# Repeated inheritance

## 16.1 OVERVIEW

Inheritance may be **multiple**: a class may have any number of parents. A more restrictive solution would limit the benefits of inheritance, so central to object-oriented software engineering.

Because of multiple inheritance, it is possible for a class to be a descendant of another in more than one way. This case is known as **repeated** inheritance; it raises interesting issues and yields useful techniques, which the following discussion reviews in detail.

The figure on the next page shows examples of repeated inheritance.

The present chapter is the last of three devoted to inheritance. It doesn't introduce any new language construct but explains the validity rules and semanitcs of repeated inheritance. As a consequence, it will complete our understanding of two important inheritance concepts: **inherited feature** and **name clash**.

← *The other two were 6 and 10.*

Our view of inheritance will only be final when we have grasped the semantics of reattachment and feature call, involving the powerful techniques of polymorphism and dynamic binding.

→ *"POLYMORPHISM", 22.11, page 598*; *"DYNAMIC BINDING", 23.12, page 630*.

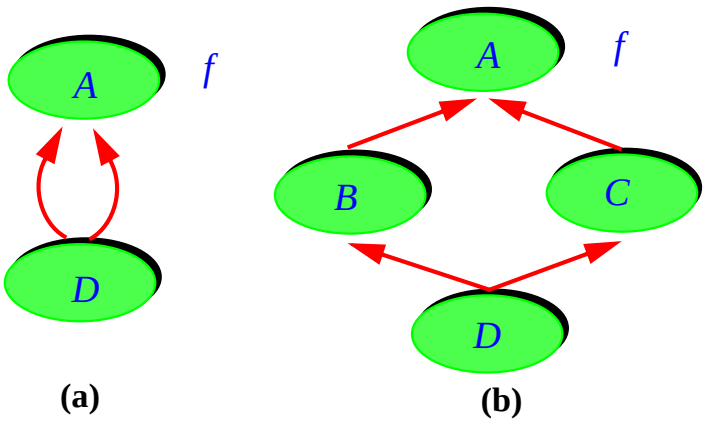
This chapter is organized in four parts:

- We look into the circumstances of repeated inheritance.
- We identify the **two questions** that repeated inheritance implies for an object-oriented language — Are features shared or replicated? If replicated, what does this mean for dynamic binding? — and answer them through simple language rules.
- We explore applications of the resulting techniques.
- We finish off the formal rules.

## 16.2 CASES OF REPEATED INHERITANCE

The Parent rule indicates that the inheritance graph of a set of classes may not contain any cycles. It is perfectly possible, however, for two classes to be connected through more than one path. The figure on the next page provides two examples.

← *"Parent rule", page 176.*




***Direct and indirect repeated inheritance***

Here is the definition:


**Repeated inheritance, ancestor, descendant**

**Repeated inheritance** occurs whenever (as a result of multiple inheritance) two or more of the ancestors of a class *D* have a common parent *A*.

*D* is then called a **repeated descendant** of *A*, and *A* a **repeated ancestor** of *D*.

*Why does the first sentence of the definition use the word "ancestor" rather than "proper ancestor"?*

As shown by the two examples in the figure, *D* can repeatedly inherit from *A* directly **(a)** as well as indirectly **(b)**.

The simplest case, called **direct repeated inheritance** and making *D* a **repeated heir** of *A*, occurs when *D* lists *A* in two or more Parent clauses:

```
class D inherit
        A rename … redefine … end
        A rename … redefine … end
        … Rest of class omitted …
```

The second case, **indirect repeated inheritance**, arises when at least one parent of *D* is a proper descendant of *A*, and at least one other is a descendant of *A*.

The discussion so far has neglected the generic parameters, if any, of the repeated ancestor *A*. In reality, a Parent is not just a class but a Class_type — a class name possibly followed by actual generic parameters. Uses of *A* as repeated ancestor with different actual generic parameters still cause repeated inheritance (*D*'s ancestors have a common parent class even though the corresponding Parent types are different); this case will show up in the consistency constraints and semantic rules.

→ *"THE CASE OF CONFLICTING GENERIC DERIVATIONS", 16.7, page 442.*

## 16.3 THE TWO QUESTIONS OF REPEATED INHERITANCE

Repeated inheritance, although not a tool for beginners, is in fact a simple mechanism if approached properly. Only two issues arise, the answers to which make up this section and the next (and the principal new concepts of this chapter): does a feature inherited twice yield one feature, or two? If it yields two, which one should dynamic binding trigger?

First, the matter of repeatedly inherited features:

**The first question of repeated inheritance:**
**Fate of a repeatedly inherited feature**

Given a feature from a repeated ancestor, what feature or features does it yield in a repeated descendant?

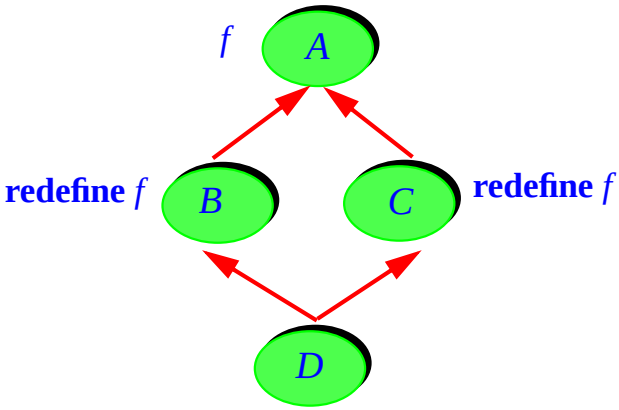
In the absence of repeated inheritance, the situation was simple: if *Y* is a descendant of *X*, every feature of *X* yields at most one feature of *Y*. But now things are not so clear any more. In either of the preceding pictures, what should *D* get out of a feature *f* of *A*: one feature, or two?

*Usually one, but the Join mechanism (10.21 and 10.22) may merge several inherited features.*

The second question arises from the combination of repeated inheritance and dynamic binding. Assume that in a case of indirect repeated inheritance, **b** on the last figure, one or both of the branches provides a new version for *f*:

*The problem arises as soon as* one *branch redefines f; for symmetry we assume both do.*

*Conflicting redefinitions*




Eiffel's dynamic binding policy (which suffers no exception) tells us that the call will use the version of *f* applicable to *D* (regardless of the declaration of *a*). But now we have two such versions. Hence:

**The second question of repeated inheritance: Ambiguities under dynamic binding**

Given a feature repeatedly inherited under two different redeclarations, which one should a call execute if its target is statically of the repeated ancestor type and dynamically of the repeated descendant type?

Developing answers to these two questions is our principal task for this chapter. Both answers will turn out to be remarkably simple, but we must study the issues carefully before we can deduce the answers.

Also remarkable is that we can for a large part tackle the two questions separately, as they have little bearing on each other.

## 16.4 SHARING AND REPLICATION

Consider first the question of the fate of a repeatedly inherited feature. In the common descendant, does it yield one feature, or two?

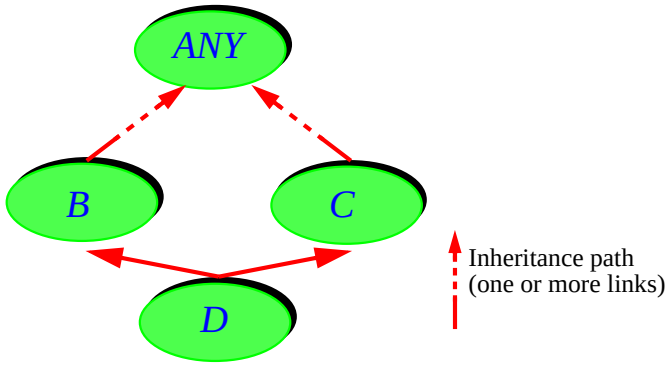
We cannot settle for a single, universal answer. Depending on the context, either solution may be the right one, and you will need some leeway for choosing between them in any particular case:

1 • In some circumstances you may use repeated inheritance precisely because you like a feature of an ancestor so much that you want two of it.

2 • Often, however, one copy is enough. For example, the scheme illustrated on the figure above may arise when you write both *B* and *C* (the intermediate ancestors) as heirs of *A* because each needs *A*'s features, such as *f*; *D* needs the new features introduced by *B* and *C*, but only one copy of *A*'s features.

An extreme example of case 2 is the universal class *ANY* of the Kernel Library, an obligatory ancestor of all Eiffel classes. The presence of *ANY* means that any use of *multiple* inheritance is automatically a case of *repeated* inheritance, since even if the two parents, *B* and *C* on the figure below, do not explicitly list a common ancestor, they are automatically descendants of *ANY*, making *D* a repeated descendant of *ANY*.

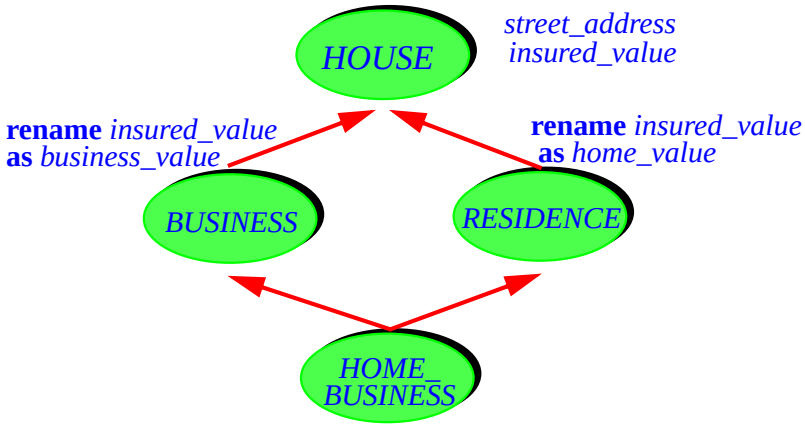
← *"ANY", 6.6, page 172; see also chapter 35 for more details.*




***Any multiple inheritance causes repeated inheritance from ANY***

For any non-trivial Eiffel system, the repeated inheritance structure induced by *ANY*, if we ever tried to draw it, would be rather luxuriant. In most cases, useful as the features of *ANY* are, you would not want your classes to inherit multiple copies of all of them.

The language could of course force you to choose one of the solutions, 1 or 2, globally for all the features from a given repeated ancestor. (This is roughly the C++ approach, through the notion of "virtual base class".) But such a solution would be too restrictive: you may need replication for some features and sharing for some others. The Eiffel policy uses the expected default, sharing, but lets you choose the other possibility, replication, for any specific feature. The criterion is straightforward: is the feature inherited under a single name, or different names?




***Homes, businesses, and home businesses***

*The* **rename** *subclauses shown will produce the desired effect:: sharing for street_address, replication for insured_value. See next.*

To see why we need such flexibility, consider the simple example, illustrated by the figure. In a system used by an insurance company, a class *HOUSE* has heirs *RESIDENCE* and *BUSINESS*. A special class *HOME_BUSINESS* handles the case of people who run a business from their house; it is legitimate to write this class as an heir to both of the previous two. The features of *HOUSE* include attributes *street_address* and *insured_value*. For the street address, an instance of *HOME_BUSINESS* should inherit a single attribute; but for *insured_value* it needs two, since the insured value may be different for the two viewpoints.

*The same reasoning will apply to routines, such as update_street_address and change_insured_value.*

The repeated inheritance mechanism gives you the desired flexibility: when writing a repeated descendant such as *HOME_BUSINESS* you can decide which repeatedly inherited features will yield single features ("sharing") and which duplicate features ("replication").

The policy is the simplest possible, and follows once again from the **no overloading** principle: within a class, make sure every name denotes a feature and only one. The principle implies that if the inherited features have the same final name, they *must* denote the same feature, and so will cause sharing; if they have different final names, they must yield different features, and will cause replication.

← "NAME CLASHES", 10.23, page 290.

This is the answer to the first question of repeated inheritance, enabling us to introduce the principal rule of this chapter:


### Repeated Inheritance rule

Let $D$ be a class and $B_1$, … $B_n$ ($n \geq 2$) be parents of $D$ based on classes having a common ancestor $A$. Let $f_1$, … $f_n$ be features of these respective parents, all having as one of their seeds the same feature $f$ of $A$. Then:

1 • Any subset of these features inherited by $D$ under the same final name in $D$ yields a single feature of $D$.

2 • Any two of these features inherited under a different name yield two features of $D$.

*Since A may be any ancestor, not just a proper one, the rule applies to direct repeated inheritance, where $B_1$, … $B_n$ are all the same as A, as well as to the indirect case.*

This is the basic rule allowing us to make sense and take advantage of inheritance, based on the programmer-controlled naming policy: inheriting two features under the same name yields a single feature, inheriting them under two different names yield two features.

**Sharing, replication**

A repeatedly inherited feature is **shared** if case 1 of the Repeated Inheritance rule applies, and **replicated** if case 2 applies.

---- REMOVE A fine point about the rule's phrasing: it refers to "parents *based on classes* having a common ancestor" rather than "parents having a common ancestor" because a Parent is syntactically not a class but a type. With **class** *D* **inherit** *P*... we are looking at ancestors not of *P* but of *P*'s base class.

←Parent *syntax: page 169.*

Also, like all semantic rules, this one assumes that class *D* is valid. Otherwise. of course, we would get no feature at all in either case.

The Repeated Inheritance rule applies to attributes as well as to routines. It provides the designer of a repeatedly inheriting class with all the needed flexibility through proper choice of names:

- If two or more of the parents of *D* happen to have a common ancestor *A*, and you do not take any particular renaming action, each feature of *A* will yield just one feature of *D*. This will usually be what you want in simple cases, such as repeated inheritance from *ANY* as mentioned above. The rule also renders harmless a common oversight: making *A* a parent of *D* because *D* needs the features of *A*, forgetting that among the other parents of *D* one is already a descendant of *A*.
- If, however, you want two or more versions of a repeatedly inherited feature, just make sure that it is inherited under different names. This is the modern version of the loaves and fishes miracle: if you have one of a good thing, you may turn it into as many as you like, just by asking.

*"They took up twelve baskets full of the loaves, and of the fishes", Mark 6:43. Scholars believe* Loaf *and* Fish *to be ancient Aramean for* attribute *and* routine. *See Proc. ALOOF 3 (Archaeo-Linguistic Object-Oriented Forum), Acapulco, 1998, pp. 798-923.*

To determine which of the two cases applies, the only criterion that matters is the **final name** of the feature in *D*. It will be affected by any renaming performed in *D* itself as well as in intermediate ancestors between *A* and *D*. This means that, as the author of *D*, you are the master when it comes to setting the fate of a feature *f* coming from an indirect repeated ancestor through parents *B* and *C*:

- If *f* has the same name in *B* and *C*, *f* will normally be shared, but you may force replication by renaming one of the inherited versions, or renaming both forms with different names.
- If there has been some renaming between *A* and *D*'s parents, *f* will normally be replicated, but you may force sharing by renaming both inherited versions to the same name.


The sharing case of the Repeated Inheritance rule enables us to understand fully the notion of name clash and the prohibition of name clashes. The guideline (made formal by the Join rule) stated that a name clash is permissible only in three cases:

←*After the definition of "Name clash" on p. 291.*

1 • At most one of the clashing features is effective.

2 • The class redefines all the clashing features into a common version.

3 • The clashing features are the same feature, inherited without redeclaration from a common ancestor.

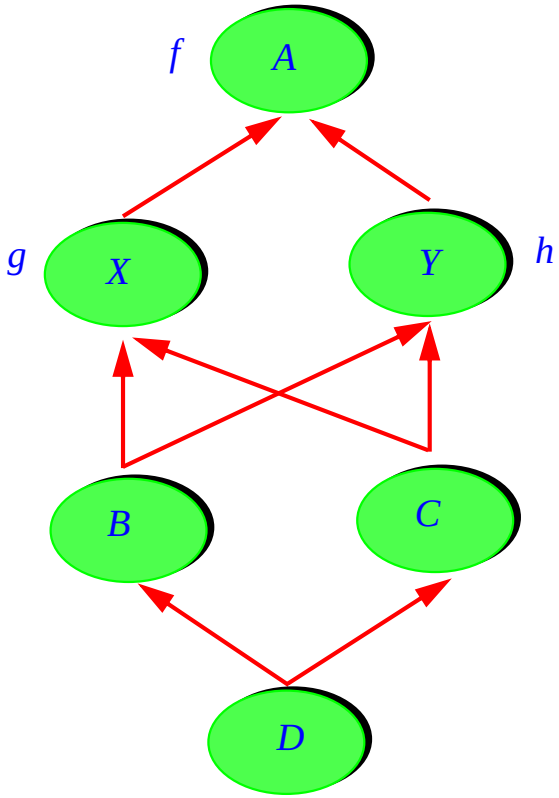
It's the Repeated Inheritance rule that gives its meaning to the last case: even though there is an appearance of name clash because two parents *B* and *C* of a class *D* have a feature with the same name, in reality they are the **same feature**, inherited from a common ancestor *A*. If *D* inherits it in both cases under the same name, there is no real name clash; the sharing part of the Repeated Inheritance rule implies, naturally enough, that *D* will get the feature from *A*, exactly as if it had been declared as a direct heir of *A* without any intermediate classes.

This assumes of course that the feature is not redefined anywhere, otherwise it wouldn't be the "same" feature. The next section will study the case of conflicting redeclarations.

One more general observation is in order on the scope of the Repeated Inheritance rule. As you will have noted from the definition, the rule only applies if *f* is the common seed of the features under consideration or, equivalently, if *A* is their origin. Remember that the *seed* of a feature is its original version in the most remote ancestor (the feature's *origin*) where it appears, regardless of any redeclaration or renaming that it may have endured between that ancestor and the current class.

← *For the precise definitions see "Origin, seed", page 305. Joining a set of features gives all of them a new seed and origin.*

***Only the seed and origin matter***

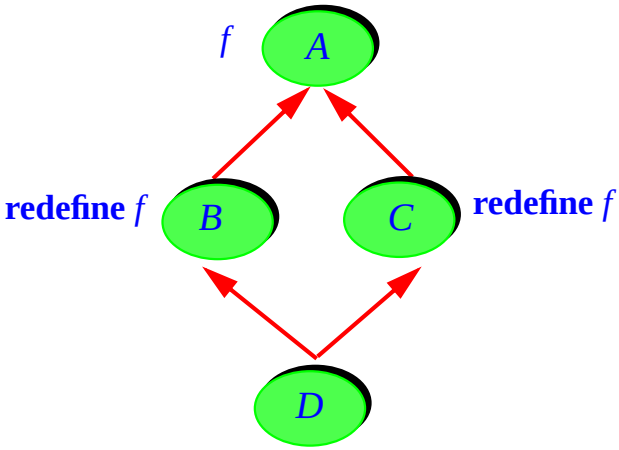



This requirement that *A* be the origin of *f* is important. Without it, as illustrated by the preceding figure, the Repeated Inheritance rule would be ambiguous. In the figure, *f* is a feature of *A*, but it is also a feature (an inherited one) of *X* and *Y*. All three classes are repeated ancestors of *D*. To infer sharing or replication from the rule, we need to know what repeated ancestor to consider. The rule's phrasing answers this question precisely: for *f*, the only relevant ancestor is class *A*, the origin of that feature. Similarly, to determine the fate of *g* and *h*, you must apply the rule (respectively) to *X* and *Y*, assumed to be the origins of these features.

## 16.5 THE CASE OF REDECLARED FEATURES

The Repeated Inheritance rule would define all we need to know about repeated inheritance were it not for the second question raised at the beginning of this chapter: ambiguities under dynamic binding.

Here is the picture again. We assume that both *B* and *C* redefine *f*:




***Conflicting redefinitions***

If *D* inherits the two versions under the same name, it gets a single feature (sharing); otherwise, two different features (replication). But then what happens in a call of the form *a*.*f*, where *a* is declared of type *A* but is attached, at run time, to an instance of *D*?

The **sharing** case is easy because even in the absence of dynamic binding we have a problem: *D* gets two features with the same name. We know this case! It's a name clash. That the two features originally come from a common seed, the *A* version, doesn't matter here: at the level of *D* they are now *different features*.

← *"NAME CLASHES", 10.23, page 290*.

Studying the join rule has taught us that in such a conflict:

← *"Join rule", page 309*.

- If all of the variants, or all but one, are deferred and still have a single signature, there is no particular problem. They will all be joined, and live happily ever after as a single feature.

  If some intermediate redefinition has led to different signatures, you may still use a join, but it will require a redefinition (or effecting) to a feature whose signature matches all the inherited ones.

- If two or more are effective, the name clash would make the class invalid. In the general case we could resolve it by renaming, but here this would mean feature *replication* (the case discussed next), whereas we are explicitly assuming *sharing*, meaning all variants have the same final name. To remove the name clash we have to force a join by undefining all the effective features except at most one.

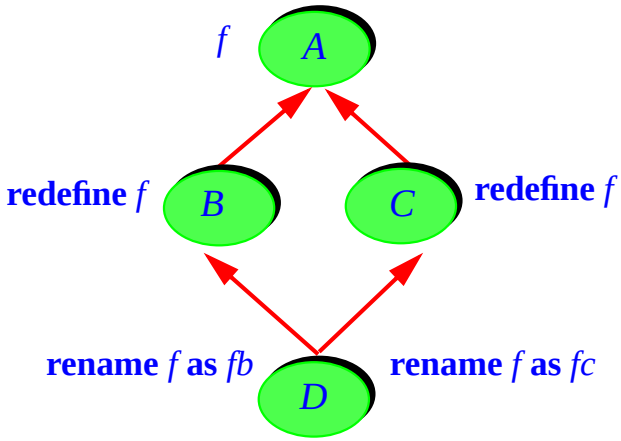
So here if both redefined versions are effective you must write *D* as either

```
class D inherit
        B
                undefine f end
        C
… Rest of class omitted …
end
```

or the form that undefines the *C* version instead. You may also redefine both. If you do not include such an undefinition or redefinition, the class is invalid. We don't need any new validity constraint to express this requirement: the rules of the Feature Adaptation chapter took care of it.

This addresses the sharing case. But what if (as in the following figure) one or both features are renamed, causing replication?

***Redeclaration and replication***




Because *D* renames the two inherited versions of *f*, we have a case of replication: *f* yields two features in *D*, called *fb* and *fc*. These features are truly different, since both *B* and *C* redefine their inherited versions of *f*. Note for generality that:

- The example assumes redefinition, but it would arise in any case of **redeclaration**, including conflicting effectings of an inherited feature.
- For symmetry, the example assumes that both *B* and *C* redefine *f*, but the problem would arise in the same way if one of these classes redefined the feature and the other kept the original.
- The renaming takes place at the level of *D*, but it could occur anywhere above, or for only one of the features, as long as the final names in *D* are different, causing replication.
- The problem will also arise, even without redefinition, in the case of **attributes**, as will be seen next.

←*Redeclaration covers redefinition and effecting. See* "Redeclare, redeclaration", page 257,

Only dynamic binding with a target of static type based on *A* and dynamic type based on *D* causes a problem. There is nothing ambiguous about calls with a target entity *d1* of type *D*:

```
d1: D
…
        -- Attach d1 to an object of type D:
create d1

d1.fb; d1.fc
        -- A call of the form d1.f would be invalid,
        -- since D has no feature of name f.
```

The first call will trigger execution of the version of *f* redefined in *B*, and the second will use the *C* version. Nothing new or surprising.

No difficulty arises either with polymorphism and dynamic binding applied to entities of types *B* or *C*:

```
b1: B; c1: C
…
create d1

        -- Attach each entity to an object of type D:
b1:= d1; c1 := d1

        -- The calls of interest:
b1.f ; c1.f
```

To keep things simple, this example assumes that *f* is a procedure without arguments, that the classes involved are all non-generic — so that they are also types — and that *D* has no creation procedure. Also, the classes involved are all reference (non-expanded); if *B* or *C* were expanded, *D* would not conform to them, making the assignments invalid.

The two assignments are **polymorphic**, allowing *b1* and *c1*, although declared of types *B* and *C*, to become attached to an object of type *D*. The type rules permit this since *D* conforms to both *B* and *C*. Complementing polymorphism, **dynamic binding** commands that the version executed in each case is the one redefined by the ancestor closest to *D*. This means that (on the last line) the first call will trigger the *B* version and the second will trigger the *C* version. Still no particular problem.

→ *"POLYMORPHISM", 22.11, page 598*; *"DYNAMIC BINDING", 23.12, page 630*.

Where the situation becomes potentially ambiguous is if you use polymorphism and dynamic binding to call *f* on an entity *a1* of type *A*, the repeated ancestor, as in

```
a1: A; d1: D
…
create d1
        -- Attach the entity to an object of type D:
a1 := d1
        -- The call of interest:
a1.f
```

Dynamic binding rules indicate that the call should trigger the version of *f* applicable to the actual object, which here is an instance of *D*. But there are two such versions of *f* resulting from the *B* and *C* redefinitions, and none of them is a priori better than the other.

Here is for example how *B*, *C* and *D* (deprived of any properties not relevant to this discussion) might appear:

```
class B inherit
        A redefine f end
feature
        f is do print ("Yes!") end
end
```

```
class C inherit
        A redefine f end
feature
        f is do print ("No!") end
end
```

```
class D inherit
    B
        rename f as fb end
    C
        rename f as fc end
end
```

*WARNING*: *D as given is invalid. As explained next, one of the branches must use non-conforming inheritance.*

Will the call *a1.f* print "Yes!", obeying *B*, or will it obey *C* and print "*No!*"?


One may imagine various language solutions:

- We could rely on the order of the Parent clauses for *B* and *C* in *D* But this is not acceptable: by reversing the order of parents, an innocuous editing change, you would change the semantics of the class. Besides, such a convention only makes sense for simple cases such as the above; with more levels of repeated inheritance, the "order" of ancestors becomes murky. In the earlier example, if *B* lists its parents in the order *X*, *Y*, but *C* lists its parents in the reverse order, what is the order of *X* and *Y* as ancestors of *D*?

← *Figure page 433.*

- We could require the class author to "select" one of the variants for use in dynamic binding, through a special language construct, every time such a conflict arises. This solution works and was indeed used in Eiffel 3. But further reflection has shown that a simpler approach was possible.

What makes that approach simpler is that it is more radical: *disallow polymorphism* whenever it could cause dynamic binding trouble. We suddenly remember that we have a straightforward way to disallow polymorphism when we don't want it: instead of plain polymorphic inheritance, use **non-conforming inheritance**, also known as *expanded inheritance* because it builds on Eiffel's notion of expanded class and indeed uses the keyword **expanded**.

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

A simple way to guarantee that an inheritance branch will not induce conformance is indeed to add that keyword to the corresponding Parent clause: if you declare a class as

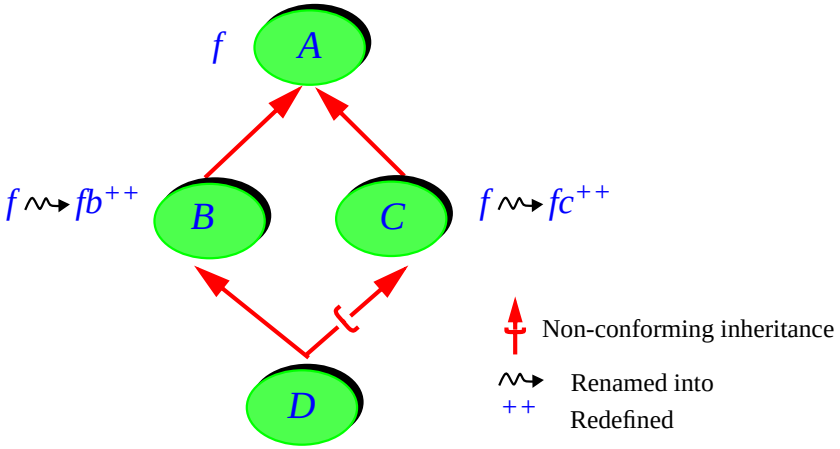
```
class D inherit
        expanded C
… No other parents …
```

then attachments such as *c1* := *d1*, with *c1* of type *C* and *d1* of type *D*, are not permitted. Without the **expanded** qualification, they would be valid.

*This discussion still assumes that the classes involved are not themselves expanded classes.*

To avoid the ambiguity in the previous example it suffices to guarantee that only one of the two branches is polymorphic, by declaring *D* as

```
class D inherit
        B
                rename f as fb end
        expanded C
                rename f as fc end
end
```




***Removing dynamic binding ambiguity through non-conforming inheritance***

This means that we have chosen only one of the two branches as permitting polymorphic attachment. So in the kind of situations seen above as causing trouble with polymorphism and dynamic binding:

```
a1: A; d1: D
…
a1 := d1; a1.f
```

There is no ambiguity any more: *d1* conforms to *a1* in only one way, through *B*, so the feature *f* to be applied is the *B* version, *fb*.

The approach just studied implies resolving all potential dynamic binding ambiguities in favor of the same parent, *B* in the example. In rare cases you might want *a1.f* to call the *B* version for some features *f*, but *a1.g* to use the *C* version for a particular *g*. We will see later in this chapter how to adapt the technique to this case.

→ *"RETAINING VICTORS FROM ALTERNATIVE BRANCHES", 16.11, page 452*.

The scheme works just as well for direct repeated inheritance:

```
class D inherit
        expanded A
                rename f as f1 end
        A
                rename
                        f as f2
                redefine
                        f2
                end
… Rest of class omitted …
end
```

*With this form the version for dynamic binding is the redefined one, f2. Moving* **expanded** *to the first branch would select instead the original version, under the name f1.*

You **must** mark one of the two Parent clauses involving *A* as cases of non-conforming inheritance — for example by using **expanded** *A* as here — to make valid such a case involving replication and redeclaration of one or more features.

This is the basic mechanism for resolving conflicts in such cases. Note that using an **expanded** qualification for one of the parent branches is the means, not the end. What the rule will state is that **conformance** may hold along at most one branch. If an inheritance branch is non-conforming for some other reason, then it does not create any conflict and there is no need for the explicit **expanded** qualification. In particular, if *C* is an expanded class — so far this section has assumed that none of the classes involved were expanded — the applicable conformance rules imply that *D* will not conform to *C* in spite of inheriting from it, so you may dispense with any special qualification, writing simply

← *The assumption was made on page 436.*

```
note
        note: "This version of the example assumes an expanded class C."
class D inherit
        B
                rename as fb end
        C
                rename f as fc end
end
```

The rule introduced by this discussion is the Repeated Inheritance Consistency constraint. The rule will be formulated precisely at the end of this chapter, but it's basically what we have just seen.

→ *"Repeated Inheritance Consistency constraint", page 458*

To gain a full understanding, we must now check what happens in two specific cases: attributes and conflicting generic derivations.

## 16.6 THE CASE OF ATTRIBUTES

The last example involved a feature *f* which was a routine. For attributes, a similar problem arises even in the absence of redefinition.

> You may redefine an attribute, but this is only useful for type redefinition, since the redefined version must still be an attribute. See condition 6 of the Redeclaration rule.

← *"Redeclaration rule", page 307.*

The cause of ambiguity here is that a replicated attribute will yield two fields rather than one in the repeated descendant. Then, with dynamic binding, a reference to such a replicated attribute may become ambiguous in the same way as a reference to a multiply redeclared routine.

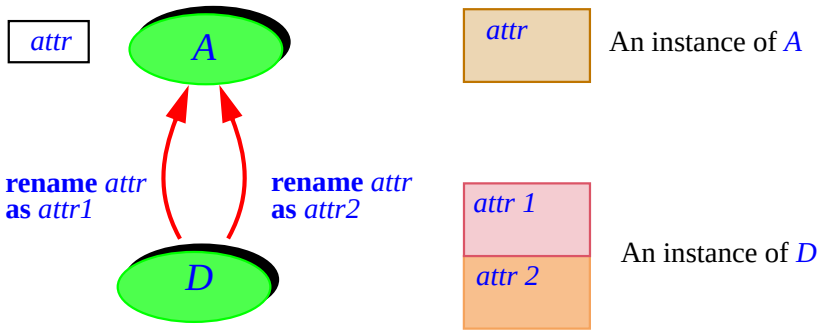
This may occur even with direct repeated inheritance of a class *D* from a class *A*, with a scheme such as this:

```
class A feature
        attr: SOME_TYPE
        some_procedure is do print (attr) end
end
```

```
class D inherit
        A
                rename attr as attr1 end
        A
                rename attr as attr2 end
end
```

*WARNING: D as shown is invalid. Using non-conforming inheritance will make it valid; see the next version.*

A direct instance of *A* has only one field, corresponding to *attr*. In an instance of *D*, however, *attr* yields two fields, for *attr1* and *attr2*:




***Attribute replication***

As in the case of conflicting redeclarations, it is not clear which one of the fields the following should print:

```
a1: A; d1: D
…
create d1; a1 := d1; a1.some_proc
```

Because any new attribute implies a new field in every instance of the applicable class, we may view replication, for attributes, as implying a kind of implicit redefinition, similar in its effects to the explicit redefinition of routines.

Similar problem, same solution: whenever the Repeated Inheritance rule implies replication of an attribute, the Repeated Inheritance Consistency constraint will require that one of the inheritance paths involve non-conforming inheritance, as in

→ *"Repeated Inheritance Consistency constraint", page 458*

```
class D inherit
        A
                rename attr as attr1 end
        expanded A
                rename attr as attr2 end
end
```

## 16.7 THE CASE OF CONFLICTING GENERIC DERIVATIONS

(This section, addresses the semantics of a rare case and may be skipped on first reading.)

Like attribute replication, different generic derivations from a common generic ancestor cause a form of implicit redefinition.

*→ Chapter 12 studies generic classes and generic derivations.*

It is not hard to devise a simple example. Assume that *A* is generic, with one formal generic parameter *G*, and has a feature *f* whose signature involves *G*:

*← The signature of a feature is the specification of its argument and result types. See "Signature, argument signature of a feature", page 149*

```
class A [G] feature
        f(x: G)is … Routine body omitted … end
end
```

```
class B inherit
        A [INTEGER]
end
```

```
class Cinherit
        A [REAL]
end
```

What the body of *f* does is irrelevant; so is the exact nature of *f* — procedure as above, attribute or function — as long as *f*'s signature depends on *G*. The texts of classes *A*, *B* and *C* as shown only include the properties relevant to this discussion.

The different generic derivations of *A* used in the Parent parts of *B* and *C* cause *f* to have different signatures in these classes:

```
in B: [ ], [INTEGER]
in C: [ ], [REAL]
```

This means that the name *f*, in these two classes, denotes **different features**: a feature is defined not only by its specification (assertions) and its implementation, but also by its signature.

What then if you want to write a class *D* as heir to both *B* and *C*? This creates a conflict, as in the two previously studied cases (routine redefinitions and attributes). Because the features are different, sharing is impossible in this case, but the same replication-based solutions are available as in the previous two:

*← The reasons that preclude sharing were analyzed at the beginning of 16.5, page 434.*

1 • Using replication and making sure that at most one of the inheritance paths uses conforming inheritance.

2 • Letting one of the versions override the other through undefinition.

The second solution requires special care here because the signatures are different. The problem is that if a version overrides the other it must have a conforming signature; but this may not be true because of conflicting generic derivations. In the above example, indeed, the signatures of the *B* and *C* versions are incompatible since neither of the types *INTEGER* and *REAL* conforms to the other. The only solution is to undefine both features and provide a fresh redeclaration in *D*. Here, in the absence of a useful common descendant to *INTEGER* and *REAL*, that fresh feature may only be of the form

```
f (x: NONE) is do … Some routine body … end
```

and hence cannot do anything useful with its argument *x*. (Recall that *NONE* is a common descendant of all classes, but has no exported feature.)

← "NONE", 6.7, page 175.

In more favorable cases, one of the actual generic parameters used for generic derivations of *A* in *B* or *C* will conform to the other; then you may use its version of *f* to overtake the other's. Redefinition into a version whose signature conforms to both (if possible not just through *NONE*) will also work.

## 16.8 KEEPING THE ORIGINAL VERSION OF A REDEFINED FEATURE

The most novel aspect of the Repeated Inheritance rule is the replication case: here for the first time there is a way for one feature of a parent to yield two or more features in an heir.

Among other applications, this mechanism enables us to "redefine our feature and eat it": provide a new version of an inherited routine, but retain the original as well.

In the majority of cases, you do not need repeated inheritance to achieve this goal, because the most common use of the original version is to help write the redefined version. We have seen the simple language mechanism that directly addresses this need: Precursor. You will simply write the redefinition of a routine as

← "ADDING TO INHERITED BEHAVIOR: PRECURSOR", 10.24, page 293.

```
your_routine (args: …)
        do
                "Something else"
                Precursor (…)
                "Yet something else"
        end
```

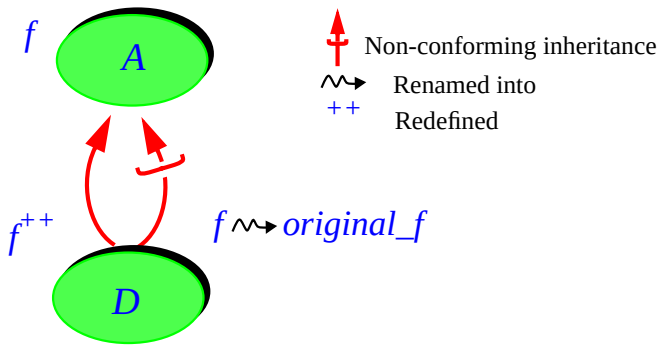
With this technique — applicable only to routines, not attributes — the inherited version does *not* remain a feature of the new class: all you have is its implementation, usable only in the corresponding redefinition.

In some cases you may want the heir class to include both the new version and the old. This scheme is not commonly useful, if only because it assumes that the old version still makes sense in the new context — do not forget, in particular, that if it is an exported routine it must preserve the new invariant as well as the old one! — but the need does occasionally arise.

← *"Unfolded form of an assertion", page 281*.

When this happens, the replication mechanism of repeated inheritance will provide the solution. The scheme is simple (see the figure below): if you want class *D*, an heir of *A*, to redefine *f* while retaining the original version, make *D* inherit a **second** time from *A* — the direct form of repeated inheritance is usually appropriate in this case — and rename *f* to a different name, without redefinition, along that second branch.

***Keeping the original version through replication***




To satisfy the Repeated Inheritance Consistency constraint, you will need to make one of the two inheritance branches non-conforming.This will usually be the second branch (the one that serves to retain the original version) since we will want the redefined version to serve as "the version of *f*" for *D* and its descendants under dynamic binding.

The outline for *D* is:

```
class D inherit
        A
                redefine f end

        expanded A
                rename f as original_f end

… Rest of class text omitted …
end
```

Although this is the common setup, you are free to choose a different combination of redefinition and renaming.

It's a very simple setup. You can use it whenever Precursor doesn't suffice because you want to keep the original as a feature of the new class with all the associated privileges. For example:

```
class MONEY_MARKET_ACCOUNT inherit

        SAVINGS_ACCOUNT
            redefine compute_interest end

        expanded SAVINGS_ACCOUNT
            rename
                compute_interest as
                    compute_interest_as_for_plain_savings
            export
                {NONE} compute_interest_as_for_plain_savings
            end

… Rest of class text omitted …
end
```

← *Compare with the examples in the discussion of* Precursor *in* 10.24, page 293.

This class illustrates what to do if you want to keep the original version, here under the name *compute_interest_as_for_plain_savings*, for internal purposes only: hide it from clients at the point of inheritance through a New_exports clause that stipulates access to no useful clients. This is required n particular if the original version does not preserve the invariant of the new class.

← *"Adapting the export status of inherited features", , page 200.*

## 16.9 USING REPLICATION: COUNTERS AND ITERATION

The technique studied in the previous section relies on the Repeated Inheritance rule's automatic mechanism for duplicating routines and attributes. Let's see a couple more applications of this possibility.

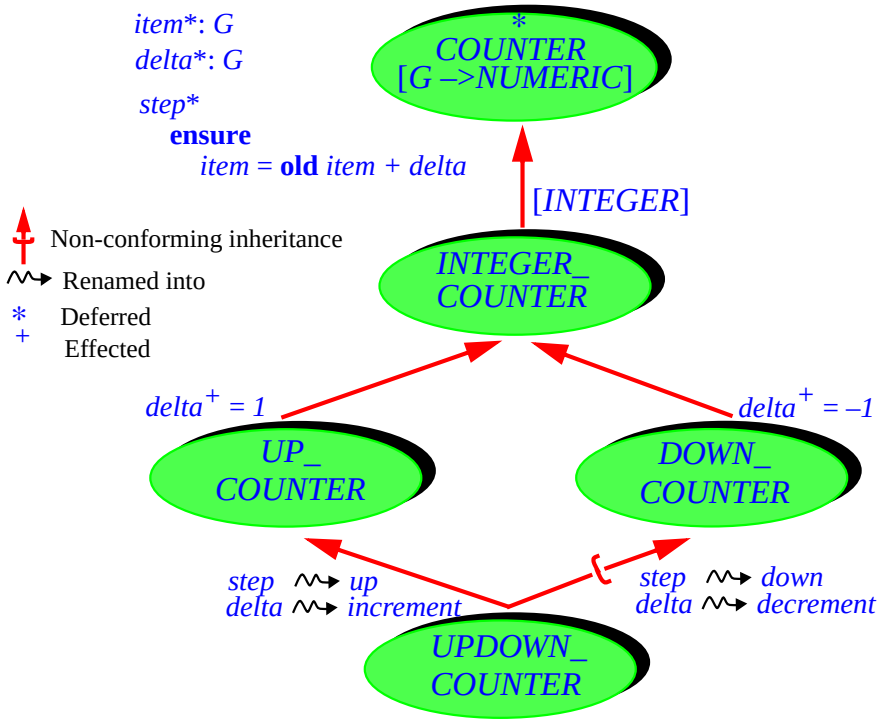
The first example is a pedagogical exercise (due to Christine Mingins). The inheritance hierarchy is shown on the following figure. We have a general notion of *INTEGER_COUNTER* with

- A query *item* giving the current value associated with the counter.
- A procedure *step* with no argument, to advance the counter by one step.
- A query *delta* giving the amount by which a *step* will change the value.

For more generality we can make *INTEGER_COUNTER* inherit from *COUNTER* [*INTEGER*] and introduce these three features at the level of the generic class *COUNTER*

Right from the start (in *COUNTER*), procedure *step* should have a postcondition stating *item* = **old** *item* + *delta*.

***Counters, up, down and both***




The figure is explicit enough that we don't need to write the actual class texts. We have two variants of *INTEGER_COUNTER*, representing counters that increment their value by +1 and –1. It suffices in *UP_COUNTER* to effect *delta* as returning *+1*, and *–1* in *DOWN_COUNTER*. Procedure *step* should be effected to execute *item* := *item* + *delta*; this may be done in *INTEGER_COUNTER* or even *COUNTER*.

Then we want a notion of counter that can count both up and down, with two procedures *up* and *down*. It suffices that *UPDOWN_COUNTER* inherit from both *UP_COUNTER* and *DOWN_COUNTER*, renaming *step* to *up* and *down*, and *delta* to *increment* and *decrement* (these two words being used as nouns, as in "an increment", not as verbs as in "increment this"). Both cases are valid uses of inheritance: an updown counter is definitely an up counter, and a down counter as well. For dynamically bound uses of *step* and *delta* on updown counters known statically as just counters, we choose the "up" interpretation, so the inheritance from *UPDOWN_COUNTER* to *DOWN_COUNTER* is non-conforming. The machinery of repeated inheritance gives us exactly what we need thanks to replication.

If the postcondition of *step* is to make sense in both versions *up* and *down* of this feature, it is critical that the redeclarations of *step* go hand in hand with those of *delta*: the postcondition must mean *item* = **old** *item* + *increment* in *UP_COUNTER* and *item* = **old** *item* + *decrement* in *DOWN_COUNTER*. This will require a semantic clarification in the next section.

→ *"Replication Semantics rule", page 451*.

The second example deals with multiple iterations. The agent mechanism actually provides a more dynamic way to address this issue, but the technique described here can still be interesting in some cases.

→ *Chapter 27 covers agents and include several iteration examples.*

Consider an **iterator** class providing a way to perform certain operations on every element of a certain structure. These operations are denoted in the iterator class by deferred routines; descendants will effect them to represent the actual operations needed in a particular iteration case. For example a class *LINEAR_ITERATION* (such as provided by the iteration cluster of EiffelBase) may include a procedure *do_until* with this general form:

← *Compare to until_do in "PARTIALLY DEFERRED CLASSES AND PROGRAMMED ITERATION", 10.15, page 271.*

```
do_until (s: TRAVERSABLE [T])
            -- Iterate on s,  up to and including
            -- the first item satisfying test.
        do
            from
                start (s); prepare (s)
            until off (s) or else test (s) loop
                action (s); forth (s)
            end
            if not off (s) then action (s) end; wrapup (s)
        end
```

Any effective descendant of *LINEAR_ITERATION*, describing an iteration scheme over a specific kind of data structure — for example a list implemented by an array with a current position *position*—, will effect *start*, *forth* and *off* to provide, for the corresponding iterative structure:

- An implementation of *start*, bringing the cursor iteration to the first position; in the array case, it will be the assignment *position* := *1*.
- An implementation of *forth*, to advance the cursor by one position: for arrays, *position* := *position + 1*.
- An implementation of *off*, to query whether we have exhausted the list of meaningful cursor positions: for arrays, the test *position > count*, where *count* is the number of occupied positions.

The class providing these effective declarations may be a class *LIST_ITERATION*. All that remains to do for a descendant needing actual iterations is to effect the routines describing the actions and tests to be performed on every list element: *prepare*, *test*, *action* and *wrapup*.

But what if a class needs **two** variants of the iteration mechanism? It is possible to use repeated inheritance from *LIST_ITERATION*, with sharing for the traversal routines (*start*, *forth*, *off*) and replication for the operation routines *prepare*, *test*, *action* and *wrapup*, which need separate versions.

An example is an an application that handles lists of atomic particles, as described by the class

```
class PARTICLE feature
        mass: REAL; speed: VECTOR
        positively_charged: BOOLEAN
         … Other attributes and routines …
end
```

where the lists are sorted by increasing mass. The application needs both to

1 • Print the mass of all particles in a list, up to and including the first positively charged one.

2 • Compute the total vector speed of the first fifty particles in the list and store it into an attribute *total_speed*. (To add speeds, we assume a procedure *add* in class *VECTOR*.)

Using repeated inheritance:

```
class PARTICLE_LIST_PROPERTIES inherit
        LIST_ITERATION [PARTICLE]
                rename
                        do_until as print_masses, prepare as do_nothing,
                        test as positive_test, action as print_one_mass,
                        wrapup as do_nothing
                end

        expanded LIST_ITERATION [PARTICLE]
                rename
                        do_until as add_speeds, prepare as set_speed,
                        test as at_threshold, action as add_one_speed,
                        wrapup as do_nothing
                end
feature
        positive_test (s: FIXED_LIST [PARTICLE]): BOOLEAN
                        -- Is particle at current cursor position in s positive?
                do
                        Result := s.item.positively_charged
                end

        print_one_mass (s: FIXED_LIST [PARTICLE])
                        -- Print the mass of particle at cursor position in s.
                do
                        print (s.item.mass)
                end
… Rest of class omitted …
```

## 16.10 THE SEMANTICS OF REPLICATION

The Repeated Inheritance rule specifies that a feature inherited repeatedly under two different names yields two features in the repeated descendant. We must clarify what replication entails, especially for routines. We need the corresponding semantic rule to ensure the correct functioning of both examples reviewed in the last section.

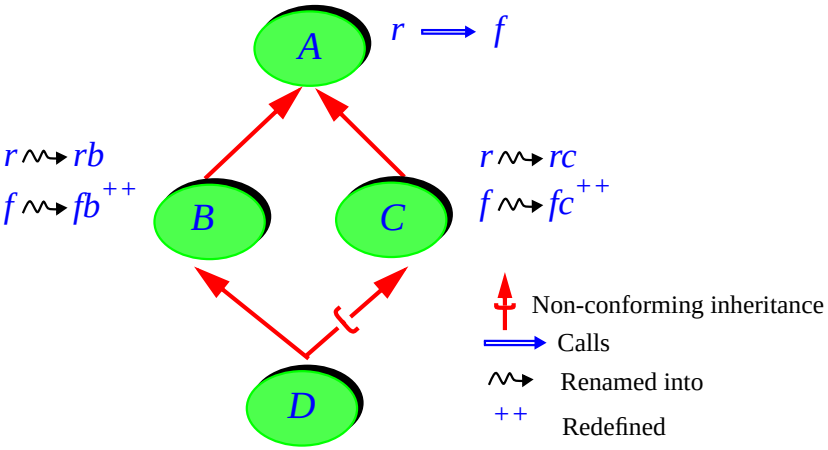
For attributes, we saw that replication is to be taken literally: instances of the common descendants will have two separate fields.

← *Figure "Attribute replication", page 441*.

For routines, we normally do not need to replicate any code. But a special case arises when *two* or more routines, calling each other, get replicated along the same branch.

Consider our usual diamond-shaped repeated inheritance structure, with two features *r* and *f* where *r* is an effective routine; *f* may be an attribute or a routine. We assume that *r* calls *f*:

***Multiple routine replications***




Both *r* and *f* get renamed differently along the two branches, so the Repeated Inheritance rule implies replication for both. In addition *f* gets redefined, so that the Repeated Inheritance Consistency constraint applies. The constraint states that at most one of the inheritance paths may support conformance; this is achieved here by using non-conforming inheritance from *D* to *C*. Viewed from *A*, then, the dynamic binding version of *f* in *D* is the *B* version, *fb*, in the sense that it's the feature called by *a1.f*, for *a1*: *A* dynamically attached to an object of type *D*.

→ *"Dynamic binding version", page 460*.

All this, as we have seen, also applies whenever *f* is an attribute, even if neither *B* nor *C* redefines it.

← *"THE CASE OF ATTRIBUTES", 16.6, page 440*.

Such situations raise a new problem: since *r* calls *f*, and *D* now has two versions of he original *f*, which one of these should *rb* and *rc* call?

Since the example include no redefinition for the features of seed *r* (*r*, *ra*, *rb*), the features *ra* and *rb* are just duplicates of the original *r*. If they are identical, they will call the same version of *f* in *D*; if so, that version should presumably, in keeping with the spirit of the Repeated Inheritance Consistency constraint, be *fb*, as *fc* comes from the non-conforming branch.

← *"Seed" was defined on page 305. A revised definition appears on page below.*

But is this right? Conceptually, *D* has two versions of *r* and two versions of *f*. The original property of *r* was that it called the corresponding version of *f*. There doesn't seem to be any good reason for a replicated version of *r* to call a version of *f* that results from a mutation of the original along a *different* inheritance branch.

A rare but illuminating case is for *f* to be the same routine as *r*:

```
r (args: …)
            -- A routine that may call itself recursively
      do
            …
            r (other_args)
      end
```

Assume *B* redefines *r* but (to keep things simple) *C* retains this original *A* version shown above. It seems reasonable to expected that the highlighted call to *r* should still be a recursive call, both in *C* and in *D*. Why should we call the *B* version? This seems a betrayal of the originally intended semantics, since the routine would now cease being recursive.

These reflections suggest that we should take the notion of replication seriously. Compiler writers, of course, will avoid physically duplicating the code of a routine whenever they can. But an Eiffel programmer should be able to believe the replication case of the Repeated Inheritance rule literally, as if it caused code duplication for a routine in the same way it causes field duplication for an attribute.

------ EXPLAIN


**Call Replication rule** *VMCR*

It is valid for a feature *f* repeatedly inherited by a class *D* from an ancestor *A*, such that *f* is shared under repeated inheritance and not redeclared, to include an unqualified call to a feature *g* of *A* or (if *f* is an attribute) to be the target of an assignment whose source involves *g* if and only if *g* is, along the corresponding inheritance paths, also shared.

If *g* were duplicated, there would be no way to know which version *f* should call, or evaluate for the assignment. The "selected" version, discussed below, is not necessarily the appropriate one.

The following rule expresses this property:

---

**Replication Semantics rule**

Let *f* and *g* be two features both repeatedly inherited by a class *A* and both replicated under the Repeated Inheritance rule, with two respective sets of different names: *f1* and *f2*, *g1* and *g2*.

If the version of *f* in *D* is the original version from *A* and either contains an unqualified call to *g* or (if *f* is an attribute) is the target of an assignment whose source involves *g*, the *f1* version will use *g1* for that call or assignment, and the *f2* version will use *g2*.

This rule (which, unlike other semantic rules, clarifies a special case rather than giving the general semantics of a construct) tells us how to interpret calls and assignments if two separate replications have proceeded along distinct inheritance paths.

Another way to state this is that replication may cause a form of **implicit redefinition**: if the replicated routine *r* calls a feature *f* that has been redefined, or is an attribute (in either case causing physical replication), then even if *r* has not been redefined anywhere in the process we must pretend that it has — to versions that call the corresponding versions of *f*.

If you review the examples of the preceding section, you will notice that they can only work under this rule:

- In the multiple counter example, the postcondition of *up*, inherited by *UP_COUNTER* from *COUNTER* as *item* = **old** *item* + *delta*, must use the version of *delta* applicable to *COUNTER*: *increment*, with value +1; for *DOWN_COUNTER*, the corresponding postcondition for *down* must use *decrement*, with value –1.

- In the multiple iteration example, *print_masses* and *add_speed*, both of them mere renamings of the general iteration procedure *do_until*, must use the versions of the list item operations *prepare*, *test*, *action* and *wrapup* applicable to its branch.

In both cases this means that even though the calling routine — *step*, the seed of both *up* and *down*, and *do_until*, the seed of both *print_masses* and *add_speeds* — is never explicitly redefined, it must take into account the separate redeclarations of features that it calls.

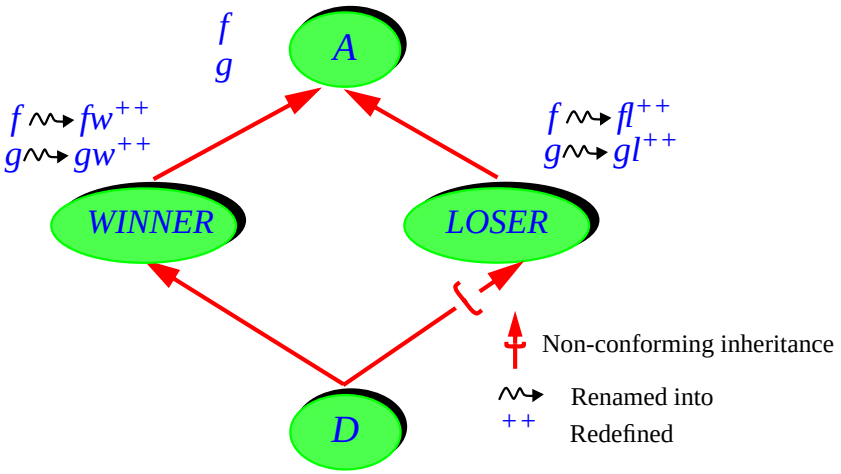
## 16.11 RETAINING VICTORS FROM ALTERNATIVE BRANCHES

This is a time for celebration: by now you know all the important concepts of inheritance and feature adaptation.

There remains to see a technique addressing a fine point of the combination between dynamic binding and replication (this section) and the precise rules for the concepts that we have studied but not yet formalized (next two sections ). All this is material that you can safely skip on first reading.

In studying the rules for redeclaration under repeated inheritance we have seen how to avoid ambiguities by forcing all branches but one to involve non-conforming inheritance. What if we want some of the versions for dynamic binding to come from another branch?

Let's consider again our basic figure for such cases:




***The winner and the loser***

*This is the figure of page 439, with a new feature f and different names for the intermediate classes.*

We have learned how to resolve the potential ambiguity of calls such as *a1*.*f* for *a1*: *A* dynamically attached to an object of type *D*: make sure that one of the inheritance paths involves non-conforming inheritance. Then the call will use the version from the other branch.

Once we have settled on where to use non-conforming inheritance, this policy will be the same for all features such as *f*. To emphasize this property, the intermediate classes (*B* and *C* in the original examples) have been renamed *WINNER* and *LOSER* on the last figure. The choice between them is indeed absolute: like the America's cup, this is a race with no second place.

But what if we want to use the *WINNER* version for feature *f*, and for another feature subject to the same problem — *g* on the figure — we want to retain the version redeclared in the other class, *LOSER*?

The reason this hasn't been a major concern until this stage of the discussion is that the case is not common. Most of the time, in repeated inheritance situations of the above type with conflicting redeclarations, one of the parents is indeed the victor, providing all the variants for dynamic binding. (Sometimes it's because its form of inheritance was more for subtyping, and the loser's was more implementation inheritance.)

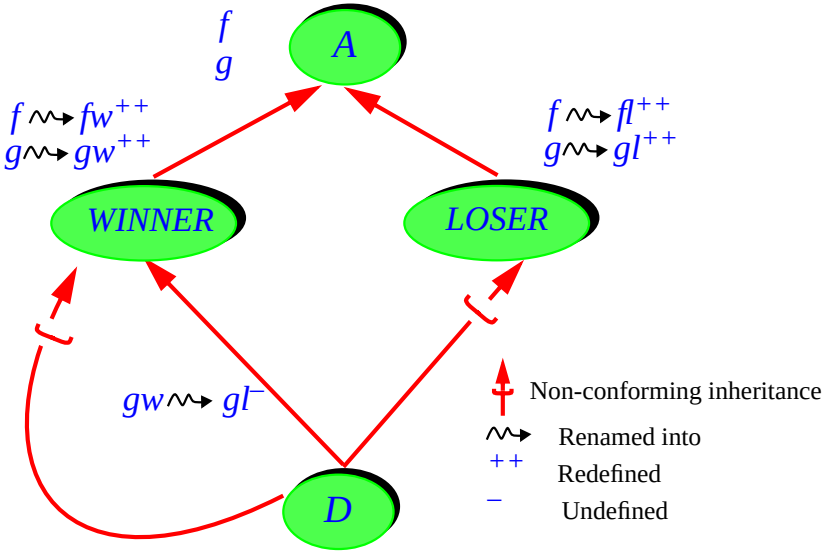
*On various forms of inheritance see the inheritance methodology chapter in Object-Oriented Software Construction, 2nd edition".*

But there will be exceptions to this observation, and we need a way to address them. The idea is simply to rely on the Join mechanism.

First assume that although you want two versions of the original *f* you need only one of *g*, the *LOSER* version. Then a simple join will solve the problem: it suffices to inherit both versions under the same name, and to undefine the one from *WINNER*; the other will take over.

If you want to keep both versions of *g*, but make *gl* the selection for dynamic binding from higher-ups, you will use essentially the same technique but in this case you need to inherit *once more* from *WINNER* (as if mere repeated inheritance from *A* were not already enough), this time in non-conforming form:

***We like it so much we want not just two but three of it!***




This gives *D* another version of *gw*, leaving you free to do whatever you like with the first — the one used for dynamic binding —so that you can let it be overriden by *gl*'s implementation through renaming, undefinition and join (the loser's revenge):

```
class D inherit
        WINNER
                rename
                        gw as gl
                undefine
                        gl
                end
                -- One more time, with feeling:
        expanded WINNER
                -- Not such a total defeat after all:
        expanded LOSER
… Rest of class text omitted …
end
```

You will obtain a similar effect by redefining the *gl* from *LOSER* and the *gl* renamed from *gw* (in the conforming *WINNER* branch) into a common feature. For attributes — which you can't undefine — this is the only possible technique.

## 16.12 THE NEED FOR SELECT

--- EXPLAIN !!!


**Select clauses**

Select ≜ **select** Feature_list

The Select subclause serves to resolve any ambiguities that could arise, in dynamic binding on polymorphic targets declared statically of a repeated ancestor's type, when a feature from that type has two different versions in the repeated descendant.

--- EXPLAIN


**Select Subclause rule** *VMSS*

A Select subclause appearing in the parent part for a class *B* in a class *D* is valid if and only if, for every Feature_name *fname* in its Feature_list, *fname* is the final name in *D* of a feature that has two or more potential versions in D, and *fname* appears only once in the Feature_list.

This rule restricts the use of Select to cases in which it is meaningful: two or more "potential versions", a term which also has its own precise definition. We will encounter next, in the Repeated Inheritance Consistency constraint, the converse requirement that if there is such a conflict a Select *must* be provided.

## 16.13 THE REPEATED INHERITANCE CONSISTENCY CONSTRAINT

Although we have seen all the concepts, it remains to formalize some of the definitions and rules:

- The **versions of a feature** and its **dynamic binding version** in a descendant of its class of origin.
- The **Repeated Inheritance Consistency constraint** — the major constraint on the use of repeated inheritance.
- The precise definition of **inherited features of a class** — needed for the more general notion of "features of a class"

- As a consequence, the precise definition of the **final name set** of a class and the **Feature Name rule**, governing the choice of feature names and avoiding unwanted name clashes.

As noted, this material and the remainder of this chapter are not required on first reading.

The purpose of the Repeated Inheritance Consistency constraint is to make sure (by permitting at most one conforming inheritance path) that for any feature of a class there is at most one *dynamic binding version* in any proper descendant. Before defining "dynamic binding version" we need to know what a "version" is, but here we've essentially done the job already by introducing the notion of "seed":

**Version**

A feature *g* from a class *D* is a **version** of a feature *f* from an ancestor of *D* if *f* and *g* have a seed in common.

The seed of a feature was defined as the original form of the feature in the class where it was first introduced, prior to any redeclarations, renamings or other transformations in proper descendants. A version of *f* is a reincarnation of *f* in a descendant.

← *"Origin, seed", page 305*

The definition of "seed" implies that if *f* is immediate (introduced by its class as a new feature) then the common seed of *f* and *g* mentioned in the above definition of "version" is *f* itself.

When may a feature have more than one version in a proper descendant of its class of origin? The answer was given by the semantic rules of this chapter: Repeated Inheritance and Replication Semantics rules. The following rule brings nothing new, but summarizes the consequences of these previous results.

← *"Repeated Inheritance rule", page 430*; *"ReplicationSemantics rule", page 451*.

**Multiple versions**

A class *D* has *n* **versions** (*n* ≥ 2) of a feature *f* of an ancestor *A* if and only if *n* of its features, all with different final names in *D*, are all versions of *f*.

-- REMOVED CLAUSES:

, and any two among them satisfy any of the following properties:

1 • A redeclaration applied to one has not been applied to the other.

2 • Any of them is an attribute.

3 • They have different signatures.

4 • Any of them calls a feature of *A* having (recursively) two or more versions in *D*.

-------- END REMOVED CLAUSES -- DISCUSSION BELOW IS OBSOLETE

Although this rule doesn't mention repeated inheritance, it can only be understood as a consequence of the rules introduced in this chapter: the only way in which *D* may, as required by the definition, have two or more versions of *f* — meaning, from the definition of "version", two or more features with the same seed — is through the replication mechanism of repeated inheritance.

Case 1 is the most common source of multiple versions: the features have been redeclared in different ways along different inheritance paths, or one has been redeclared and the others haven't.

To cover both of these cases, the rule uses careful phrasing: at least one redeclaration has occurred (along one of the inheritance branches) that applies to one of the features but not to the other. This may mean, for the other, no redeclaration at all, or a different redeclaration.

Case 2 follows from the discussion of what replication means in the special case of attributes. Note that it suffices that one of the features be an attribute; it may have as its seed a function that, along the other branch, was either not redeclared or redeclared as a function.

← *"THE CASE OF ATTRIBUTES", 16.6, page 440*.

Case 3, as stated, sounds very general, but if you reflect about it you will realize that it is only relevant in the other special case of replication: conflicting generic derivations. True, another source of differing signatures would be redefinition; but then the more general case 1 will also apply.

← *"THE CASE OF CONFLICTING GENERIC DERIVATIONS", 16.7, page 442*.

Case 4 follows from the discussion of replication semantics: even if a routine has not been explicitly redeclared, it may have an implicit redefinition as a result of replication under repeated inheritance, if it calls a feature that has been redeclared. This case only applies to routines, since only a routine may call another feature (routine or attribute). Note that the call may be in the Routine_body but it might also be, as in the *COUNTER* example, in a Precondition or Postcondition, as well as in a Rescue clause.

← *"THE SEMANTICS OF REPLICATION", 16.10, page 449*.

For the reader interested in theoretical consistency: clause 4 may appear to risk infinite recursion, since it is possible for a routine *r* to call a routine *s* which also calls *r*. This was the case with the example of a recursive routine Interpreting the definition constructively — as a definition by induction, or a fixpoint — avoids this problem: to determine the set of features with more than one version in *D* we first apply cases 1, 2 and 3, the non-recursive cases, to all relevant features; then we repeatedly apply clause 4 to include any features that call a feature already in our set, stopping at the first iteration that yields nothing new. The process is guaranteed to terminate, since the set of features of *D* (and hence too the transitive closure of the call graph) is finite.

*For an introduction to fixpoints and the theory of recursive definitions see "Introduction to the Theory of Programming Languages".*

Throughout this chapter we have used the Repeated Inheritance Consistency constraint, which removed ambiguities for dynamic binding in the presence of conflicting redeclarations. For all practical purposes the earlier informal statements of the constraint were sufficient, but now we can express it in a completely precise form:


**Repeated Inheritance Consistency constraint** *VMRC*

It is valid for a class *D* to have two or more versions of a feature *f* of a proper ancestor *A* if and only if it satisfies one of the following conditions:

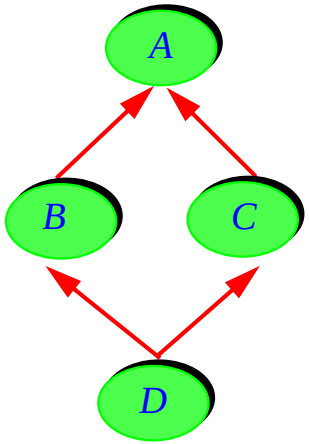
1 • There is at most one conformance path from *D* to *A*.

2 • There are two or more conformance paths, and the Parent clause for exactly one of them in *D* has a Select clause listing the name of the version of *f* from the corresponding parent.


A "conformance path" is a sequence of classes from *D* to *A* such that each of the associated current types conforms to the next. Thanks to the non-conforming inheritance it is possible for *D* to have some inheritance paths to *A* that are not conformance paths.

← *"Conformance path", page 381*; *"Current type", page 357*.

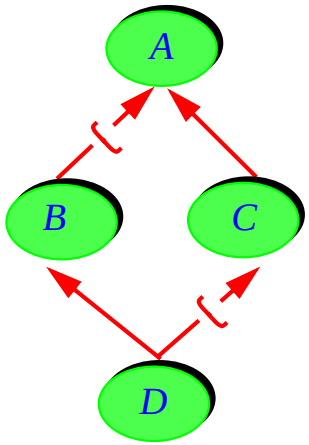

According to this constraint **it is not invalid** for a class to have more than one conformance path to a proper ancestor if no replication causes any ambiguity for dynamic binding. As soon as such a potential ambiguity arises, however, you need to make sure that all inheritance paths, except possibly one, involve at least one non-conforming link.



***More than one path conforms***

Conversely, nothing forces you, in a repeated inheritance situation with or without replication, or in any inheritance situation, to have a conforming path. A class may inherit from another, singly or multiply, without conformance of the associated current types. This is the case of facility or implementation-only inheritance, which does not permit subtyping. It is not the most common use of inheritance, but it is possible:

***No path conforms***

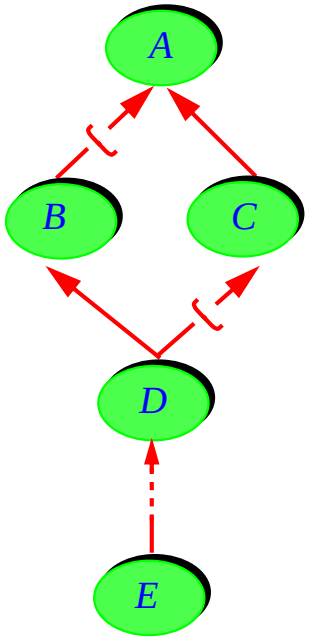



***Suffering from an proper ancestor's repeated inheritance?***

In this case there is no polymorphism: with *a1*: *A* and *d1*: *D*, attachments such as *a1* := *d1* are invalid. (Similarly, with the assumptions of the figure, *a1* := *b1* and *c1* := *d1* with *b1*: *B* and *c1*: *C*.)

A final comment on the Repeated Inheritance Consistency constraint — important in particular for compiler writers — is that the rule as stated might seem to require, for any feature *f* of a class *A*, verification in **every** proper descendant *E* of *A*, at least every *E* such that repeated inheritance with replication occurs somewhere between *A* and *E*, even if the culprit is not *E* but an intermediate descendant *D*:

You don't have to worry about what happens in *E*, however: thanks to the definition of "version", if possible dynamic binding ambiguities arises for *E*, that can only be (if the only cases of repeated inheritance are those appearing on the figure) because they arise for *D*; once you resolve them for *D* in accordance with the Repeated Inheritance Consistency constraint, that will take care of *E* as well.




Thanks to the constraint we can now define *the* dynamic binding *version* (note the singular) of a feature in any descendant of its class of origin:

DEFINITION

**Dynamic binding version**

For any feature *f* of a type *T* and any type *U* conforming to *T*, the **dynamic binding version** of *f* in *U* is the feature *g* of *U* defined as follows:

1 • If *f* has only one version in *U*, then *g* is that feature.

2 • If *f* has two or more versions in *U*, then the Repeated Inheritance Consistency constraint ensures that either exactly one conformance path exists from *U* to *T*, in which case *g* is the version of *f* in *U* obtained along that path, or that a Select subclause name a version of *f*, in which case *g* is that version.

As you will have noted:

- The definition has moved on from classes to types, since this is what matters for feature calls and dynamic binding. All the concepts transpose immediately; in particular, "features of a type" was defined precisely in an earlier chapter.

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

- If *T* and *U* are the same type, case 1 applies; so the definition indicates — as it should — that *f* is its own dynamic version.

The definition enables us to obtain a **single** dynamic binding version for every inherited feature. This is of course the very purpose of the entire present discussion, and the reason for the Repeated Inheritance Consistency constraint.

The result is at the very heart of the object-oriented machinery of Eiffel: when discussing the fundamental computational mechanism, feature call, we will specify that a call $a.f(\ldots)$ triggers the **dynamic binding version of** $f$ in the type of the object dynamically attached to $a$. Thanks to the preceding rules and definitions, we now have the guarantee that this notion will always be unambiguously defined, even under the most sophisticated forms of multiple and repeated inheritance.

## 16.14 THE INHERITED FEATURES OF A CLASS

(Like the previous one, you may skip this last section on first reading.)

The final prize we earn from all the work done in this chapter is the ability to provide a precise, conclusive definition of a key notion: the features of a class — in particular its inherited features.

As specified in the original discussion of features, the "*features of a class*" include its immediate features (those introduced in the class itself), and its inherited features, which were defined informally as the features "*obtained from*" the parents' features.

← *Chapter 5; see "IMMEDIATE AND INHERITED FEATURES", 5.4, page 133.*

The reason for being informal at that earlier stage is now clear: two mechanisms, repeated inheritance and join, affect how a class may "obtain" features from its parents. Without these mechanisms, every feature from a parent (every **precursor**) would yield one feature in the heir. But:

- The **join** mechanism merges two or more features from parents into a single one in their common heir.
- With **sharing** under repeated inheritance, two or more precursors, inherited from different parents but coming from the same features of a common ancestor, yield a single feature of $D$.
- Conversely, with **replication** under direct repeated inheritance ($D$ has two or more Parent clauses listing the same parent), a single precursor may yield two or more features of $D$.

Only with the benefit of these observations can we now obtain a precise definition of the "inherited features of a class", and hence (since immediate features — the new, non-inherited ones — raise no particular problem) of the **features of a class**. Here is the full definition:


### Inherited features

Let *D* be a class. Let *precursors* be the list obtained by concatenating the lists of features of every parent of *D*; this list may contain duplicates in the case of repeated inheritance. The list *inherited* of **inherited features** of *D* is obtained from *precursors* as follows:

1 • In the list *precursors*, for any set of two or more elements representing features that are repeatedly inherited in *D* under the same name, so that the Repeated Inheritance rule yields sharing, keep only one of these elements. The Repeated Inheritance Consistency constraint (sharing case) indicates that these elements must all represent the same feature, so that it does not matter which one is kept.

2 • For every feature *f* in the resulting list, if *D* undefines *f*, replace *f* by a deferred feature with the same signature, specification and header comment.

3 • In the resulting list, for any set of deferred features with the same final name in *D*, keep only one of these features, with assertions and header comment joined as per the Join Semantics rule. (Keep the signature, which the Join rule requires to be the same for all the features involved.)

← *"Join Semantics rule", page 312*.

4 • In the resulting list, remove any deferred feature such that the list contains an effective feature with the same final name. (This is the case in which a feature *f*, inherited as effective, effects one or more deferred features: of the whole group, only *f* remains.)

5 • All the features of the resulting list have different names; they are the inherited features of *D* in their parent forms. From this list, produce a new one by replacing any feature that *D* redeclares (through redefinition or effecting) with the result of the redeclaration, and retaining any other feature as it is.

6 • The result is the list *inherited* of inherited features of *D*.

This definition looks a little like an algorithm, but it's not; you may view it as a plain mathematical specification. There is no requirement that compilers implement the corresponding mechanisms by mimicking the rule's successive steps, as long as the result is compatible.

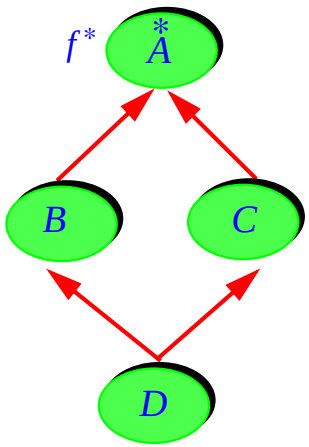
The order of the clauses is significant. Note in particular that the very first step, clause 1, takes care once and for all of repeated inheritance. This removes a small potential ambiguity, which we may remove through a semantic rule (not a new property, just a consequence of the preceding definition):


**Join-Sharing Reconciliation rule**

If a class inherits two or more features satisfying both the conditions of sharing under the Repeated Inheritance rule and those of the Join rule, the applicable semantics is the Repeated Inheritance rule.

The situation is illustrated by the figure below: *f* is deferred at the level of *A*, and nothing else — renaming, effecting … — happens to it down to the level of *D*. It's a case of sharing under repeated inheritance, but we might also apply the Join semantics, as always when a class inherits under a single name a set of features, all deferred (or, although this doesn't apply here, all deferred except one). You may have wondered about this case: which of the two semantic rules should we apply? You may also have brushed off the question: does it matter at all?

***Join, or sharing?***




* Deferred

It matters not much, but it matters just a little and we must leave no semantic stone unturned. The only difference has to do with assertions. Assume that *f*, deferred as it may be, has a postcondition

```
ensure
        your_condition
```

Then the Join Semantics rule prescribes combining the header comments of the joined features, and also their assertions: through an **or** for the preconditions, and an **and** for postcondition. Because *a* **and** *a* has the same value as *a*, no really bad semantic consequence will follow, but for example a class documentation tool, such as a flat-short form displayer, might mistakenly display the postcondition of *f* in *D* as something like:

←*Clauses 3 and 4 of the "Join Semantics rule", page 312.*

← *"Contract view, flat-short form", page 211*

```
ensure
                -- From A:
        your_condition
        and
                -- From A:
        your_condition
```

Not a disaster, but unnecessarily complex. The Join-Sharing Reconciliation rule explicitly defines the resulting postcondition in such a case to be just *your_condition*, with a similar consequence for preconditions and header comments.


Let's come back to more general properties of the definition of Inherited Features. To understand the definition, note that the lists under consideration are lists of **features**, not of feature names, although the features that remain at the end all have different final names in *D*. The list *inherited* obtained at step 6 of the definition may still contain duplicate features — with different feature names — as a result of repeated inheritance with replication. This is why we define *precursors* as a list rather than a set. (Unlike a set, a list may contain duplicates.)

In fact these observations also yield a new definition of the "precursors" of a feature, equivalent to the original one but more precise:

← *"Precursor (joined features)", page 309. See also the first, simplified definition on page 262.*

DEFINITION

**Precursor**

A **precursor** of an inherited feature of final name *fname* is any parent feature — appearing in the list *precursors* obtained through case 1 of the definition of "Inherited features" — that the feature mergings resulting from the subsequent cases reduce into a feature of name *fname*.

In accordance with this definition the successive steps of the definition of "inherited features" may only merge features — elements of the list *precusors* — if they all have the same final name. This is an important property because without it the earlier definition of the final name of an inherited feature would not make sense.

← *"Final name, extended final name, final name set", page 183.*

Recall that according to this definition the final name *m* of a feature *f* obtained from a precursor of name *n* in a parent *B* is:

- *n* in the absence of renaming.
- Otherwise, the *m* appearing in a Rename_pair of the form **rename** *n* **as** *m* in the Parent clause for *B* in *D*.

Obviously, if *f* is obtained from two or more precursors, all this is meaningless unless we are sure that *m* is the same for all these precursors.

This also clarifies the notion of **final name set** of a class, originally introduced — in the same definition as "final name" — as the set of final names of all the features of a class. These final names are:

← *"Final name, extended final name, final name set", page 183.*

- For immediate features, the names under which the class declares them.
- For inherited features, the inherited names except as overridden by renaming.

Two or more precursors merged into one — because of either a join or sharing under repeated inheritance — yield just one element of the final name set. If a feature from a repeated ancestor yields several features under replication, this adds all the corresponding names to the final name set.

*Both the Repeated Inheritance rule and the Join rule require all the merged features to have the same final name.*

Finally, we introduce a simple constraint capturing the fundamental rule on choosing feature names:


**Feature Name rule** *VMFN*

It is valid for a feature *f* of a class *C* to have a certain final name if and only if it satisfies the following conditions:

1 • No other feature of *C* has that same feature name.

2 • If *f* is shared under repeated inheritance, its precursors all have either no Alias or the same alias.

Condition 1 follows from other rules: the Feature Declaration rule, the Redeclaration rule and the rules on repeated inheritance. It is convenient to state it as a separate condition, as it can help produce clear error messages in some cases of violation.

Two feature names are "the same" if the lower-case version of their identifiers is the same.

← *"Same feature name, same operator, same alias", page 153*.


The important notion in this condition is "**other feature**", resulting from the above definition of "inherited features". When do we consider *g* to be a feature "other" than *f*? This is the case whenever *g* has been declared or redeclared distinctly from *f*, unless the definition of inherited features causes the features to be merged into just one feature of *C*. Such merging may only happen as a result of sharing features under repeated inheritance, or of joining deferred features.

← *"Inherited features", page 462*.

Also, remember that if *C* redeclares an inherited feature (possibly resulting from the joining of two or more), this does not introduce any new ("other") feature. This was explicitly stated by the definition of "introducing" a feature.

← *"Inherited, immediate; origin; redeclaration; introduce", page 133*

Condition 2 complements these requirements by ensuring that sharing doesn't inadvertently give a feature more than one alias.

The Feature Name rule crowns the discussion of inheritance and feature adaptation by unequivocally implementing the No Overloading Principle: no two features of a class may have the same name. The only permissible case is when the name clash is apparent only, but in reality the features involved are all the same feature under different guises, resulting from a join or from sharing under repeated inheritance.

Consequences of the Feature Name rule includes the following properties, which for convenience we may group into a new constraint:


**Name Clash rule** *VMNC*

The following properties govern the names of the features of a class *C*:

1 • It is invalid for *C* to introduce two different features with the same name.

2 • If *C* introduces a feature with the same name as a feature it inherits as effective, it must rename the inherited feature.

3 • If *C* inherits two features as effective from different parents and they have the same name, the class must also (except under sharing for repeated inheritance) remove the name clash through renaming.

*WARNING: not a validity constraint in the usual form; see comment at bottom of preceding page.*


This is not a new constraint but a set of properties that follow from the Feature Name rule and other rules. Instead of Eiffel's customary "This is valid if and only if …" style, more directly useful to the programmer since it doesn't just tell us how to mess things up but also how to produce guaranteeably *valid* software, the Name Clash rule is of the more discouraging form "You may not validly write …". It does, however, highlight frequently applicable consequences of the naming policy, and compilers may take advantage of it to report naming errors.