be of type *X*, the second of type *Y* and the third of type *Z*. Tuple types share a number of properties with class types, hence the first variant of Type, covering them both.

→ Chapter 13 discusses tuples.

An Attachment_mark **?** indicates that the type is **detachable**: its values may be void — not attached to an object. The **!** mark indicates the reverse: the type is **attached**, meaning that its values will always denote an object; language rules, in particular constraints on attachment, guarantee this. No Attachment_mark means the same as **!**, to ensure that a type, by default, will be attached.

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

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

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

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

Tuple_type, the last category, covers types of the form *T*

→ Chapter 13 discusses tuples.

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

11.5 INSTANCES AND VALUES

For each kind of type in the language, we must specify — along with associated syntax rules and validity constraints — the *semantics* of the type.

Defining the semantics of a type *T* involves answering two questions:

- What objects can be produced, during execution, from the description given by *T*?
- What are at run time the possible values of an entity or expression of type *T*?

The answers have precise names:

Direct instances and values of a type

The **direct instances** of a type *T* are the run-time objects resulting from: representing a manifest constant, manifest tuple, Manifest_type, agent or Address expression of type *T*; applying a creation operation to a target of type *T*; (recursively) cloning an existing direct instance of *T*.

The **values** of a type *T* are the possible run-time values of an entity or expression of type *T*.

→ See also "Type, generating type of an object; generator", page 498.

Specifying the direct instances might seem sufficient; the reason we also need to consider values is the difference between **expanded** and **reference** types. A type's values are objects in the first case, references to objects in the second.

Expanded types include as a special sub-category the basic types: *BOOLEAN*; *CHARACTER* and its sized variants such as *CHARACTER_8*; *INTEGER* and its sized variants such as *INTEGER_8* and *NATURAL_64*; *REAL* and its sized variants. *REAL_32* and *REAL_64*; and *POINTER*, covering addresses of features to be passed to external (non-Eiffel) routines. Clearly, an entity of integer type should give us an integer value, not a reference to a dynamically allocated cell that contains an integer.

→ *"PASSING THE ADDRESS OF AN EIFFEL FEATURE", 31.8, page 823*.

Reference provide more flexibility thanks to dynamic object allocation, allowing the execution to create objects when and only when it needs them; reference semantics, supporting linked data structures. Expanded types, for their part, are useful not only for basic types but also for describing *sub-objects* avoiding indirections. The role of expanded types, and the criteria for choosing between expanded and reference, are further studied below.

The notion of type has, besides the expanded-reference distinction, a number of variants detailed in the following sections:

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

In understanding type semantics, another useful notion is that of *instance*, complementing the notion of *direct* instance defined above:

Instance of a type

The **instances** of a type *TX* are the direct instances of any type conforming to *TX*.

Since every type conforms to itself, this is equivalent to stating that the instances of *TX* are the direct instances of *TX* and, recursively, the instances of any other type conforming to *TX*.

In the well-known example of an inheritance hierarchy with a class *FIGURE* at the top and descendants describing successively more specific geometrical figures, such as *CLOSED_FIGURE*, *POLYGON*, *RECTANGLE*, *SQUARE*, each inheriting from the preceding one, a direct instance of *SQUARE* is also an instance of all the others, including *SQUARE* itself.

This also illustrates that a deferred type such as *FIGURE*, which cannot have direct instances (since creation instructions of target *FIGURE* are invalid), may have instances if the class has effective descendants.

→ *"Creation and deferred classes", page 529*.

A semantic rule connects the notion of value and instance:

Instance principle

Any value of a type *T* is:

- If *T* is reference, either a reference to an instance of *T* or (unless *T* is attached) a void reference.
- If *T* is expanded, an instance of *T*.

Thanks to this rule, it suffices, when studying type semantics, to define the *direct* instances of each possible type. The instances follow immediately and — since the type's declaration indicates whether it is reference (and if so, attached) or expanded — so do the values.

11.6 INSTANCES OF A CLASS

Along with the instances, direct and indirect, of a *type*, it is convenient to talk about the corresponding notion for a *class*:

Instance, direct instance of a class

An instance of a class *C* is an instance of any type *T* based on *C*.

A direct instance of *C* is a direct instance of any type *T* based on *C*.

For non-generic classes the difference between *C* and *T* is irrelevant, but for a generic class you must remember that by itself the class does not fully determine the shape of its direct instances: you need a type, which requires providing a set of actual generic parameters.

11.7 BASE CLASS, BASE TYPE AND TYPE SEMANTICS

At its core, the notion of type in Eiffel proceeds from the notion of class. Indeed, we can bring down the properties of any type to those of an associated Class_or_tuple_type and, through it, to those of a class:

Base principle

Any type *T* proceeds, directly or indirectly, from a Class_or_tuple_type called its **base type**, and an underlying class called its **base class**.

The base class of a type is also the base class of its base type.

A Class_type is its own base type; an anchored type **like** *anchor* with *anchor* having base type *U* also has *U* as its base type. For a formal generic parameter *G* in **class** *C* [*G* –> *T*] … the base type is (in simple cases) the constraining type *T*, or *ANY* if the constraint is implicit.

The base class is the class providing the features applicable to instances of the type. If *T* is a Class_type the connection to a class is direct: *T* is either the name of a non-generic class, such as *PARAGRAPH*, or the name of a generic class followed by Actual_generics, such as *LIST* [*WINDOW*]. In both cases the base class of *T* is the class whose name is used to obtain *T*, with any Actual_generics removed: *PARAGRAPH* and *LIST* in the examples. For a Tuple_type, the base class is a fictitious class *TUPLE*, providing the features applicable to all tuples.

For types not immediately obtained from a class we obtain the base class by going through base type: for example *T* is an Anchored type of the form **like** *anchor*, and *anchor* is of type *LIST* [*WINDOW*], then the base class of that type, *LIST*, is also the base class of *T*.

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

Base rule

The **base type** of any type is a Class_or_tuple_type, with no Attachment_mark.

The **base class** of any type other than a Class_or_tuple_type is (recursively) the base class of its base type.

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

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.

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 204); and a* *Precursor* *construct (syntax page 296).*

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*.

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

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

→ Conformance: chapter 14; creation: chapter 20.

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

Class Type rule *VTCT*

A Class_type is valid if and only if it satisfies the following two conditions:

1 • Its Class_name is the name of a class in the surrounding universe.

2 • If it has an Attachment_mark, that class is not expanded.

The class given by condition 1 will be the type's base class. Regarding condition 2, an expanded type is always attached, so an Attachment_mark would not make sense in that case.

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

Type Semantics rule

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

1 • Whether *T* is expanded or reference.

2 • Whether *T*, if reference, is attached or detachable.

3 • What is *T*'s base type.

4 • If *T* is a Class_or_tuple_type, what are its base class and its type parameters if any.

→ For Formal_generic_name types the expanded/reference status depends on each generic derivation. See "SEMANTICS OF GENERIC TYPES", 12.10, page 355.

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_or_tuple_type). Finally, the **values** of *T* will be its instances if it is an expanded type, otherwise references to such instances.

→ *Chapter 19*.

In application of the Type Semantics rule, every presentation of a new kind of type in this chapter and the next two has a SEMANTICS paragraph that simply defines the base type (item 3 above), the base class in the case of a Class_or_tuple_type (4), and whether it is expanded or reference (1).

To simplify the discussion, we allow ourselves to use "base class" and "base type" directly for expressions:

Base class and base type of an expression

Any expression *e* has a **base type** and a **base class**, defined as the base type and base class of the type of *e*.

11.8 CLASS TYPES WITHOUT GENERICITY

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

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

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

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

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

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

Non-generic class type semantics

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

→ The generic version will be only slightly different: "Generically derived class type semantics", page 355.

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

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

→ See chapter 23 about calling features on entities.

Only one constraint, the Class Type rule, applies to a Class_type that is not generic: the Identifier must be the name of a class of the universe.

11.9 EXPANDED TYPES

Most of the types you define will probably be reference types similar to the last examples (*LIST, WINDOW, PARAGRAPH*…), as they offer the flexibility of creating objects on demand, and the ability to define linked structures. You can also use expanded types.

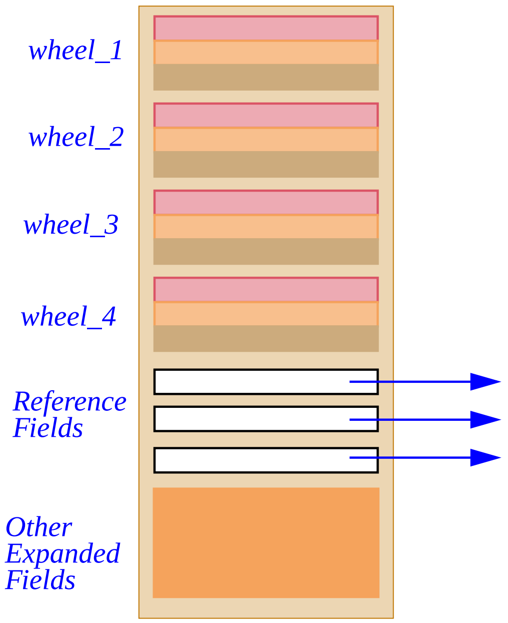
Role of expanded types

An earlier chapter previewed some of the possible reasons for using expanded types:

← "EXPANDED CLIENTS", 7.5, page 192.

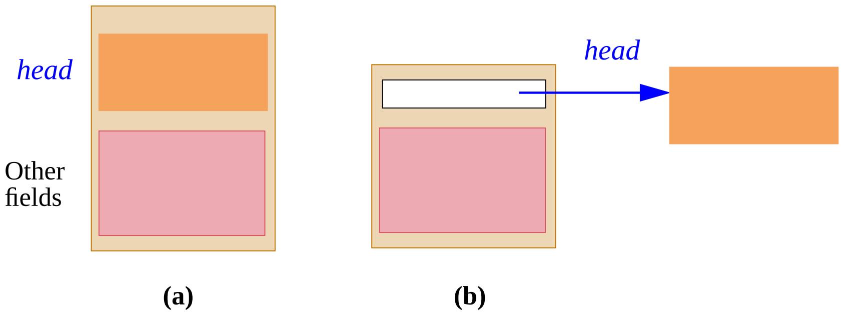
- Realism in modeling external world objects, especially when you want to describe objects that have sub-objects.
- Possible efficiency gain.
- Basic types.
- Interface with other languages.
- Machine-dependent operations.

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



Sub-object vs. reference to another object

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



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 , page 330, about basic types.

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

Defining expanded types

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

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

---- WHOLE DISCUSSION OF “ EXPANDED T” REMOVED -----

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

Expanded type, reference type

A type *T* is **expanded** if and only if it is not a Formal_generic_name and the base class of its deanchored form is an expanded class.

T is a **reference type** if it is neither a Formal_generic_name nor expanded.

This definition characterizes every type as either reference or expanded, except for the case of a Formal_generic_name, which stands for any type to be used as actual generic parameter in a generic derivation: some derivations might use a reference type, others an expanded type.

→ Genericity is discussed in the next chapter.

Tuple types are, as a consequence of the definition, reference types.

Basic types

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

→ Detailed in chapter 30.

- *BOOLEAN*, describing boolean values (true and false).
- *CHARACTER*, describing single characters.
- *INTEGER* and its variants supporting specific sizes: *INTEGER_8*, *INTEGER_16*, *INTEGER_64*. The sizes of values of type *INTEGER* must be settable through a compilation option (the recommended value is 64).
- *REAL*: floating-point numbers and its variants supporting specific sizes: *REAL_32*, *REAL_64*. The sizes of values of type *REAL* must be settable through a compilation option (the recommended value is 64).
- *POINTER*, serving to pass addresses of Eiffel features and expressions to non-Eiffel routines.

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

→ See chapters 36 about ARRAY and STRING and 13 about tuples.

Basic type

The basic types are *BOOLEAN*, *CHARACTER* and its sized variants, *INTEGER* and its sized variants, *REAL* and its sized variants and *POINTER*.

Like most other types, the basic types are defined by classes, found in the Kernel Library. In other words they are not predefined, "magic" types, but fit in the normal class-based type system of Eiffel.

Compilers typically know about them, so that they can generate code that performs arithmetic and relational operations as fast as in lower-level languages where basic types are built-in. This is only for efficient implementation: semantically, the basic types are just like other class types.

Properties of basic types, especially their conformance and semantics, appear in a chapter devoted to them.

→ Chapter 30.

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

→ “EXPANDED TYPE CONFORMANCE”, 14.9, page 386.

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

11.10 ANCHORED TYPES

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

An Anchored type is of the form

like *anchor*

with the predictable definitions:

Anchor, anchored type, anchored entity

The **anchor** of an anchored type **like** *anchor* is the entity *anchor*. A declaration of an entity with such a type is an **anchored declaration**, and the entity itself is an **anchored entity**.

The anchor must be either an entity, or **Current**. If an entity, *anchor* must be the final name of a feature of the enclosing class.

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

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

Anchored examples

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

→ *The encounter was towards the end of* 10.9, page 263.

```
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

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

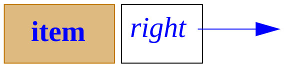
```
set_assignment (a: EMPLOYEE_ASSIGNMENT)
            -- Make a the employee's current assignment.
      require
            exists: a /= Void
      do
            assignment := a
      ensure
            set: assignment = a
      end
```

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:



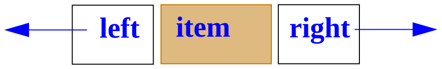
Linkable list cell

This figure and the next appeared previously on page 264.

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

```
right: like Current
```

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



Bi-linkable list cell

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

Anchoring to an expanded or generic

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

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

The expanded case is not very exciting because redefinition 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 307 and "Direct conformance: expanded types", page 388.

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 359; see also "THE TYPE OF AN EXPRESSION", 28.11, page 773.

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 388. On the rules for agents, see

Avoiding anchor cycles

To go from the preceding informal presentation of anchored types to their precise constraint and semantics requires that we address the issue of anchor chains and prohibit cycles.

The syntax permits *x* to be declared of type **like** *anchor* if *anchor* is itself anchored, of type **like** *other_anchor*. Although most developments do not need such anchor chains, they turn out to be occasionally useful for advanced applications. But then of course we must make sure that an anchor chain is meaningful, by excluding cycles such as *a* declared as **like** *b*, *b* as **like** *c*, and *c* as **like** *a*. The following definition helps.

Anchor set; cyclic anchor

The **anchor set** of a type *T* is the set of entities containing, for every anchored type **like** *anchor* involved in *T*:

- *anchor*.
- (Recursively) the anchor set of the type of *anchor*.

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

The anchor set of *LIST* [**like** *a*, *HASH_TABLE* [**like** *b*, *STRING*]] is, according to this definition, the set {*a*, *b*}.

Because of genericity, the cycles that make an anchor "cyclic" might occur not directly through the anchors but through the types they involve, as with *a* of type *LIST* [**like** *b*] where *b* is of type **like** *a*. Here we say that a type "involves" all the types appearing in its definition, as captured by the following definition.

Types and classes involved in a type

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

- *T* itself.
- If *T* is of the form *a T'* where *a* is an Attachment_mark: (recursively) the types involved in *T'*.
- If *T* is a generically derived Class_type or a Tuple_type: all the types (recursively) involved in any of its actual parameters.

The *classes* involved in *T* are the base classes of the types involved in *T*.

A [*B*, *C*, *LIST* [*ARRAY* [*D*]]] involves itself as well as *B*, *C*, *D*, *ARRAY* [*D*] and *LIST* [*ARRAY* [*D*]. The notion of *cyclic anchor* captures this notion in full generality; the basic rule, stated next, will be that if *a* is a cyclic anchor you may not use it as anchor: the type **like** *a* will be invalid.

--- Auxiliary notion:

Constant type

A type *T* is **constant** if every type involved in *T* is a Class_or_tuple_type.

The restriction to Class_or_tuple_type excludes formal generic parameters and anchored types. Constant types are the only ones permitted for constant attributes denoting manifest types.

Validity and semantics of anchored types

The notions just introduced enable us to define the validity of anchored types. Every type has an *deanchored* version, an "unfolded form" which expands the **like**:

← *"TWO-TIER DEFINITION AND UNFOLDED FORMS", 2.11, page 99*.

Deanchored form of a type

The **deanchored form** of a type *T* in a class *C* is the type (Class_or_tuple_type or Formal_generic) defined as follows:

1 • If *T* is **like Current**: the current type of *C*.

2 • If *T* is **like** *anchor* where the type *AT* of *anchor* is not anchored: *AT*.

3 • If *T* is **like** *anchor* where the type *AT* of *anchor* is anchored but *anchor* is not a cyclic anchor: (recursively) the deanchored form of *AT* in *C*.

4 • If *T* is *a AT*, where *a* is an Attachment_mark: *a DT*, where *DT* is (recursively) the deanchored form of *AT* deprived of its Attachment_mark if any.

5 • If none of the previous cases applies: *T*.

Although useful mostly for anchored types, the notion of "deanchored form" is, thanks to the phrasing of the definition, applicable to *any* type. Informally, the deanchored form yields, for an anchored type, what the type "really means", in terms of its anchor's type. It reflects the role of anchoring as what programmers might call a macro mechanism, a notational convenience to define types in terms of others.

Case 4 enables us to treat **? like** *anchor* as a detachable type whether the type of *anchor* is attached or detachable.

Anchored Type rule *VTAT*

It is valid to use an anchored type *AT* of the form **like** *anchor* in a class *C* if and only if it satisfies the following conditions:

1 • *anchor* is either **Current** or the final name of a query of *C*.

2 • *anchor* is not a cyclic anchor.

3 • The deanchored form *UT* of *AT* is valid in *C*.

The base class and base type of *AT* are those of *UT*.

An anchored type has no properties of its own; it stands as an abbreviation for its unfolded form. You will not, for example, find special conformance rules for anchored type, but should simply apply the usual conformance rules to its deanchored form.

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

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

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

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

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

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

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

→ *"CURRENT TYPE, FEATURES OF A TYPE", 12.11, page 357*.

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

→ *"SEMANTICS OF GENERIC TYPES", 12.10, page 355*.

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

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

11.11 GUARANTEEING ATTACHMENT

-----ADD EXPLANATIONS

Attached, detachable

A type is **detachable** if its deanchored form is a Class_type declared with the **?** Attachment_mark.

A type is **attached** if it is not detachable.

By taking the "deanchored form", we can apply the concepts of "attached" and "detachable" to an anchored type **like** *a*, by just looking at the type of *a* and finding out whether it is attached or not.

As a consequence of this definition, an expanded type is attached.

As the following semantic definition indicates, the idea of declaring a type as attached is to guarantee that its values will never be void.

Attached type semantics

Every run-time value of an attached type is non-void (attached to an object).

In contrast, values of a detachable type may be void.

These definitions rely on the run-time notion of a *value* being attached (to an object) or void. So there is a distinction between the *static* property that an entity is attached (meaning that language rules guarantee that its run-time values will never be void) or detachable, and the *dynamic* property that, at some point during execution, its value will be attached or not. If there's any risk of confusion we may say "statically attached" for the entity, and "dynamically attached" for the run-time property of its value.

The validity and semantic rules, in particular on attachment operations, ensure that attached types indeed deserve this qualification, by initializing all the corresponding entities to attached values, and protecting them in the rest of their lives from attachment to void.

From the above semantics, the **!** mark appears useless since an absent Attachment_mark has the same effect. The mark exists to ensure a smooth transition: since earlier versions of Eiffel did not guarantee void-safety, types were detachable by default. To facilitate adaptation to current Eiffel and avoid breaking existing code, compilers may offer a compatibility option (departing from the Standard, of course) that treats the absence of an Attachment_mark as equivalent to **?**. You can then use **!** to mark the types that you have moved to the attached world and adapt your software at your own pace, class by class if you wish, to the new, void-safe convention.

11.12 STAND-ALONE TYPES

Stand-alone type

A Type is **stand-alone** if and only if it involves neither any Anchored type nor any Formal_generic_name.

In general, the semantics of a type may be relative to the text of class in which the type appears: if the type involves generic parameters or anchors, we can only understand it with respect to some class context. A stand-alone type always makes sense — and always makes the same sense — regardless of the context.

We restrict ourselves to stand-alone types when we want a solidly defined type that we can use anywhere. This is the case in the validity rules enabling creation of a root object for a system, and the definition of a once function.