6

The inheritance relation

6.1 OVERVIEW

Inheritance is one of the most powerful facilities available to software developers. It addresses two key issues of software development, corresponding to the two roles of classes:

- As a **module extension** mechanism, inheritance makes it possible to define new classes from existing ones by adding or adapting features.
- As a **type refinement** mechanism, inheritance supports the definition of new types as specializations of existing ones, and plays a key role in defining the type system.

This chapter introduces the fundamental properties of inheritance, concentrating on the first view — the module aspect. It describes in particular the *renaming* mechanism, which brings considerable flexibility by letting you decide anew in each class on the names of the features it inherits. Later chapters discuss the type view of inheritance, which leads to Eiffel's type system, and explore the feature adaptation mechanisms that go with it: redefinition, effecting, undefinition, and the sharing and replication mechanisms of repeated inheritance.

→ Chapters 11 to 13 on typing and 14 on conformance.

6.2 AN INHERITANCE PART

To define a class as inheriting from one or more others, include one or more Inheritance parts, each introduced by the keyword **inherit**.

Below is a slightly simplified form (omitting in particular the Notes clause) of the beginning of class *FIXED_TREE* from the EiffelBase Library. It shows a typical Inheritance part, indicating that *FIXED_TREE* obtains some of its features from three other classes:

- *TREE*, describing the general notion of tree, regardless of representation.
- *CELL*, describing elements used to store an individual piece of information (such as a tree node).
- *FIXED_LIST*, providing some of the implementation.

```
class
    FIXED_TREE [T]
inherit
    TREE [T]
        redefine
            attach_to_higher
        end
    CELL [T]
inherit {NONE}
    FIXED_LIST [T]
        rename
            off as child_off,
            after as child_after,
            before as child_before
        redefine
            duplicate, first_child
        end
feature
    … (Rest of class omitted) …
```

The classes listed in the two Inheritance parts, *TREE, CELL* and *FIXED_LIST*, are said to be the "parent classes", or just "parents", of *FIXED_TREE*. This is defined as a case of *multiple* inheritance.As the fixed-tree example shows, there is often a need to adapt the features of parents to a new class. This is achieved through the Feature_adaptation part of a Parent part, highlighted above: a Redefine clause for the *TREE* parent and a Rename clause for *FIXED_LIST*.

→ *The notion of parent is defined precisely in the next section.*

The first inheritance clause, introduced by just **inherit**, guarantees conformance of the class to the two parents listed. The other one, introduced by **inherit** {*NONE*}, provides non-conforming inheritance, giving the new class access to the features of the parent — *FIXED_LIST* — without introducing a "subtyping" (conformance) relation.

A Feature_adaptation part may contain Redefine and Rename subclauses, as here, as well as others — Undefine, New_exports, Select — listed in the syntax below.

6.3 FORM OF THE INHERITANCE PART

Here is the relevant syntax:

Inheritance parts

Inheritance ≜ Inherit_clause⁺

Inherit_clause ≜ **inherit** [Non_conformance] Parent_list

Non_conformance ≜ "**{**" [*NONE*] "**}**"

Parent_list ≜ {Parent ";" …}⁺

Parent ≜ Class_type [Feature_adaptation]

Feature_adaptation ≜ [Undefine]
[Redefine]
[Rename]
[New_exports]
[Select]
end

As with all other uses of semicolons, the semicolon separating successive Parent parts is optional. The style guidelines suggest omitting it between clauses that appear (as they should) on successive lines.

→ *"OPTIONAL SEMICOLONS", 34.10, page 909.*

A Parent_list names one or more Parent parts. Each is relative to a Class_type, that is to say a class name *B* possibly followed by actual generic parameters (as in *B* [*T*, *U*]). *B* must be the name of a class in the universe to which the current class belongs. This property yields a definition:

→ *Class types are studied in chapter 11. The requirement that B be a class of the universe follows from the Class Type rule, page 325.*

Parent part for a type, for a class

If a Parent part *p* of an Inheritance part lists a Class_type *T*, *p* is said to be a Parent part **for** *T*, and also for the base class of *T*.

So in **inherit** *TREE* [*T*] there is a Parent part for the type *TREE* [*T*] and for its base class *TREE*. For convenience this definition, like those for "parent" and "heir" below, applies to both types and classes.

The earlier declaration of *FIXED_TREE* contains Parent parts for classes *TREE*, *CELL* and *FIXED_LIST*.

Specifying {*NONE*} (a Non_conformance marker) in an Inherit_clause yields a restricted form of inheritance, where the new class has access to the features and invariant of each parent listed, but the corresponding types do not conform to the parent types. This is known as *non-conforming inheritance* and detailed later in this chapter.

→ *"NON-CONFORMING INHERITANCE", 6.8, page 178.*

After the Class_type in a Parent part you may also specify an optional Feature_adaptation clause listing the modifications that the new class wants to perform on the features it inherits from that parent. These modifications may affect various properties of the features, each handled by a subclause of Feature_adaptation:

- Their effectiveness status, deferred or effective (Undefine).
- Their signature and implementation (Redefine).
- Their names (Rename).
- Their export status (New_exports).
- Their resolution of dynamic binding conflicts under repeated inheritance (Select).

Rename is studied later in this chapter, the others in subsequent chapters, in particular one devoted entirely to feature adaptation.

→ *See* 6.9, page 180 *on* Rename; *chapter* 10 *on* Feature_adaptation, *especially* Redefine *and* Undefine (*the latter in* 10.19, page 283); , page 200 *on* New_ exports; 16.12, page 455 *on* Select.

The syntax also tells us exactly when inheritance is "multiple":

Multiple, single inheritance

A class has **multiple inheritance** if it has an Unfolded Inheritance part with two or more Parent parts. It has **single inheritance** otherwise.

What counts for this definition is the number not of parent classes but of Parent parts. If two clauses refer to the same parent class, this is still a case of multiple inheritance, known as **repeated inheritance** and studied later on its own. If there is no Parent part, the class (as will be seen below) has a de facto parent anyway, the Kernel Library class *ANY*.

→ *See chapter* 16

The definition refers to the "Unfolded" inheritance part which is usually just the Inheritance part but may take into account implicit inheritance from *ANY*, as detailed in the corresponding definition below.

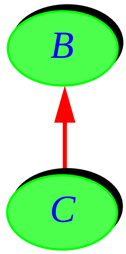
→ *Page* 173.

Multiple inheritance is a frequent occurrence in Eiffel development; most of the effective classes in the widely used EiffelBase library of data structures and algorithms, for example, have two or more parents. The widespread view that multiple inheritance is "bad" or "dangerous" is not justified; most of the time, it results from experience with imperfect multiple inheritance mechanisms, or improper uses of inheritance. Well-applied multiple and repeated inheritance is a powerful way to combine abstractions, and a key technique of object-oriented software development.

6.4 GRAPHICAL CONVENTION

In pictorial representations of system structures, where classes appear as labeled ellipses, the inheritance relation is represented by single arrows (red if color is available) pointing from heirs' ellipses to parents' ellipses.

Parent and heir



6.5 RELATIONS INDUCED BY INHERITANCE

Inheritance introduces the "parent" and "heir" relations between classes:

Inherit, heir, parent

A class *C* **inherits** from a type or class *B* if and only if *C*'s Unfolded Inheritance Part contains a Parent part for *B*.

B is then a **parent** of *C* ("parent type" or "parent class" if there is any ambiguity), and *C* an **heir** (or "heir class") of *B*. Any type of base class *C* is also an heir of *B* ("heir type" in case of ambiguity).

Listing *{NONE}* indicates that the relation does not imply conformance of the associated types:

Conforming, non-conforming parent

A parent *B* in an Inheritance part is **non-conforming** if and only if every Parent part for *B* in the clause appears in an Inherit_clause with a Non_conformance marker. It is **conforming** otherwise.

The reflexive transitive closures of the basic relations are also of interest:

"Reflexive transitive closure" means the relation iterated any number of times (zero or more).

Ancestor types of a type, of a class

The **ancestor types** of a *type* *CT* of base class *C* include:

1 • *CT* itself.

2 • (Recursively) The result of applying *CT*'s generic substitution to the ancestor types of every parent type for *C*.

The ancestor types of a *class* are the ancestor types of its current type.

The basic notion is for ancestor types of a type. Case 1 indicates that a type is its own ancestor. Case 2, the recursive case, applies the notion of *generic substitution* introduced in the discussion of genericity. The idea that if we consider the type *C [INTEGER]*, with the class declaration **class** *C [G]* **inherit** *D [G] …*, the type to include in the ancestors of *C [INTEGER]* as a result of this Inheritance part is not *D [G]*, which makes no sense outside of the text of *C*, but *D [INTEGER]*, the result of applying to *D [G]* the substitution *G* → *INTEGER*; this is the substitution that yields the type *C [INTEGER]* from the class *C [G]* and is known as the generic substitution of that type.

From ancestor types we obtain ancestor classes, called just ancestors:

Ancestor, descendant

Class *A* is an **ancestor** of class *B* if and only if *A* is the base class of an ancestor type of *B*.

Class *B* is a **descendant** of class *A* if and only if *A* is an ancestor of *B*.

Any class, then, is both one of its own descendants and one of its own ancestors. *Proper* descendants and ancestors exclude these cases.

Proper ancestor, proper descendant

The **proper ancestors** of a class *C* are its ancestors other than *C* itself. The **proper descendants** of a class *B* are its descendants other than *B* itself.

6.6 *ANY*

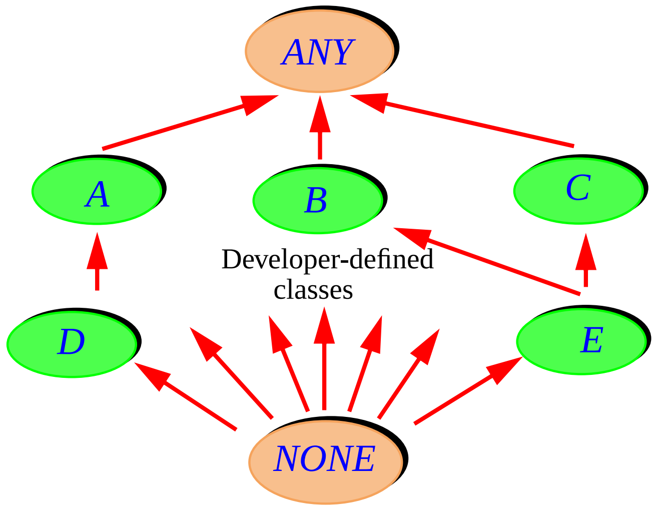
No class that you write is an orphan.

An important property of the inheritance structure is that every class inherits, directly or indirectly, from a class called *ANY*, of which a version is provided in the Kernel Library. The semantics of the language depends on the presence of such a class, whether the library version or one that a programmer has provided as a replacement.

The convention ensuring this property — illustrated by the figure on the facing page — is that any class that doesn't have an explicit Inheritance part is considered to have *ANY* as its parent.

The figure also shows, at the bottom, a fictitious class *NONE*, studied next. But there's nothing fictitious about *ANY:* → *"NONE", 6.7, page 175*.

The inheritance structure



Class *ANY* rule — *VHCA*

Every system must include a non-generic class called *ANY*.

The key property of *ANY* is that it is not only an ancestor of all classes and hence types, but that all types **conform** to it, according to the following principle, which is not a separate validity rule (although for reference it has a code of its own) but a consequence of the definitions and rules below.

Universal Conformance principle — *VHUC*

Every type conforms to *ANY*.

To achieve the Universal Conformance principle, the semantics of the language guarantees that a class that doesn't list any explicit Parent is considered to have *ANY* as its parent. This is captured by the notion of Unfolded Inheritance Part. The definition of "parent" below, and through it the definition of "ancestor", refer to the Unfolded Inheritance Part of a class rather than its actual Inheritance part.

Unfolded Inheritance Part of a class

Any class *C* has an **Unfolded Inheritance Part** defined as follows:

1 • If *C* has an Inheritance part: that part.

2 • Otherwise: an Inheritance part of the form **inherit** *ANY*.

The fictitious clause **inherit** *ANY* is conforming.

If a class had one or more Parent clauses, but all were non-conforming, it would violate the Universal Conformance principle; we won't allow this. The solution is simple: in this (rare) case, just add **inherit** *ANY*, explicitly.

→ *"Parent rule", page 176, condition 4.*

The special status of *ANY* implies two key properties, corresponding to the type and module views of inheritance:

1 • *ANY* is the most general of directly useful types: any type that you may define will conform to *ANY*.

2 • The features of *ANY*, describing general-purpose operations, are universal: any class that you may define will have access to them.

As a consequence of property 1, if you want a routine to be applicable to objects of arbitrary developer-defined types, you may give it an argument of type *ANY*. An example is the function, declared in *ANY* itself, that produce a duplicate of an object:

```
cloned (other: ANY): like Current
            -- Void if other is void; otherwise, new object
            -- field-by-field identical to object attached to other
… Rest of routine omitted …
```

→ "CLONING AN OBJECT", 21.4, page 567.

Property 2 provides every developer-defined class with a set of important universal features coming from *ANY*. Some examples are the function *cloned* as sketched above, the procedures *default_rescue* and *default_create* giving default exception and creation behavior and the function *out* producing a string representation of any object.

→ See 26.5, page 686, about *default_rescue*.

If you write a class that has no explicit Parent, and hence automatically inherits *ANY*, you can't do anything — renaming, redefinition, …— to the features from *ANY*. If you do want to adapt them, the solution is simply to make the inheritance explicit:

```
class C inherit
    ANY

            redefine copy, default_rescue, … end
feature
      …
end
```

The special role of *ANY* turns any case of *multiple* inheritance into a case of *repeated* inheritance: on the earlier figure, *E* is an heir to both *B* and *C*, and hence an indirect descendant of *ANY* in two ways. Such situations are addressed through the normal rules of repeated inheritance (discussed below and detailed in a later chapter). Unless you specify otherwise, repeated inheritance from *ANY* will produce the expected effect for a class such as *E*: the class will have just one version of every feature from *ANY*, as if it there were just one inheritance path.

← Page 173.
→ Chapter 16; see especially "SHARING ANDREPLICATION", 16.4, page 428.

6.7 *NONE*

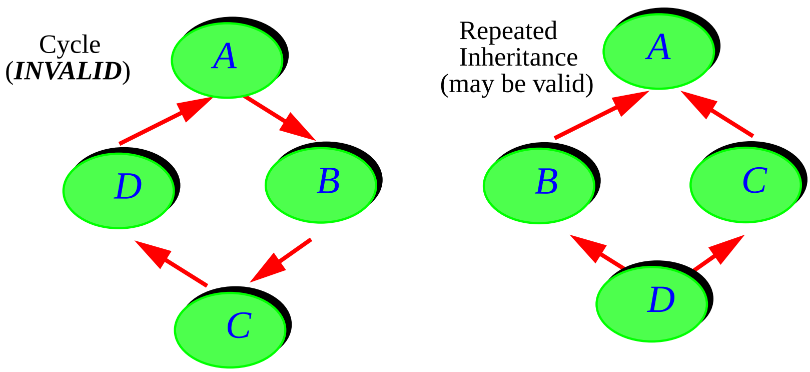
The overall inheritance figure shows, along with *ANY* at the top, another special class at the bottom: *NONE*. This class is considered to inherit from all classes that have no other heirs — assuming appropriate renaming to remove any resulting name clashes. ← *Page 173*.

Unlike *ANY*, *NONE* does not actually exist as a class text (if only because that text would need to be updated every time anyone, anywhere, writes a new class!), but serves as a convenient fiction to make the inheritance structure and the type system complete.

NONE has no useful instance. It serves as the type of *Void*, which denotes a void reference. Since *NONE* is assumed to be a descendant of every class, the Parent rule below implies that no class may be an heir of *NONE*. The class does not export any feature, to help ensure that no feature call has a void target. → *Page 176*.

6.6 PROHIBITING CYCLES

An important constraint governs the inheritance relation: there must be no inheritance cycles.



Invalid cycle vs. valid repeated inheritance

In other words, you may not build a class structure as in the left part of the figure, where *D* inherits from *B*, *B* from *A*, *A* from *C* and *C* from *D*. More generally, it is invalid to have a set of classes C_0, C_1, …, C_n ($n \geq 1$), where C_0 and C_n are the same class and every C_i is an heir of C_{i+1}.

The reason for this restriction is easy to understand: making *C* an heir to *B* means defining the set of features of *C* as an extension of *B*'s feature set; the relationship cannot be mutual.

Prohibiting cycles does not mean prohibiting a class *D* from being a descendant of another class *A* in more than one way, as illustrated by the structure appearing in the right part of the above figure. This is a caseof **repeated inheritance**, valid if it meets the relevant validity constraints. → *Chapter 16*.

These observations lead to the validity constraint on Inheritance parts:

Parent rule *VHPR*

The Unfolded Inheritance Part of a class *D* is valid if and only if it satisfies the following conditions:

1 • In every Parent part for a class *B*, *B* is not a descendant of *D*.

2 • No conforming parent is a frozen class.

3 • If two or more Parent parts are for classes which have a common ancestor *A*, *D* meets the conditions of the Repeated Inheritance Consistency constraint for *A*.

4 • If one or more Parent parts are present, at least one of them is conforming.

5 • No two ancestor types of *D* are different generic derivations of the same class.

6 • Every Parent is generic-creation-ready.

Condition 1 ensures that there are no cycles in the inheritance relation. → *Page 458*.

The purpose of declaring a class as **frozen** (case 2) is to prohibit subtyping. We still permit the *non-conforming* form of inheritance, which permits reuse but not subtyping.

Condition 3 corresponds to the case of repeated inheritance; the Repeated Inheritance Consistency constraint will guarantee that there is no ambiguity on features that *D* inherits repeatedly from *A*. → *Page 458*.

Condition 4 governs *non-conforming* inheritance; it ensures the Universal Conformance principle. If there are no Inheritance part we accept this — since the rule applies to the Unfolded Inheritance Part of the class — as shorthand for one of the form **inherit** *ANY*; but with an Inheritance part that would only have branches marked {*NONE*}, this rule would not apply, and so the current type would not conform to *ANY*. If at least one branch is conforming, then the corresponding parent type will (through recursive application of the same property) conform to *ANY*, and so will the current type. → *Studied below: "NON-CONFORMING INHERITANCE", 6.8, page 178*.

Condition 5 avoids ambiguity when one of the ancestor classes is a generic class *A* [*G*] with, for example, a feature *f* (*x*: *G*); if we allowed a class *C* to inherit from both *A* [*T*] and *A* [*U*] for different types *T* and *U*, there could be no proper signature for *f* in *C*. → *Studied below: "NON-CONFORMING INHERITANCE", 6.8, page 178*.

Condition 6 also concerns the case of a generically derived Parent *A* [*T*]; requiring it to be "generic-creation-ready" guarantees that creation operations on *D* or its descendants will function properly if they need to create objects of type *T* → *"Generic-creation-ready type", page 352*.

When applying the Parent rule, do not be misled by the "if" part of the "if and only if": to guarantee that an Inheritance part is valid, you will also have to check conditions which do not appear explicitly in the rule. In particular:

- Every parent *P* must be a valid type; this means among other requirements that if *P* is generically derived, appearing as *B* [*X*, …], then *B* must be the name of a generic class in the surrounding universe and the actual parameters *X*, … must be valid types matching the formal parameters of *B*.

→ *The Class Type rule, "VTCT", page 325, requires P to be the name of a class in the universe. On generic parameters, see the rule "VTGD", page 351.*

- Every Feature_adaptation clause (with its Rename, Redefine and other subclauses) must be valid.

The Parent rule does not need, however, to express such requirements explicitly: The General Validity rule implicitly adds to the constraint on any construct the condition that all the sub-components are valid too. Be sure to remember this convention — without which the validity rules would become hopelessly complicated — whenever you see an "if and only if" validity constraint throughout this book. If you have the impression that the constraint does not cover every necessary condition, this is probably just because it omits the validity requirements on sub-components, as permitted by the General Validity rule.

← *General Validity rule: page 98.*

6.7 ADAPTING INHERITED FEATURES

The major purpose of inheriting from one or more classes is to obtain their features (together with the associated assertions, and the classes' invariants) as an addition to one's own. The features obtained by a class from its parents are called its *inherited* features. As already noted, this yields one of the two categories of features of a class; the others are *immediate* features, introduced in a class itself.

← *"Features of a class" and "inherited features" were first discussed in 5.4, page 133*

The very notion of inherited feature indicates how inheritance provides an accumulation process enabling classes to use features defined in one or more previously existing classes – its proper ancestors.

Although a class inherits all its proper ancestors' features, it retains the flexibility to adapt them to its own context in various ways:

- A feature introduced in a certain class under a certain name may be known under different names in descendant classes. This is achieved through **renaming**.
- A feature defined with a certain signature, specification and implementation may get a new declaration changing any of these properties. This is achieved through **redefinition**.
- A feature introduced with a certain signature may get a new one. This is also achieved through redefinition, and through the associated mechanism of **anchored declaration**.

- A feature introduced in a proper ancestor with a specification but no implementation, known as a *deferred* feature, may get an implementation. This is the process of **effecting**.
- If a class *C* inherits two or more deferred features with compatible signatures and specifications, it may merge them into a single feature. This is a **join**.

→ *"THE JOIN MECHANISM", 10.21, page 286*.

- When a class *C* inherits the same feature from two or more of its parents, which themselves inherit it from a common ancestor, simple techniques are available to ensure that the result in *C* is only one feature (sharing) or several (duplication). The applicable rules are those of **repeated inheritance**.
- Under repeated inheritance, polymorphism and dynamic binding could cause conflicts, which you must remove through the Select mechanism.

The first of these techniques, renaming, is purely syntactical, affecting feature names rather than the features themselves. It is studied later in this chapter. The others determine the semantic adaptation of features to the context of new descendants; later chapters explore them in detail.

→ *Chapter 10 on feature adaptation and 16 on repeated inheritance*

6.8 NON-CONFORMING INHERITANCE

(The mechanism described here is for advanced users. On first reading you may skip the present section.)

→ *Skip to "RENAMING", 6.9, page 180*

One of the principal applications of inheritance — in its "type" rather than "module" persona — is to govern conformance. The basic idea is simple: in the most common cases, an assignment of the form *a1* := *b1* with *a1* of type *A* and *b1* of type *B* is valid if *B* is a descendant of *A*. You can similarly call *f* (*b1*) if *f* has a formal argument of type *A*. The details appear in the conformance chapter.

→ *Chapter 14*.

Sometimes, you may want inheritance *without* conformance: the module-only side of inheritance, disallowing such assignments and arguments passing. To force this it suffices to prefix the mention of *A* in the corresponding Parent part by keyword {*NONE*}, as in

```
class B inherit
        {NONE} A
                ... Feature_adaptation clause if needed ...
... Rest of class omitted ...
```

Adding {*NONE*} in this fashion does not affect the basic properties of the inheritance relation; it simply means that type *B* will not conform to *A* through this inheritance link.

In a case of repeated inheritance, B might still conform to A through another inheritance link.

The syntax is reminiscent of the possibility of declaring features in a clause **feature** {*NONE*}, rather than just **feature**, to restrict its export status.

This facility is useful only in specific cases of restricting an inheritance link to "implementation inheritance" or "facility inheritance": you want the reusability benefits of inheritance, but not the subtyping part.

Some simple-minded presentations of object technology will tell you that this is "wrong" and that inheritance should always involve subtyping. Although they can legitimately point to incorrect uses of inheritance, it is improper to disallow implementation inheritance altogether, as it has many perfectly valid uses. The chapter on the methodology of inheritance in *Object-Oriented Software Construction* discusses these issues in detail and presents a taxonomy of the uses of inheritance.

In this book we will see two major applications of non-conforming inheritance, both of which use it to remove potential ambiguities: repeated inheritance and convertibility.

- The repeated inheritance chapter will show that it is sometimes possible for a class to obtain two different versions of a feature inherited from a common ancestor through more than one path. This creates a potential ambiguity because of polymorphism and dynamic binding, since a call of the form *a*.*f*, where *a* is of the repeated ancestor type, could in principle trigger either of the two variants if *a* is attached at run time to an instance of the common descendant type. When such a conflict arises, you will resolve it through a Select clause. The problem only arises, however, if both paths are conforming; by using non-conforming inheritance whenever you don't need subtyping you reduce the need for Select and simplify your class texts.
- The study of convertibility will show how to make a type convertible to another by including conversion procedures, as in → *Chapter 15*.

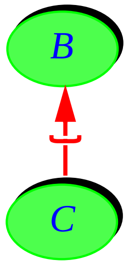
```
class A create
    from_B convert {B}
... Rest of class omitted ...
```

which makes assignments such as *a1* := *b1* (and corresponding argument passing) valid; they will cause a conversion using the listed creation procedure *from_B*. To avoid any ambiguity, the Conversion Procedure rule prohibits such a scheme when *B* conforms to *A*, as this would also make the assignment valid but with a different semantics (reference reattachment with no conversion). The general principle is that a type may conform or convert to another, but not both. In some cases you might still like *B* to inherit from *A* for its features only. It suffices in this case to make *B* list {*NONE*} *A*, rather than just *A*, as its Parent. → *"Conversion Procedure rule", page 403*; *"Conversion principle", page 400*.

This discussion also explains why we needed condition 4 of the Inheritance rule, requiring that if there are Parent parts they can't all be non-conforming: we need at least one conforming branch to ensure that all types conform to *ANY* — the Universal Conformance rule. ← *Page 176* (both rules).

The graphical representation of inheritance links has a slightly different form (similar to the convention for the "expanded client" relation) to signal non-conforming inheritance:

→ *Page 194, in the next chapter.*



Parent and non-conforming heir

6.9 RENAMING

As part of its Feature_adaptation, any Parent part may include a Rename subclause, which serves to adapt names of inherited features to the local context of the new class.

Here is a Rename subclause from the previous example:

```
rename
    off as child_off,
    after as child_after,
    before as child_before
```

Renaming is especially useful in two cases:

- With renaming, you may correct any **name clash** occurring because of multiple inheritance. A name clash occurs when two or more parents of a class have a feature of the same name, and would usually make the class invalid if not removed by renaming.
- Renaming also enables a class to offer its inherited features to its clients and descendants under a terminology appropriate to its own context, rather than to the context of the parents from which it inherited them. In other words, it helps make sure that, beyond offering the right *features*, you also offer them under the right *feature names*.

→ *"NAME CLASHES", 10.23, page 290, discusses the exact cases in which name clashes are prohibited.*

The general syntax of a Rename clause is:

Rename clauses

Rename ≜ **rename** Rename_list

Rename_list ≜ {Rename_pair "," …}*

Rename_pair ≜ Feature_name **as** Extended_feature_name

The first component of a Rename_pair is just a Feature_name, the identifier for the feature; the second part is a full Extended_feature_name, which may include an **alias** clause. Indeed:

- To identify the feature you are renaming, its Feature_name suffices.
- At the same time you are renaming the feature, you may give it a new operator or bracket alias, or remove the alias if it had one.

Forms of feature adaptation other than renaming, in particular effecting and redefinition, do not affect the Alias, if any, associated with a Feature_name.

So if *B* has the features

```
plus alias "+"
multiplied alias "∗"
divided alias "/"
item alias "[ ]"
f
g
```

you may define a new class

Warning: this is an extreme case, illustrating the possibilities but not intended as a model of style!

```
class C inherit
    B
        rename
            plus as sum alias "+",
            multiplied as times,
            divided as divided alias "//",
            item as item,
            f as f alias "[ ]",
            g as h alias "||"
        end
    … Rest of class omitted …
```

Then for the features offered by *C* to its direct clients:

- *plus* changes its identifier to *sum* and keeps its alias. Without the **alias** part it would no longer have an operator alias in *C*.
- *multiplied* is renamed to *times* and loses its alias.
- *divided* keeps its identifier but changes its alias; you can't change just the alias without giving a full new Extended_feature_name, which in this case reuses the previous Feature_name (the identifier *divided*).
- *item* keeps its identifier and loses its bracket alias; again you have to repeat the identifier.
- *f* takes over the bracket alias vacated by *item*. Since every class may have at most one feature with the bracket alias, this would not be possible without the change to *item*.
- *g* gets a new identifier and a new alias, the free operator ||.

→ *"Free operator", page 883*

The aliases all assume that the corresponding features have the right signatures; for example "+" as a Binary requires a one-argument query.

← “Alias Validity rule”, page 162

The Rename clause is subject to a constraint.:

> **Rename Clause rule** *VHRC*
>
> A Rename_pair of the form *old_name* **as** *new_name*, appearing in the Rename subclause of the Parent part for *B* in a class *C*, is valid if and only if it satisfies the following conditions:
>
> 1 • *old_name* is the final name of a feature *f* of *B*.
>
> 2 • *old_name* does not appear as the first element of any other Rename_pair in the same Rename subclause.
>
> 3 • *new_name* satisfies the Feature Name rule for *C*.
>
> 4 • The Alias of *new_name*, if present, is alias-valid for the version of *f* in *C*.

→ The Feature Name rule, page 466, expresses that no other feature of C has new_name as its final name.

→ “Feature Name rule”, page 466.

In condition 4, the “alias-valid” condition captures the signature properties allowing a query to have an operator or bracket aliases. It was enforced when we wanted to give a feature an alias in the first place and, naturally, we encounter it again when we give it an alias through renaming.

← “Alias Validity rule”, page 162.

← Clauses 5 and 7 of “Feature Declaration rule”, page 160.

Renaming is a purely syntactical mechanism:

> **Renaming principle**
>
> Renaming does not affect the semantics of an inherited feature.

The “positive” semantics of renaming (as opposed to the negative observation captured by this principle) follows from the definition of *final name* and *extended final name* of a feature below.

→ Page 183.

This principle indeed adds nothing by itself to the semantics of the language; it is there to remove any uncertainty. Experience has shown that renaming sometimes confuses newcomers to object technology — surprisingly, since the idea is particularly simple: to distinguish between a feature and its name.

See “Repentant Java programmer can’t understand the difference between a feature and a feature name”, in Proc. BEIROOT ‘05 (Bizarre Experiences In Remedial Object-Oriented Training), Beirut, Aug. 2005, pages 22345-27226.

6.10 FEATURES AND THEIR NAMES

A class defines a set of features, each with a certain feature names. The two concepts are clearly distinct.

A feature is a certain component (attribute or routine), characterized by a signature, an associated algorithm (for a routine), a value (for a constant attribute), and possibly other properties. Such a feature is “*a feature of*” one or more classes: the class which introduces it, and (subject to feature adaptation mechanisms) all the descendants of that class.

Every feature of a class has a name in that class. This association between a feature and a feature name only eixts relative to the class. The same *feature* may have different *feature names* in different classes.

This is precisely what renaming achieves. The presence, in a Parent clause for *B* in *C*, of a Rename subclause of the form

```
rename …, f as g, …
```

implies that the inherited feature known as *f* in *B* is known as *g* in *C*.

The precise definitions are the following:

The notion of "class of origin" was first introduced on page 133. The full definition appears on page 305.

Final name, extended final name, final name set

Every feature *f* of a class *C* has an **extended final name** in *C*, an Extended_feature_name, and a **final name**, a Feature_name, defined as follows:

1 • The final name is the identifier of the extended final name.

2 • If *f* is immediate in *C*, its extended final name is the Extended_feature_name under which *C* declares it.

3 • If *f* is inherited, *f* is obtained from a feature of a parent *B* of *C*. Let *extended_parent_name* be (recursively) the extended final name of that feature in *B*, and *parent_name* its final name of *f* in *B*. Then the extended final name of *f* in *C* is:

- If the Parent part for *B* in *C* contains a Rename_pair of the form **rename** *parent_name* **as** *new_name*: *new_name*.
- Otherwise: *extended_parent_name*.

The final names of all the features of a class constitute the **final name set** of a class.

→ How the final name set is actually determined depends on renaming, redefinition and joining, as discussed in chapters 10 and 16. See further comments about the final name set on page 465.

Since an inherited feature may be obtained from two or more parent features, case 3 only makes sense if they are all inherited under the same name. This will follow from the final definition of "inherited feature" in the discussion of repeated inheritance.

→ "Inherited features", page 462.

The extended final name is an Extended_feature_name, possibly including an Alias part; the final name is its identifier only, a Feature_name, without the alias. The recursive definition defines the two together.

Also convenient is the notion of "inherited name" of an inherited feature:

Inherited name

The **inherited name** of a feature obtained from a feature *f* of a parent *B* is the final name of *f* in *B*.

In the rest of the language description, references to the "name" of a feature, if not further qualified, always denote the final name.

Renaming — to press the point! — does not change any of the inherited features, but simply changes the names under which those features will be known by clients and descendants. Consider a feature *f*, which has the final name *old_name* in a class *B*. By writing an heir *C* as

```
class C inherit
    …,
    B
        rename …, old_name as new_name , … end
```

you decide to make the inherited feature available to *C*, *C*'s descendants and (if it is exported) *C*'s clients under the name *new_name*.

As a consequence, you have also freed the inherited name of *f*, here *old_name*, so that another feature of *C* may now use this name. That other feature could come from various places:

1 • It could be a new feature introduced by *C* itself, for which you wish to use the name *old_name*.

2 • It could be a feature inherited from a parent of *C* other than *B*, and having the name *old_name* in that parent. Here, without renaming, you would have introduced a — usually invalid — name clash in *C*.

3 • It could even be a feature inherited from *B* or another parent under some other name, and renamed *old_name* in *C*. This case is somewhat contorted, but it does occasionally arise.

Whatever the case, remember that if you do decide to reuse *old_name* for another feature of *C*, you do not introduce any connection between that feature and the original feature *f*, obtained from *B* under the inherited name *old_name*. The two are unrelated; for example one could be a procedure and the other an attribute.

The following example illustrates these properties. Assume a class *COLORS* with features of names *red*, *orange*, *black*, *white*, and *FRUITS* with a feature of name *orange_fruit*. You can write a class of the form

```
class FRUITS_AND_COLORS inherit
    COLORS
        rename
            orange as orange_color, red as red_color,
            black as white, white as black
        end
    FRUITS
        rename
            orange_fruit as orange
        end
feature
    red: INTEGER
end
```

There is no assumption that these classes and features have any use as abstractions reflecting their names; they just illustrate some language properties.

The feature *orange* of class *COLORS* is known in *FRUITS_AND_COLORS* as *orange_color*; this makes the name *orange* available for the feature inherited from *FRUITS* under the name *orange_fruit*. The feature *red* of *COLORS* is known in *FRUITS_AND_COLORS* as *red_color*, making the name *red* free for a new attribute introduced in *FRUITS_AND_COLORS* with no connection to the original *red*. Finally each of *COLORS*'s features *black* and *white* is known in *FRUITS_AND_COLORS* under the other's name.

As this example illustrates, you should understand the renamings induced by a Rename subclause as all simultaneous; this allows such constructions as **rename** *black* **as** *white*, *white* **as** *black* to make sense. In other words, even if the Rename subclause includes a Rename_pair *old_name* **as** *new_name*, other occurrences of *old_name* or *new_name* as the first element of a Rename_pair in the same subclause must still be interpreted as in the parent.

This last case, which swaps the names of two inherited features, is rather extreme. It illustrates, however, the importance of renaming to the building of professional-quality reusable software components. Writing a class as heir to another means endowing the new class with a certain *functionality*, as provided by the parent's features. But this does not by itself make these features available under a *terminology* consistent with the heir's specific context. Renaming is there to guarantee that, for the heir, its clients and its descendants, the terminology is just as right as the functionality is.

An auxiliary notion resulting from this discussion proves convenient:

Declaration for a feature

A Feature_declaration in a class *C*, listing a Feature_name *fn*, is a **declaration for** a feature *f* if and only if *fn* is the final name of *f* in *C*.

Although it may seem almost tautological, we need this definition so that we can talk about a declaration "for" a feature *f* whether *f* is immediate — in which case *fn* is just the name given in its declaration — or inherited, with possible renaming. This will be useful in particular when we look at a *redeclaration*, which overrides a version inherited from a parent.

6.11 INDEPENDENCE OF INHERITANCE AND EXPANSION

The "expanded" or "reference" status of a class is not inherited.

As you may remember, a Class_header may begin with

→ See "CLASS TEXT STRUCTURE", 4.6, page 117..

```
expanded class C…
```

as opposed to the more common **class** *C* or **deferred class** *C*. If the **expanded** mark is present, the class and types based on it are said to be expanded. Creation of an instance, as in

```
x: C
…
create x.…
```

will yield an objects with *copy semantics* rather than reference semantics. What effect does this have on heirs of *C*?

SEMANTICS

The answer is straightforward: no effect. The only consequence of the expansion status of a class is the semantics of objects of the corresponding types, such as the object attached to *x* above. An expanded class may inherit from a non-expanded one, and conversely. The expansion status is not transmitted, but entirely determined by the class's own Class_header.

This convention makes it easy to provide both a reference and expanded versions of the same class, as in

```
class RC feature
    … Full class declaration: feature declarations, invariant etc. …
end

expanded class EC inherit
    RC
    -- No need to write anything else, except possibly
    -- Notes and Creation clauses
end
```

The two classes have the same features; one is expanded, the other is not. Because of the rules on creation, each will have to list the procedures, if any, that it plans to use as creation procedures.