Adapting Inherited Features in Eiffel

Roger F. Osmond, 16 February 2000

(updated 15 May 2018)

Introduction

Inheritance is the mechanism in Object Oriented languages by which one can acquire in one class the attributes and capabilities of another class (a descendent or child class *inherits* from an ancestor or parent class). There are various forms of inheritance, as well as various purposes to which inheritance is applied.

Single inheritance occurs when a child class inherits a single parent class.

Multiple inheritance occurs when a child class inherits more than one parent class.

Repeated inheritance occurs when a child class inherits the same ancestor class more than once, either directly or indirectly.

Pretty much all Object-Oriented languages support at least single inheritance. Eiffel supports multiple and repeated inheritance. Included in Eiffel's inheritance support is the ability to *adapt* inherited features via redefinition, renaming, un-definition, and selection. This paper explains these mechanisms and their typical use.

Basics of Inheritance

The Purpose of Inheritance

Dr. Bertrand Meyer defines in his epic "*Object Oriented Software Construction, 2nd Edition*", eleven different forms of, or reasons for inheritance. I list just a few obvious generalizations.

- To acquire specific attributes or capabilities
- To specialize certain attributes and capabilities
- To share common attributes and capabilities

Inheritance is the means by which we can develop a model of the problems we encounter, and the solutions we envision, and ultimately provide high quality solutions. If our problems are typical, our models require fairly rich inheritance relationships. Eiffel has, from the very beginning, supported multiple inheritance, presumably because nothing less would adequately support the models we need to do our work.

Single versus Multiple Inheritance

Some Object-Oriented languages, notably Smalltalk and Java, support only single inheritance. Even C++ supported only single inheritance until version 2.0 of the language (and only after considerable public pressure).

Many champions of these languages will tell you that single inheritance is good because multiple inheritance is bad. This is an interesting application of logic, for it begins with a false assertion. How can multiple inheritance, conceptually at least, be "bad"?

Imagine if your design called for modeling human genetic traits. You would define a person class and each person would have two parents, and zero or more siblings, each of which is a person too. This is fine so far, but where do the genetic traits enter the picture? They enter via *familial* inheritance; a person acquires certain attributes from each parent. Depending on the relative dominance and resulting probability, the child's attributes reflect these acquisitions.

The temptation is to create our model based on a literal transcription of the vocabulary. It seems a reasonable approach, at first.

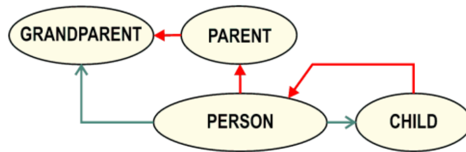


Figure 1 – Intuitive Model

A person inherits from its parent, who in turn inherits from its parent (the grandparent of the first person).

There a few problems with this model. Most critically, the model attempts to represent *instances* of classes as classes. From there, it never gets back on track.

A more reasonable model identifies PERSON as a class, and then has the various subtypes of PERSON be new classes, but children of PERSON.

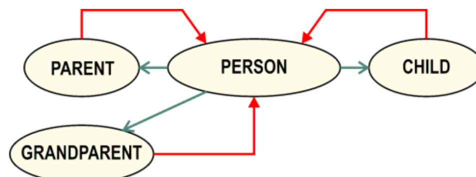


Figure 2 – Somewhat More Realistic Model

The model depicted in Figure 2 differs quite a bit from the previous one. Rather than PERSON inheriting from PARENT, PARENT is a *supplier* to PERSON (i.e. PERSON has-a PARENT). A similar change happens to the PERSON-to-CHILD relationship. Each of the classes CHILD, PARENT, and GRANDPARENT inherits from (i.e. is a kind of) PERSON. The personal subclasses exist in this model because they matter to the model, in that form. The model is one the represents familial relationships, and it could be expanded, as needed, to depict siblings, cousins, great-grandparents, spouses, domestic partners, ex-spouses, even the guy who fixes your car, but each of those entities would be a child of PERSON.

The model would not necessarily need to include each of the relationships, as each is derivable from a few basic ones, and a few less obvious ones.

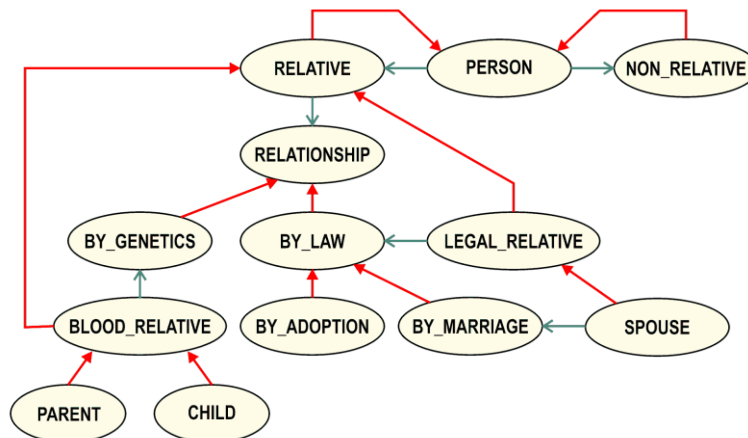


Figure 3 – Person Model - Expanded

The model in Figure 3 shows that PERSON can have a supplier relationship with other kinds of (subclasses of) PERSON. At the highest level, there are 2 major subclasses of PERSON: RELATIVE and NON_RELATIVE.

RELATIVE has a supplier relationship with (*has-a*), you guessed it, RELATIONSHIP. Subclasses of RELATIONSHIP include adverb-like abstract classes BY_GENETICS and BY_LAW.

LEGAL_RELATIVE is subclass of RELATIVE, whose relationship is BY_LAW.

BLOOD_RELATIVE is a subclass of RELATIVE, whose relationship is BY_GENETICS.

PARENT and CHILD each are subclasses of BLOOD_REALATIVE.

BY_MARRIAGE and BY_ADOPTION are subclasses of BY_LAW, and SPOUSE is a LEGAL_RELATIVE, whose relationship is BY_MARRIAGE.

Grandparents are parents of parents. Cousins are children of the children of grandparents and so on, and could be implemented using the model depicted in Figure 3.

But we haven't quite put to bed the whole lineage thing. What about representing a family tree? Never mind the gene selection, how about something simple, like national origin?

Assume a person has two parents, one Norwegian and the other English. From a modeling standpoint, that person is Norwegian and is English.

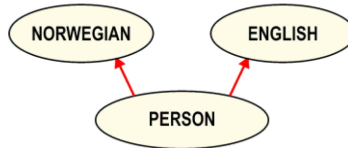


Figure 4 – Person with Multiple National Origins

An argument can be made, and often is, that this is not inheritance at all, that the nationality aspects can be acquired from a supplier. That argument can be applied to every inheritance relationship, if the goal is to eliminate inheritance as a relationship altogether. But, is this person not both Norwegian and English? If you were to ask that person, then they would surely say that they were. Why can't a model reflect what that person believes to be true? Isn't that the point? The model can do this, but not without multiple inheritance.

Multiple inheritance is necessary to model this problem accurately. It also stands to reason that, if multiple inheritance is necessary in the model, then it should be supported by the language and any associated tools used to implement the model.

> *The conventions used in the illustrations are adapted from the BON notation. Simply, single red closed-end arrows denote inheritance relationships, where the source of the arrow is the child (the* child inherits *from the parent). A client supplier relationship appears as a single green open-ended arrow, where the* client uses *the supplier.*
>
> *The arrow always originates at the class from which the relationship, not the attribute, originates. This is* not *a data flow diagram.*

Let's look at another example, the classic automotive taxonomy. Your car is, in the example, a blue 1995 Ford SHO 4-door sports sedan with a 5-speed manual transmission, a 220 HP engine, moon roof, climate control and so on. Your mission is to model the automotive world in such a way that you can classify your car, as well as any other vehicle, in a reasonable and efficient manner.

Begin with the description of your car. Which of its characteristics are likely inherited from a parent class? Thus begins the object-oriented analysis and modeling exercise.

If we limit our modeling to single inheritance, then we might derive a simple hierarchy like this one.

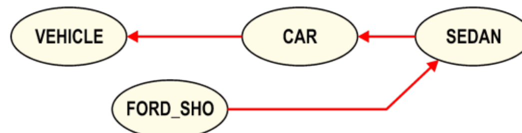


Figure 5 – A Simple Car Hierarchy

On closer analysis, we find such characteristics as "wheeled" or "four-wheeled" and "powered" or "gasoline-powered". One can argue, and many do, that these characteristics can be modeled as simple attributes, without inheritance. This is undeniably true, as proven by the existence of software that deals

with these characteristics, and is wholly non-object-oriented, or even *object-disoriented*. Remember though, that the goal is to derive a *reasonable and efficient* model.

Just as Scott and Amundsen each believed that he could reach the South Pole by his preferred method, only Amundsen's method was reasonable and efficient. While both approaches seemed to work on a smaller scale ("in the lab"), only Amundsen achieved his goal and lived to tell about it.

Here is a possible class taxonomy resulting from more insightful analysis and multiple inheritance.

The model is incomplete and obviously contrived, but it serves a purpose.

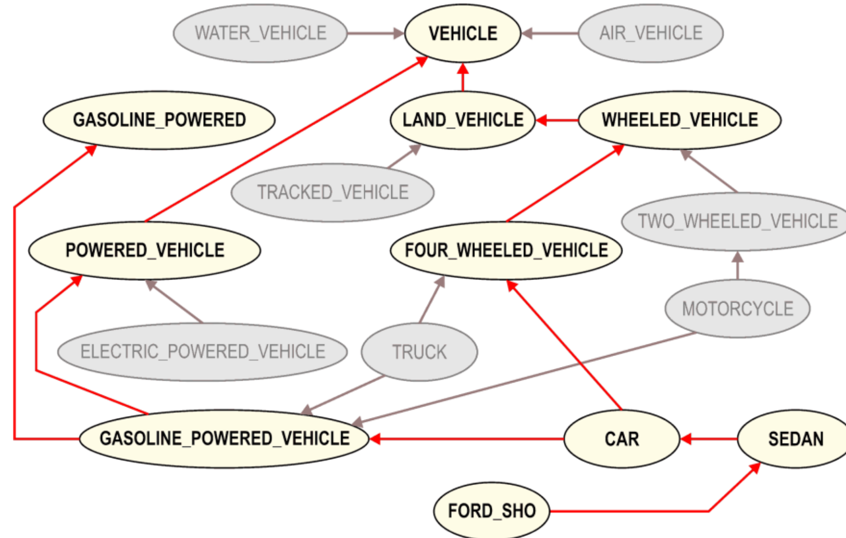


Figure 6 – Modeling a 1995 Ford SHO

So, if it is true that one can model the problem without inheritance, and can do so with single inheritance, then why use multiple inheritance? The same argument could be made for using a screwdriver for wood carving. You can do it, but you can't do it very well and you certainly cannot produce any respectable work product with it. The tool just does not fit the problem well.

I can sing a little, but I am no Caruso, so as long as the singing problems I try to solve are limited to the scale of the shower, then I am fine. As soon as the scale of the singing problem exceeds that, I am in hot water, so to speak.

We use multiple inheritance because it is more *reasonable* and *efficient* (in a broad view) and because the resulting model, in the form of classes, has components that are less complex, and with less duplication of logic. We achieve greater *genericity*, and more options for specificity. We use multiple inheritance because we can achieve a far greater degree of reuse and reusability.

Greater reuse results (eventually) in software that is less brittle and requires less effort to maintain, and has fewer errors. The lifecycle cost of the software is lower. We are able to address larger problems. In other words, by using multiple inheritance, we not only reach our objective, but we will also live to tell about it.

The truth is simply that multiple inheritance is needed in the model if that model is to reflect well the problem. It is also true that modeling with multiple inheritance is easier than with single inheritance because you don't have to *adapt the problem to fit the tool*.

Early users of C++ (I among them) realized that single inheritance was inadequate. There was a lot of pressure from the user community to have C++ support multiple inheritance.

This is not to say that supporting multiple inheritance in a language system is easy, or that it has always been done properly (it is very difficult – that's one reason why so few languages do it). The problem is

not that multiple inheritance is bad, it is that multiple inheritance is difficult for a compiler writer to support.

Recall that the early versions of C++ were implemented as a preprocessor, cfront, *not a compiler – multiple inheritance would have been especially tough to support correctly then. I know, because I have written preprocessor-based front-end languages.*

That is why multiple inheritance is unsupported or poorly supported in most languages that claim to be Object Oriented. It is simply too difficult a job for the compiler writers. Retrofitting multiple inheritance to a language that was not designed for it is close to impossible because multiple inheritance affects nearly every aspect of a language. So, instead of providing us with the tools that we need, they tell us to use the tools that they have, and try to convince the world that their shortcomings are actually desirable features (sound familiar?). To me, they are making a virtue of necessity (a more charitable interpretation than usual) or simply whistling past the graveyard.

Eiffel is different. Eiffel supports, encourages, and embraces multiple inheritance. Eiffel's creator knew from the outset that simplifying his job (by implementing only single inheritance) would result in a tool that did not satisfy the requirements of its users. He realized that one cannot adequately model relationships that include multiple inheritance with a tool that doesn't support it fully. This doesn't sound like a difficult conclusion, but it must have been, else the creators of the other languages (some many years newer) would have seen it – assuming of course that they were so motivated.

Repeated Inheritance

One of the aspects of multiple inheritance often cited as a danger, and sometimes as a reason to avoid multiple inheritance altogether, is the phenomenon of repeated inheritance. To experienced Eiffel programmers, this can be amusing, as they tend to exploit repeated inheritance to advantage. Others less experienced in this area might be forgiven their trepidation.

Repeated inheritance can be explicit (as is seen often in the Eiffel standard libraries) or implicit. Explicit repeated inheritance occurs when the designer decides to inherit the same class more than once in a given descendent.

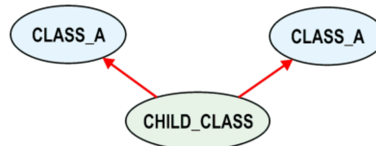


Figure 7 – Explicit / Direct Repeated Inheritance

Implicit repeated inheritance occurs when the designer inherits classes that in turn inherit a common ancestor. In practice this occurs quite often. The following diagram illustrates this relationship.

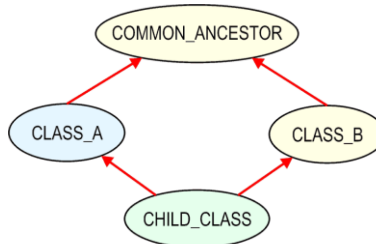
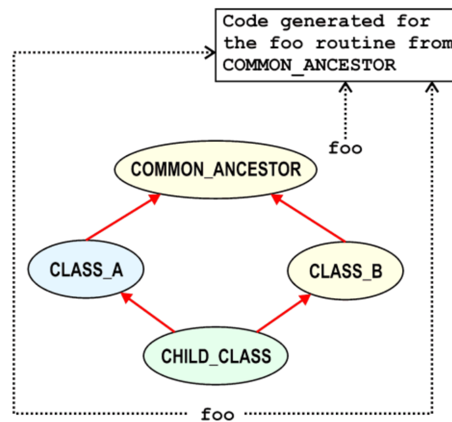


Figure 8 – Implicit / Indirect Repeated Inheritance

In Figure 8, CHILD_CLASS inherits from CLASS_A and from CLASS_B. Each of these parents inherits from COMMON_ANCESTOR, resulting in implicit repeated inheritance. This condition is handled correctly and completely by Eiffel. To understand the problems that arise from incomplete or incorrect handling of this condition, it is necessary to look at the mechanics of inheritance.

Originally, C++ was a preprocessor that allowed the definition of classes and objects using C language compilers (a worthwhile and not uncommon cause at the time). The mechanism begins as a conceptual cut-and-paste, the same as the **#include** mechanism of the C preprocessor (*cpp*). The designated file's content is pasted into the file (in memory) that includes it. This is a powerful mechanism and one that has enjoyed widespread acceptance for decades. It is not, however, adequate to support multiple inheritance.

Imagine now that the class COMMON_ANCESTOR has a routine in it, called (per tradition) "**foo()**". This routine is not modified in any way in either CLASS_A or CLASS_B. According to our implicit repeated inheritance, our CHILD_CLASS might have two copies of **foo()**, unless the compiler does something about it.



In the crudest of implementations of C++, this apparent conflict is left to the linker to resolve. Pretty much any resolution in this case would work (assuming reentrant routines of course) because there is no difference in the *implementations* of the two different *instances* of the **foo()** routine.

The Eiffel compiler recognizes that there exists this repeated inheritance condition, then verifies that the two versions are indeed identical, and if so, the compiler generates the code for only one instance.

If the compiler sees that there are different implementations, one having been redefined or renamed along the way, or simply a different routine with the same name, then the compiler tells the user that there is a conflict. The conflict must be corrected before the system will compile.

Not so Basic Inheritance

In the first part of this document, we covered the basics of inheritance. In this part, we look at some of the less obvious aspects of inheritance, along with some of the potential benefit of using inheritance.

Compile-Time versus Run-Time

Much of the confusion felt by programmers first exposed to Object-Orientation centers around the concept of binding. Binding is the process of associating a type with a language element.

In Eiffel, one can define functions, attributes, arguments, and local instances that have types. Each is represented by a named *handle*.

The handle is the means by which we access a feature. It is the type of the *handle* that the compiler validates. Here are a few examples (line numbers are for illustration only).

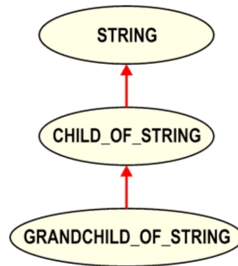
```
 1 my_name: STRING
 2
 3 length_of_my_name: INTEGER
 4  do
 5    Result := my_name.count
 6  end
 7
 8 do_something
 9  local
10    tstr: STRING
11  do
12    my_name := "Bozo"
13    tstr := my_name
14    tstr := length_of_my_name -- This is wrong
15  end
```

Line 1 contains the declaration of an *attribute* whose type is STRING and whose handle is **my_name**. Line 3 begins the definition of a *function* (a routine that returns a typed value) whose type is INTEGER and whose handle is **length_of_my_name**. The type of the function is the type of the keyword **Result**. The compiler verifies on line 5 that the assignment of the value of **my_name.count** to **Result** is type-consistent; that is, that the *handles* on both sides of the assignment statement are type-consistent (STRING's **count** feature is an INTEGER).

Line 8 begins the definition of the procedure (a routine that does not return a value) **do_something**. This procedure has a local entity whose handle is **tstr** and whose type is STRING. On line 12, the compiler must verify that the value on the right side (the STRING constant "Bozo") is type-consistent with the handle on the left side (the attribute **my_name**). The same sort of test occurs on lines 13 and 14. The assignments on lines 12 and 13 are correct, but the statement on line 14 is wrong because the type of **length_of_my_name** is not consistent with the type of the local entity **tstr**.

All of this verification is done at compile time.

Now let us look at a more complex example. Imagine a simple class hierarchy in which there are two descendants of STRING.



Each of the two descendent classes, CHILD_OF_STRING and GRANDCHILD_OF_STRING has features unique to them (that they did not inherit.) Here are their class definitions.

```
class CHILD_OF_STRING
inherit
  STRING
create
  make
feature
 child_feature: INTEGER
end -- class CHILD_OF_STRING
```

```
class GRANDCHILD_OF_STRING
inherit
  CHILD_OF_STRING
create
  make
feature
 grandchild_feature: INTEGER
end -- class GRANDCHILD_OF_STRING
```

Now we use these classes in an example routine (again, line numbers are for illustration only).

```
 1 my_routine
 2  local
 3    t1: STRING
 4    t2: CHILD_OF_STRING
 5    t3: GRANDCHILD_OF_STRING
 6  do
 7    create t3.make(0)
 8    t3.append ("some string")
 9    t1 := t3
10    t2 := t3
11    -- Now we run into trouble
12    t3 := t2
13    create t3.make (0)
14    t3.append ("another string")
15  end
```

Each local entity in `my_routine` has a typed handle, "`t1`", "`t2`", and "`t3`". In the body of the routine, these handles appear in a series of statements. Lines 8, 9 and 10 pass the compiler's type consistency test without a problem. This is because the object created and referred to by the handle `t3` is type consistent with (being a proper descendent of) the classes to which the other handles belong. The type of `t3` is GRANDCHILD_OF_STRING, and so `t3` is *at least* a CHILD_OF_STRING (line 10) and *at least* a STRING (lines 8 and 9).

Everything is fine until the assignment on line 12 causes a compiler error.

```
VJAR: Source of assignment is not compatible with target.
    Error code: VJAR

    Type error: source of assignment is not compatible with target.
    What to do: make sure that type of source (right-hand side)
    is compatible with type of target.

    Class: APPLICATION
    Feature: my_routine
    Target name: t3
    Target type: detachable GRANDCHILD_OF_STRING
    Source type: CHILD_OF_STRING
...
```

How can this be? On line 10 we did a "**t2 equals t3**" assignment. No, we certainly did not. We did a "**t2 gets t3**", or if you prefer (and a little more precise), a "**t2 refers to t3**".

We reset the *handle* `t2` to refer to the *entity to which* `t3` *refers*. There is no *equality* test here. Unlike other languages that seem to encourage bug-by-punctuation, Eiffel does not. Assignments and equalities are clearly different and so are less easily confused.

The compiler must work within the limits of the compile-time environment. The right side *handle* in the assignment on line 12 (`t2`) is of type CHILD_OF_STRING and the left side *handle* in the assignment (`t3`) is of type GRANDCHILD_OF_STRING. This is a *type-inconsistent* assignment (at compile time) because the compiler can't be certain that the entity to which `t2` refers is an instance of GRANDCHILD_OF_STRING. It can be certain only that `t2` refers to an instance of CHILD_OF_STRING (or **Void**).

Even though at run-time *we* know that the object to which `t2` refers would be a GRANDCHILD_OF_STRING, this is not consistent with the semantics at compile time, so the compiler must flag the assignment as an error because the handles involved are not type-consistent.

At run-time, we can see that the behavior is such that the assignment *would have* worked. The following series of diagrams illustrates the behavior of the generated code at run-time.

After line 8, the handle `t3` refers to the GRANDCHILD_OF_STRING object whose value is "some_string".

After line 9, the handle `t1` refers to the same object as did the handle `t3`.

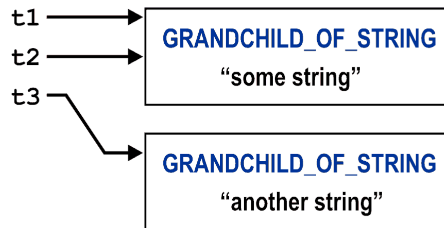
After line 10, the handle `t2` also refers to that same object.

We can make this assignment compile by deferring binding (and the checking it needs) until runtime using the attachment test (crammed onto one line to preserve numbering).

```
12 if attached {GRANDCHILD_OF_STRING} t2 as tgc then t3 := tgc end
```

This is correct both at compile-time and run-time. The Eiffel run-time system performs the type check at run-time and will not execute the assignment if there is still a type conflict at that time, just as it will not execute any other statement within a conditional block unless the condition is True.

On lines 13 and 14, we retarget the handle `t3` to refer to a new GRANDCHILD_OF_STRING object, having a string value of "another string". This has no effect on the remaining handles; they still refer to the same object as before, as illustrated in the following diagram.



Inheritance and Polymorphism

Polymorphism is often either misunderstood or only partially understood. The word means literally "having many forms". In the simplest sense, polymorphism means that a child looks like its parent. There is more to it than that of course. Polymorphism means that one can view an object in different contexts as belonging to any class in its proper lineage.

In our vehicle example from earlier in this paper, we would view an instance of the class FORD_SHO as an object of that class. We could also view it as a GASOLINE_POWERED, or as a SEDAN, or a CAR, or as a VEHICLE, depending on our need at that point.

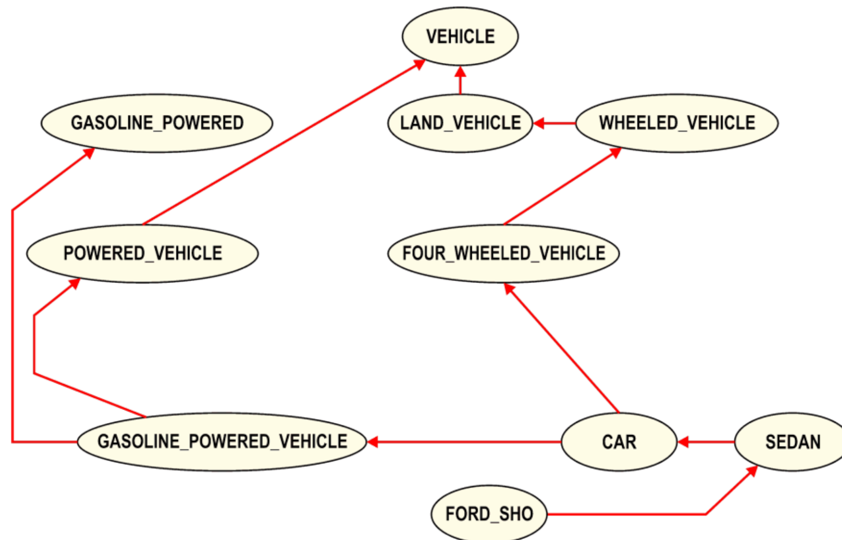


Figure 9 – Ford SHO Model Revisited

The following code segment illustrates the different views just described.

```
some_routine
 local
   a_vehicle: VEHICLE
   a_sedan: SEDAN
   a_car: CAR
   an_environment_killer: GASOLINE_POWERED
 do
   a_vehicle := my_sho
   a_sedan := my_sho
   a_car := my_sho
   an_environment_killer := my_sho
   my_fleet.extend (my_sho)
   a_car := my_fleet.first
   an_environment_killer := my_fleet.last
   a_vehicle := a_sedan
   a_vehicle := a_car
 end
my_sho: FORD_SHO
my_fleet: LINKED_LIST [CAR]
```

Genericity; Constrained or Otherwise

Most core Eiffel container data structures are quite generic. They are not at all fussy about the classes of objects they can contain. The programmer can create, for example LINKED_LISTs containing STRINGs, or INTEGERs, or some programmer-defined classes. The type of the items contained is pretty much up to the programmer to decide. So, if the programmer wishes to create a list of strings, then he or she need only declare it as such, as in:

```
 a_string_list: LINKED_LIST [STRING]
```

Similarly, the programmer can decide that a two-way sorted list is better suited to the task at hand.

```
a_string_list: SORTED_TWO_WAY_LIST [STRING]
```

How does Eiffel help to support this? Among other things, Eiffel supports genericity, as well as the concept of *constrained genericity*. The LINKED_LIST class, for example, has the following declaration.

```
class LINKED_LIST [G]
inherit
 DYNAMIC_LIST [G]
  redefine
    go_i_th, put_left, move, wipe_out, isfirst, islast, first, last,
    finish, merge_left, merge_right, readable, start, before, after, off
  end
```

The first line says that LINKED_LIST is a logical container (as evidenced by the '[' ']' pair) whose elements can be of any type. This is *unconstrained genericity*. The 'G' is merely a placeholder for a type to be chosen at some later point, by the programmer (as with **a_string_list**, above.) This is called a *formal generic parameter*. This formal generic parameter appears elsewhere in the class to anchor the as-yet-undetermined type.

Remember that this type determination, or lack of it, is in space, not in time. This is a programmer's realm, not that of the run-time system.

LINKED_LIST has a feature called **item** that represents the current item (a linked list is a cursor-based structure; thus **item** refers the object at the cursor, it being represented by the feature **active**.)

```
item: G
  -- Current item
  do
    Result := active.item
  end
```

Now let's look at the SORTED_TWO_WAY_LIST. Here is its class declaration.

```
class SORTED_TWO_WAY_LIST [G -> COMPARABLE]
inherit
  TWO_WAY_LIST [G]
    undefine
      has, search
    redefine
      prune_all, extend, new_chain
  end
  SORTED_LIST [G]
    undefine
      move, remove, before, go_i_th, isfirst, start, finish,
      readable, islast, first, prune, after, last, off, prune_all
  end
```

The first line of SORTED_TWO_WAY_LIST differs from the one in LINKED_LIST. This is because the SORTED_TWO_WAY_LIST class uses *constrained genericity* to require of its contained objects that they are type-consistent with (belong to a proper descendent of) the class COMPARABLE.

It is by the features of COMPARABLE that the sorted list maintains its sort order. The list doesn't know about the implementation of the objects it contains, but it does know that they must be proper descendants of COMPARABLE. Each element of a SORTED_TWO_WAY_LIST, then must have the feature **is_less** (aliased as "<").

The data structure asks each of its elements whether they are less than another to determine sort order. In this way, constrained genericity allows the data structure to remain as independent of its contained objects as possible ("minding its own business"), maximizing both flexibility and reusability. This is polymorphism at work.

The following code segment declares two sorted lists, one containing STRINGs and one containing INTEGERs. Because STRING and INTEGER are proper descendants of COMPARABLE, these declarations are correct.

```
 my_alphabetical_list: SORTED_TWO_WAY_LIST [STRING]
 my_number_list: SORTED_TWO_WAY_LIST [INTEGER]
```

If we were to define a class that does not inherit COMPARABLE, then we could not legally declare a sorted list of these objects.

```
class COMP_TEST

feature
 value: INTEGER

end
```

```
my_test_list: SORTED_TWO_WAY_LIST [COMP_TEST]
```

This would not compile because the constrained genericity defined in SORTED_TWO_WAY_LIST is violated by the declaration.

```
VTCG: Actual generic parameter does not conform to constraint
    Error code: VTCG

    Error: actual generic parameter does not conform to constraint.
    What to do: make sure that actual parameter is a type conforming to the
     constraint (the type appearing after `->' for the corresponding formal).

    Class: APPLICATION
    Feature: make

    For type: SORTED_TWO_WAY_LIST [COMP_TEST]
    Formal #1: COMP_TEST
    Type to which it should conform: COMPARABLE
...
```

We can change our class though to make it comply and therefor compile. The process is really rather simple, as illustrated in the new and improved class text that follows.

```
class COMP_TEST
inherit
  COMPARABLE
feature
 value: INTEGER

 is_less alias "<" (other: like Current): BOOLEAN
  -- Is Current object less than other?
  do
    Result := value < other.value
  end
end
```

The following code segment illustrates the use of SORTED_TWO_WAY_LIST and LINKED_LIST. Note well that each list feature used has the same apparent interface for each of the list classes.

```
 ll: LINKED_LIST [INTEGER]
 sl: TWO_WAY_SORTED_LIST [INTEGER]

 my_routine
  local
    ival: INTEGER
  do
    create ll.make
    create sl.make
    from ival := 4
    until ival = 0
    loop
      ll.extend (ival)
      sl.extend (ival)
      ival := ival - 1
    end
  end

 my_print_routine
  do
    from
      ll.start
      sl.start
    until ll.exhausted or sl.exhausted
    loop
      print ("Linked list item: " + ll.item + ",  ")
      print ("Sorted list item: " + sl.item + "%N")
      ll.forth
      sl.forth
    end
  end
```

The output of `my_print_routine` would be as follows.

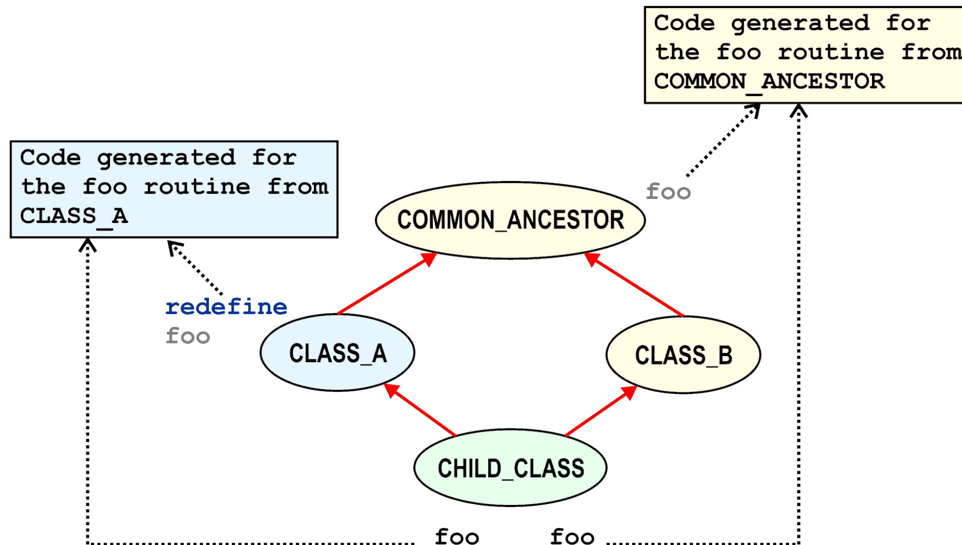
```
Linked list item: 4,  Sorted list item: 1
Linked list item: 3,  Sorted list item: 2
Linked list item: 2,  Sorted list item: 3
Linked list item: 1,  Sorted list item: 4
```

Note well that, although the apparent interfaces are the same, the *behaviors* are quite different. LINKED_LIST extends itself in order of insertion, whereas the SORTED_TWO_WAY_LIST extends itself in sorted (numerical in this case) order, as shown in the output.

Eiffel's Facilities for Feature Adaptation

Eiffel lets the programmer *redefine*, *undefine*, *rename, select,* and change the *export* status of inherited features. These adaptations exist to support class specialization of course, but also to resolve conflicts amongst repeatedly inherited features.

The following diagram illustrates the phenomenon of repeated inheritance, and a conflict that arises when one version of a repeatedly inherited feature has been specialized in an ancestor.

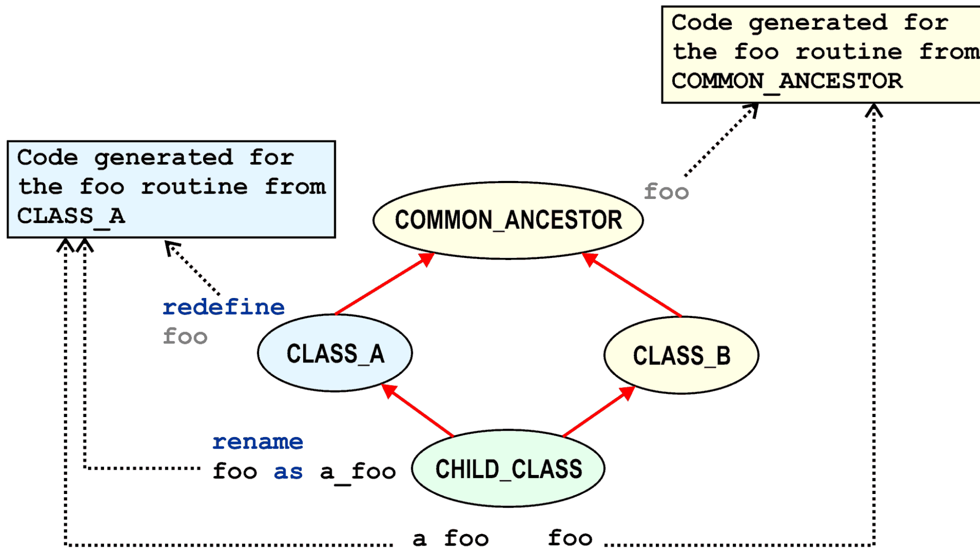


Which foo is it?

The redefined feature is the venerable `foo()`. CLASS_A *redefines* `foo()`, presumably to specialize it in some way. CHILD_CLASS then has two conflicting implementations of `foo()`, the one it inherits from CLASS_B (the original one from COMMON_ANCESTOR) and the one it inherits from CLASS_A. This conflict has to be resolved for compilation to succeed.

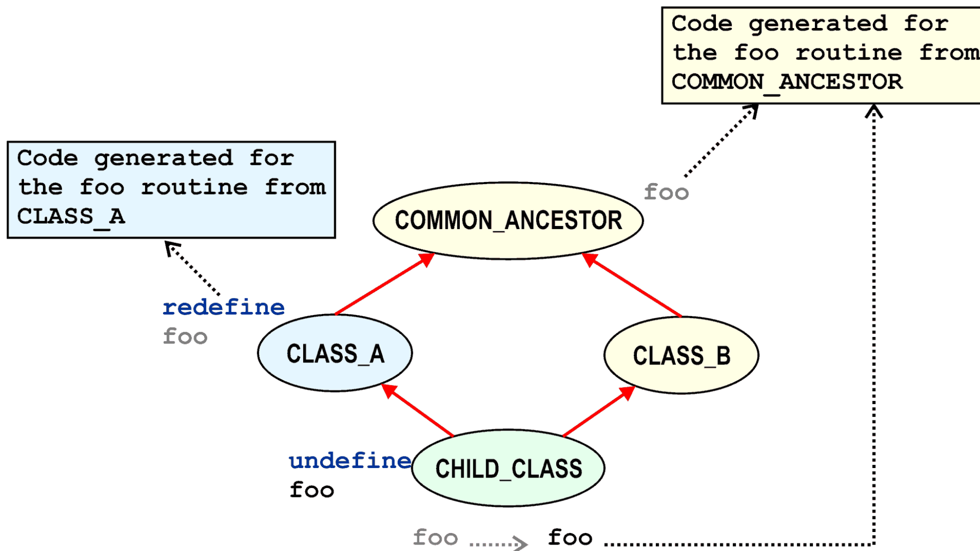
In some cases, the programmer wants to keep multiple instances of repeatedly inherited features. In this case, the inherited features must be *renamed* so that the compiler knows that you want them to be different (see *Rename* in the next section).

The next diagram shows the conflict resolved by *renaming* one of the inherited versions of `foo()` to `a_foo()`, leaving CHILD_CLASS with two routines, `a_foo()` and the old `foo()`.

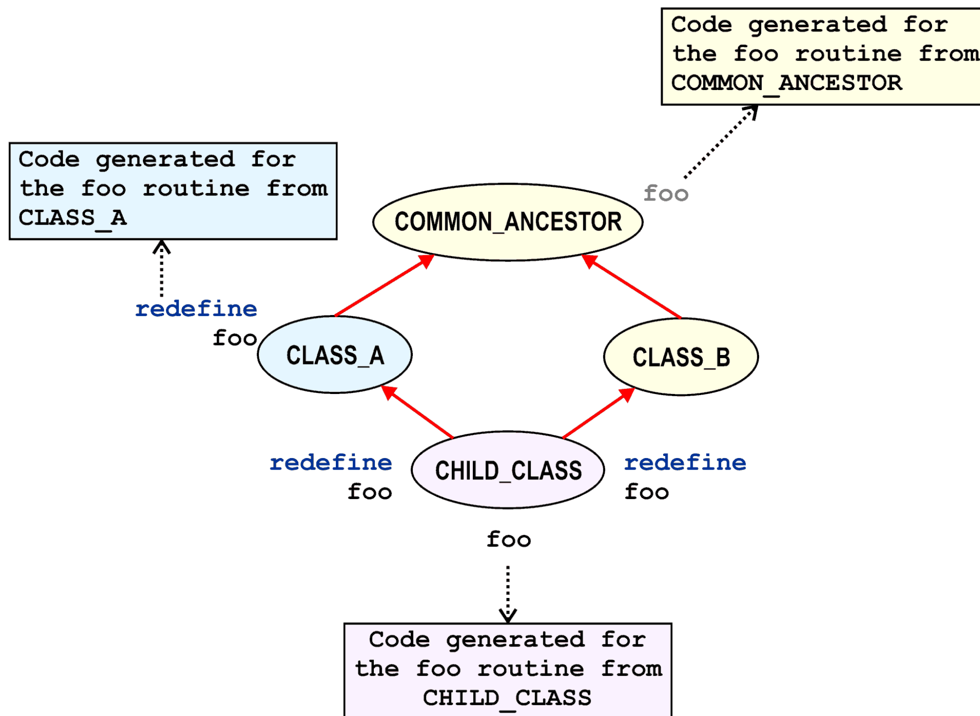


If the programmer had renamed the original version of `foo()` to something like `no_foo()`, then there would in fact be `no_foo()` like an `old foo()`. I just can't help myself, sorry.

In cases where only one version is wanted, the programmer might *redefine* or *undefine* one of the inherited versions, leaving the other as the only version, as illustrated in the following diagram.



If the programmer wants an implementation of `foo()` that differs from both possible inherited implementations, then the choice is to redefine both of them, as illustrated in the following diagram.



The programmer could, as an alternative, *undefine* one instance and *redefine* the other, but the effect is the same – the only remaining instance by that name is the instance implemented in CHILD_CLASS.

The Inherit Clause

The inherit clause of a class declares the direct parents of that class as well as the inherited features that the inheriting class will adapt (as seen in previous examples). The inherit clause follows the class name and precedes any features of the class, and the creation clause if any. The syntax of the inherit clause follows.

```
inherit
  <PARENT CLASS NAME>
   [[rename <rename pairs>]
   [export <export clause>]
   [undefine <undefine list>
   [redefine <redefine list>
   [select <select list>
  end]
  [<ANOTHER_PARENT CLASS NAME>
   [[rename <rename pairs>]
   [export <export clause>]
   [undefine <undefine list>
   [redefine <redefine list>
   [select <select list>
  end]]
```

Each of these subordinate clauses is covered in turn, below.

Rename

Rename says that from this point in the class hierarchy, the name of the given inherited feature will now be different. This means that for each instance of a class (for each object of this type) or of a proper descendent class, the name of that feature is the new name.

The syntax of the rename clause is:

```
rename
   <old_name> as <new_name>[,
   <another_old_name> as <another_new_name>]
```

where <old_name> is the name of a feature as it appears in the ancestor class and <new_name> is the name by which that feature is to be known hence. The name in the ancestor is not changed. The name is change only in the child and any of the child's descendants.

Each rename pair must include the '**as**' keyword. If there are multiple rename pairs, then the pairs must be separated by commas. They need not be on different lines, but this often provides the best readability.

In this example, the class A has a feature called **a_name**. Class B inherits class A, and with it the feature **a_name**. Class B wants to call that feature **b_name** for some strange reason unknown to us, but no doubt a legitimate one. It does this by renaming the feature in the inherit clause.

```
class A
feature
 a_name: STRING
end -- class A
```

This is class B's inherit clause.

```
class B
inherit
  A
    rename
      a_name as b_name
  end
 end -- class B
```

Now imagine a third class, C, that inherits class B. By what name is this feature known in class C? It is known by **b_name** because that is the name it inherited from its parent, class B.

```
class C
inherit
  B
feature
end -- class C
```

This is a contrived example and doesn't serve all that well to enlighten, but it is at least simple.

Here is perhaps a better example, from an illustrative point of view at least. This is the external specification of **make**, one of the creation routines of the STRING class.

By convention, most of our creation routines are called make *(or some variant thereof). This is not a rule of the language, but is a useful convention.*

```
make (n: INTEGER)
  -- Allocate space for at least `n' characters.
  require
    non_negative_size: n >= 0
  ensure
    empty_string: count = 0
    area_allocated: capacity >= n
  end
```

The class COMPANY_NAME renames STRING's **make** routine and defines a **make** of its own.

```
class COMPANY_NAME
inherit
  STRING
    rename
      make as str_make
  end

create
  make

feature
 make
  do
    str_make (32)
  end

end -- class COMPANY_NAME
```

This is a fairly common occurrence, and happens when the designer wants to have a "**make**" at this level of the hierarchy (with a different signature), but wants to retain the capabilities provided by the original. The **make** routine of STRING requires a starting size (strings are dynamically resizable in Eiffel, but it is sometimes more efficient to create them with a reasonable starting size.)

The COMPANY_NAME class has a predefined starting size (32) and so renames STRING's **make** to **str_make**, then calls **str_make** from its own creation routine (coincidentally also called **make**). Any client that wishes to create an object of type COMPANY_NAME then must do so by calling the **make** routine of COMPANY_NAME. Anyone calling **make** in an object by way of a handle of that type will be calling the new routine (the one with no arguments.) Note the creation clause in COMPANY_NAME. It lists only **make** as a creation routine. The inherited routine, now called **str_make**, is not a legitimate creation routine of COMPANY_NAME.

*A good convention (not seen in these examples) is to put all creation routines in a feature block whose export status is **NONE** (see Export). Because the creation routines are listed in the creation clause, they are accessible to a creation operation. By defining their export status as **NONE**, they become inaccessible except by that particular object. Even other objects of the same class have no access to them. Upon renaming and removing from the creation clause (as with str_make), they can be no longer called from outside of that object.*

The original **make**, from STRING still exists. It has not evaporated; it is only hiding behind an alias. This is the essence of renaming.

In case it wasn't obvious, any changes one makes in a child class have no effect on its ancestors, only on the adapted features in that child class and its proper descendants. You might give your parents gray hair, but you can't give them blue eyes!

Another common and useful application of renaming is to change the name of an inherited feature to reflect better the context of the child class. An example is often given where a STACK class is the child of a more generic data structure. This is a poor example of making data structures interchangeable and of making their interfaces standardized. It is also NOT the way that the STACK class is implemented in the Eiffel standard libraries. What it does well though, is illustrate an application of renaming. Let's begin by assuming that we are going to implement our DUMB_STACK class as a linked data structure, and so we want to inherit the TWO_WAY_LIST class.

```
class DUMB_STACK [G]
inherit
  TWO_WAY_LIST [G]
    rename
      put_front as push,
      first as top,
      remove as tw_list_remove
  end
feature
 pop
  -- Remove the top-most item
  do
    start
    tw_list_remove
  end
end -- class DUMB_STACK
```

Our class renames **put_front** as **push**, and **first** as **top**, presumably to make the feature names reflect more accurately the capabilities implemented by the features – make it look more stack-like. The **remove** feature is also renamed, to hide it from clients that might wish to remove an item without adhering to the stack policy (this can also be accomplished via the **export** clause). If a client tries to call the **remove** function, the compiler will reject the call because there is no feature called **remove** in class DUMB_STACK.

Note well that any clients of TWO_WAY_LIST still have access to the **remove** routine by the original name. If there is a *handle* of this type, and it refers to an instance of DUMB_STACK, then **remove** is quite accessible. Because of this, it is probably a better idea to redefine the **remove** routine (see *Redefine*, below).

Polymorphism makes this apparently simple concept a little less simple (but more powerful of course). Still it is not such a difficult concept. Here is an example where polymorphism might give the novice a little trouble. Our class hierarchy is deliberately simple.

Imagine that for some reason you have two automobile classes, one representing cars in the U.S. and one representing cars in Great Britain.

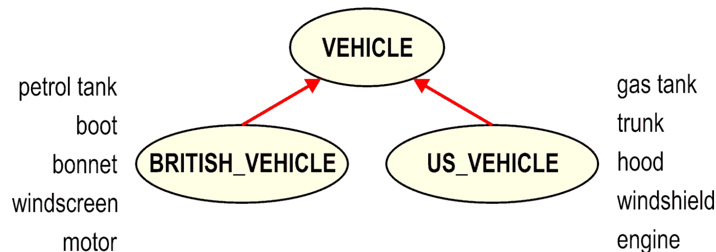


Figure 10 – British and US Vehicles

We all know that the British drive on the wrong side of the road (I kid the British). It is said that the U.S. and England are two countries separated by a common language. It appears true, because in addition to the minor detail of side-of-road preference, there are differences in terminology. For example, where Americans might call the sheet metal covering the engine compartment a *hood*, our friends from across the pond might call it a *bonnet*. To be as useful as possible, our simple class hierarchy should present the features of British vehicles, in their native language.

To avoid seeming too conciliatory, our revised hierarchy (Figure 11) will take on a decidedly American bias. Remember this is for clarity of illustration, not world peace. The previous hierarchy is more rational of course, but this one serves the purpose of illustration.

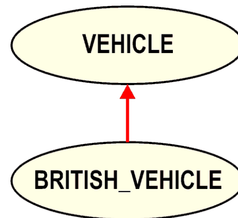


Figure 11 – British Vehicle as Subclass of Vehicle

The BRITISH_VEHICLE class's inheritance clause would then look like this.

```
class BRITISH_VEHICLE
inherit
  VEHICLE
    rename
      gas_tank as petrol_tank,
      trunk as boot,
      hood as bonnet,
      windshield as windscreen,
      engine as motor
  end
```

It should be easy to see that the capabilities and attributes represented by the renamed features still exist. They are simply known by different names. A client wishing to lift the **hood** of a BRITISH_VEHICLE would be unable to do so, because no such feature exists by that name, but try to lift the **bonnet**, and you're in business.

Now imagine a dealership that sells both American and British vehicles, and the sales manager doesn't care at all about the national origin of the cars he sells, only that they sell. To him, they are simply VEHICLEs, his inventory. How then does the sales manager refer to these features? His objects (from his point of view) are VEHICLEs, but clearly they must be either American (VEHICLE) or British (BRITISH_VEHICLE).

The answer lies in polymorphism. It turns out that the object knows (of course). For the compiler to know, the type of the handle to that object must be consistent with the features being called. The following code fragment illustrates a few different cases.

```
 1 foo
 2  local
 3    tv: VEHICLE
 4    tbv: BRITISH_VEHICLE
 5    inventory: LINKED_LIST [VEHICLE]
 6  do
 7    create inventory.make
 8    create tv.make
 9    inventory.extend (tv)
10    create tbv.make
11    inventory.extend (tbv)
12
13    print (inventory.first.hood)
14    print (inventory.last.bonnet)
15    tv := tbv
16    print (tv.hood)
17  end
```

Note that the local object **inventory** is a list of VEHICLEs. Simple polymorphism allows **inventory** to contain objects belonging to the class VEHICLE or its descendants, including BRITISH_VEHICLE.

On line 8, we create a new object of type VEHICLE, adding it to **inventory** on line 9. On line 10 we create a new BRITISH_VEHICLE object and add it to **inventory** on line 11. So far, so good.

On line 13, we try to print out the value of the **hood** attribute of VEHICLE (we had put **tv**, an object of type VEHICLE into the first position of the **inventory** list). This compiles successfully.

When we try to compile line 14, though, we get an error.

Why is this? After all, we added an object of type BRITISH_VEHICLE to the list in the last position, so the object at **inventory.last** should be a BRITISH_VEHICLE. It is, but the compiler doesn't know it, and can't legitimately know it, and shouldn't know it even if it could. The code says that **inventory** is a list of VEHICLEs and so the compiler enforces the type rules. How then do we get a BRITISH_VEHICLE object out of a list of VEHICLEs?

For the compiler to allow a reference to a feature of a class, it must have a *handle* to an object of that or a compatible class (type-compatible). Because a LINKED_LIST of VEHICLEs, from the compiler's point of view, is a collection of handles to VEHICLEs, regardless of their potential run-time types, the compiler treats that particular reference as one to VEHICLE.

This is where run-time support for polymorphism comes in. The run-time system isn't limited to seeing only handles. It can see *objects*. Line 14 of our example should be rewritten as the following lines.

```
14a     if attached {BRITISH_VEHICLE} inventory.last as bv then
14b       print (bv.bonnet)
14c     end
```

Line 14a uses an attachment test. The attachment test checks the run-time type of, in this example **inventory.last** to verify that (1) it is attached (not **Void**) and (2) that it is attached to an object of the given type, in this case BRITISH_VEHICLE.

The attachment test, in this example creates a temporary, *known-to-be-attached* handle (**bv**) whose scope is the attachment test. It is the temporary handle that is used at run-time to refer to the instance of BRITISH_VEHICLE. The condition in this case will be **True** because the last element of the list is in fact an object consistent with BRITISH_VEHICLE.

In practice, the logic might be a little denser and the results less certain, and so it might be reasonable to have logic for the **False** case as well.

```
14a     if attached {BRITISH_VEHICLE} inventory.last as bv then
14b       print (bv.bonnet)
14c     elseif attached {VEHICLE} inventory.last as uv then
14d       print(uv.hood)
14e     else
14f       print ("We're totally screwed")
14g     end
```

There is still a little voice nagging at you, saying "Yeah, but what about …?" Let's go back to our original code, the one with line 14 that wouldn't compile. Line 14 was:

```
14    print (inventory.last.bonnet)
```

We got a compilation error because the compiler's static view of things said that **inventory.last** was a VEHICLE and that class had no feature called **bonnet**. What if we instead had this line?

```
14    print (inventory.last.hood)
```

That would compile fine, but wouldn't we get some other kind of error? The object clearly was going to be BRITISH_VEHICLE, and objects of that class don't have a feature called **hood**.

This is not a problem. Again, this is where the beauty of polymorphism comes in. The compiler generates code that invokes the feature called 'hood' for an object of class VEHICLE. The renaming in Eiffel does the rest. Recall that the renamed feature doesn't go away, it merely hides behind an alias. The generated code really doesn't care about this. It's all offsets and addresses by the time it gets to be running code. So, the reference resolves quite correctly in the run-time environment because the object in question is the right object.

Remember (repeat after me) class is king at compile-time, and object is king at run-time.

Now what about lines 15 and 16? This is simple polymorphism at work.

```
15    tv := tbv
16    print (tv.hood)
```

Recall that **tv** is a handle to an element of class VEHICLE, and **tbv** is a handle to an element of class BRITISH_VEHICLE. Line 15 works nicely because the right-side value in the assignment (**tbv**) is at least the same type as the handle on the left side (**tv**). As a result of the assignment, **tv** now refers to a BRITISH_VEHICLE object.

This should cause a name conflict on line 16 then, right? No, it does not. On line 16, the **hood** feature of VEHICLE is accessed by way of a handle whose compile-time type is VEHICLE. The fact that the run-time object is a BRITISH_VEHICLE is not interesting at compile time. At run-time, the object's **bonnet** feature is called, but we know by now that **bonnet** is the same feature as **hood**, only called by a different name from handles of a child class.

Export

Eiffel allows the designer to change the export status of an inherited feature in two ways. First, any features that are redefined must associate the redefined implementation with a **feature** keyword. The **feature** keyword then defines the export status of the features (attributes, procedures, functions, constants) within that feature block.

The second method is to list the feature and its new export status in the **export** section of the inherit clause. Changing the export status does not otherwise affect the implementation of the given feature.

The syntax of the export clause is:

```
export
   {<a_class>} <a_feature>,
     <another_feature>
   {<another_class>} <yet_another_feature>
```

where <a_class> is the name of a client class (or the special labels ANY or NONE), and <a_feature> is the name of a feature as it appears in the current class (the new name if renamed, else the name from the parent). If there are multiple features listed, then they must be separated by commas. They need not be on different lines. Our example involves three classes, EX1, its descendent EX2, and a hypothetical client class EX3.

```
class EX1
feature {ANY}
 my_attr: INTEGER
 my_other_attr: INTEGER
feature {NONE}
 my_private_attr: INTEGER
end -- class EX1
```

The **feature** keyword in an Eiffel class defines the export status for all features between it and the next **feature** keyword (or the end of the class). When no class list follows the **feature** keyword, the export

status is ANY. In class EX1, the features **my_attr** and **my_other_attr** have an export status of ANY, and the feature **my_private_attr** has an export status of NONE (no other objects, regardless of class, can see that feature.)

In class EX2, we change the export status so that **my_private_attr** is now visible, but only to objects the class EX3 (and its descendants). We change also the export status of **my_attr.**
This might not be obvious because we also renamed my_attr to my_parent_attr. Because my_parent_attr is the name by which the feature is to be known hence, that is the name by which we alter its export status (and coincidentally why rename is the first section in the inherit clause). my_parent_attr is now not visible to other objects. The feature **my_other_attr** (not listed in the export clause) retains its original export status.

```
class EX2
inherit
 EX1
  rename
    my_attr as my_parent_attr
  export
   {NONE} my_parent_attr
   {EX3} my_private_attr
  end
end -- class EX2
```

Violation of export status is a compile-time error.

Sometimes the programmer wants a more extreme export limitation. One case where this might occur is when the object identified by a **once** function should not be modified. We use a **once** function returning a STRING as illustration.

```
 my_string_once: STRING
  once
    create Result.make_from_string ("This is a permanent value")
  end
 evil_gina
  -- Truncate our once-d string to confuse everybody else
  do
    my_string_once.clear_all
  end
```

You can see from the example that simply once-ing an object does not prevent access to its features. We can change the export status of some key features though.

```
class STINGY_STRING
inherit
  STRING
   export {NONE} ALL
  end
end
```

```
my_string_once: STINGY_STRING
  once
    create Result.make_from_string ("This is a permanent value")
  end
 evil_gina
  -- Cannot truncate our once-d string
  do
    my_string_once.clear_all
  end
```

Now **evil_gina** has no right to access the **clear_all** routine, because its export status has been shut off. Regrettably this is not enough to protect it from the true miscreant. Polymorphism lets the programmer set on self-destruction commit the original crime with relative ease, by simply making his handle, **tstr** of type STRING instead of anchoring it to the type of the once function.

```
evil_gina
  -- Truncate our once-d string by sneaking up on it using STRING
  local
    tstr: STRING
  do
    tstr:= my_string_once
    tstr.clear_all
  end
```

Of course, a user-defined class that never exports (or only via a child that redefines its export status) such routines would not be susceptible in this manner. In our STINGY_STRING case, we must redefine the dangerous routines if we want to be truly safe so that, regardless of the type of the handle, the object at runtime will execute the safe (probably neutered) routine.

Undefine

The **undefine** mechanism provides a means by which to make an already *implemented* feature unimplemented again, as if it were *deferred* in the parent (see *Deferred Features*). When you *undefine* an inherited feature, you are removing its implementation, but retaining its signature and its assertions.

The syntax of the undefine clause is:

```
undefine
   <a_feature>[,
   <another_feature>]
```

Where <a_feature> is the name of a feature as it appears in the parent class. When multiple features are undefined, they must be separated by commas. They need not be on different lines.

Our first **undefine** example is a collection of three classes. Class UD1 has a routine **to_string**. Classes UD2 and UD3 each inherit class UD1. Class UD4 inherits both UD2 and UD3, setting up a repeated inheritance condition. The compiler detects this and generates the correct code (only one copy of the **to_string** routine exists for UD4). The result of calling **to_string** for an object of type UD4 is the string "I am UD1".

```
class UD1
feature
 make
  do
  end
 to_string: STRING
  do
    Result := "I am UD1"
  end
end -- class UD1
```

```
class UD2
inherit
  UD1
feature
 some_attr: INTEGER
end -- class UD2
```

```
class UD3
inherit
  UD1
feature
 some_other_attr: INTEGER
end -- class UD3
```

```
class UD4
inherit
  UD2
  UD3
create
 make
feature
end -- class UD4
```

Now we change UD3 to redefine the implementation of `to_string` and set up a conflict between the inherited versions of `to_string`.

```
class UD3
inherit
  UD1
    redefine
      to_string
  end
feature
 to_string: STRING
  do
    Result := "I am no longer UD1"
  end
 some_other_attr: INTEGER
end -- class UD3
```

This now generates the following error message.

```
Error code: VMFN
Error: two or more features have same name.
What to do: if they must indeed be different features, choose different
  names or use renaming; if not, arrange for a join (between deferred
  features), an effecting (of deferred by effective), or a redefinition.

Class: UD4
Feature: to_string: STRING inherited from: UD3 Version from: UD3
Feature: to_string: STRING inherited from: UD2 Version from: UD1

------------------------------------------------------------------------------
Degree: 4 Processed: 2 To go: 0 Total: 2
```

We have at least three options for removing this conflict, **rename**, **redefine** and **undefine**. This time we will use the **undefine** mechanism. We can undefine one of the inherited versions (the one from UD2), removing its implementation (from **UD4**) and making it behave *as if* it were *deferred* in the parent. Now the version we inherit from UD3 becomes the only implemented version. This means that we not only remove the multiply defined condition, but we also resolve the "deferred" condition we caused by undefining the feature from UD3.

```
class UD4
inherit
  UD2
    undefine
      to_string
  end
  UD3
create
 make
feature
end -- class UD4
```

Now the **to_string** routine from UD4 is the one inherited from UD3. The version from UD2 no longer exists within UD4's scope. The result of calling **to_string** for an object of type UD4 is now the string "I am no longer UD1".

Redefine

When you redefine an inherited feature, you change its implementation, its signature, its assertions, or all three. Any changes must conform to the covariant typing rules of the language.

The syntax of the redefine clause is:

```
redefine
   <a_feature>[,
   <another_feature>]
```

Where <a_feature> is the name of a feature as it appears in the parent class. When multiple features are redefined, they must be separated by commas. They need not be on different lines.

Eiffel lets you redefine a routine to an attribute, but not vice versa. Attributes can be redefined, but only to more specific types (covariance). You can redefine external routines (e.g. calls to C functions), but you cannot redefine an external routine to become an Eiffel routine, and vice versa.

Our first redefinition example is a simple hierarchy of three classes. The topmost class defines a routine called **to_string** that yields a string representation of the object. The child and grandchild of this class redefine that routine as well as the creation routine, **make**.

First the grandparent class.

```
class RC1

create
  make

feature
 make
  do
    attr1 := 5
  ensure
    initialized: attr1 /= 0
  end

 to_string: STRING
  do
    Result := attr1.out + "%N"
  ensure
    valid_result: Result /= Void and then not Result.is_empty
  end
 attr1: INTEGER
end -- class RC1
```

```
class RC2
inherit
  RC1
    redefine
      make, to_string
  end

create
  make
feature
 make
  do
    Precursor
    attr2 := 10
  ensure then
    initialized: attr2 /= 0
  end

 to_string: STRING
  do
    Result := Precursor
    Result.append_integer (attr2)
    Result.append ("%N")
  end

 attr2: INTEGER
end -- class RC2
```

```
class RC3 inherit
  RC2
    redefine
      make, to_string
  end
create
  make
feature
 make
  do
    attr1 := 5280
    attr2 := 1999
    attr3 := 42
  end

 to_string: EASY_STRING
  require else
    never_mind: True
  do
    create Result.make (80)
    Result.append ("Jimmy cracked corn and I don't care%N")
  end

 attr3: INTEGER
end -- class RC2
```

The **make** routine (procedure) has different implementations at each of the three levels of our hierarchy. Each descendent redefines the implementation to include initialization of the attributes known at that level of the hierarchy.

The RC2 implementation *extends the postcondition* introduced by RC1 to ensure that the attribute (**attr2**) introduced in RC2 is included in the initialization. RC2 uses the Precursor keyword to invoke the implementation inherited from RC1, so that RC2's **make** routine simply expands, rather than replaces the inherited implementation.

RC3 is a bit of a pain and decides to rebel against the constraints of its parents (RC3 must be a teenager.) Rather than rely on the implementations of **make** as inherited from its ancestors, RC3 has decided to re-implement from scratch. This is allowed, but shows poor judgement of course. What the author of RC3 might or might not realize is that the postcondition defined by its ancestors are still in force (covariance at work), and so an assertion exists that ensures that at least **attr1** and **attr2** are initialized when **make** finishes.

The **to_string** routine (function) also has different implementations at each of the three levels of the hierarchy. RC2 redefines its version as an expansion of the inherited version, by using the Precursor. Once again, RC3 tries to carve its own path. Its version of **to_string** not only replaces the inherited implementation entirely, but also changes the result type to EASY_STRING (a descendent of STRING) and redefines the *precondition* by OR-ing in the predicate True – effectively neutralizing any inherited preconditions. The *postconditions* inherited from its ancestors however remain unchanged. The type change is correct because EASY_STRING is a STRING. The changed precondition is correct, per covariant rules, because it is relaxed relative to its parent.

Extractable Documentation

Eiffel supports the notion of extractable documentation (*this is not strictly part of the topic of this paper, but it is somewhat related, and can be helpful*.) Eiffel classes then can be projected in several different forms.

The *interface* form of a class shows the interfaces of all exported features of a class, along with the features' preconditions and postconditions, and any class invariants.

The *flat* form of a class shows all of the class's implemented features as well as its inherited features.

The Eiffel IDE also offers the option to generate documentation for an entire project (via the Project menu 'Generate Documentation' item., in a wide variety of forms.

Precursor

The **Precursor** keyword makes a lot of things relating to inheritance quite a bit easier. As seen in the previous examples, the **Precursor** keyword substitutes for the inherited implementation of a redefined feature. The preceding examples were admittedly rather simplistic. They did not show the use of **Precursor** for redefined routines with arguments, or for cases of multiple inheritance. Examples of these follow.

In this example, class PC1 defines a routine called **make** that (in addition to the being the creation routine for the class) initializes the **name** attribute to the given argument. A child class, PC2, redefines the **make** routine by first relaxing the precondition to allow any value, even a Void, and then checking the value in its implementation. Finally, the redefined routine calls the inherited version with the new argument, using the **Precursor** keyword.

```
class PC1
create
 make
feature
 make (v: detachable STRING)
  require
   arg_is_valid: v /= Void and then not v.is_empty
  do
    name := v
  end
 name: STRING
 end -- class PC1
```

```
class PC2
inherit
  PC1
    redefine
      make
  end
creation
 make
feature
 make (v: detachable STRING)
  require else
    v = Void or else v.is_empty
  local
    tstr: like name
  do
    if v = Void or else v.is_empty then
      tstr := "Dummy"
    else
      tstr := v
    end
    Precursor (tstr)
  end
end -- class PC2
```

Note that the example uses **detachable STRING** as the argument to make. This lets us use the example in a Void-safe context.

In this next example, the **Precursor** keyword is used in a multiple inheritance context. Class PM1 defines a routine called **to_string**. Class PM2 also defines a routine called **to_string** (an entirely different

routine, just the same name). Finally, class PM3 inherits both of these classes, and redefines both versions of **to_string**, using a *qualified* **Precursor**.

Note that PM3 also renames the **make** routines it inherits from PM2 and PM1, then calls them by their new names in its own **make** routine. It could just as easily have used **redefine** and **Precursor**.

```
class PM1
create
 make
feature
 make do end
 to_string: STRING
  do
    Result := "I am PM1"
  end
end -- class PM1
```

```
class PM2
create
 make
feature
 make do end
 to_string: STRING
  do
    Result := "I am PM2"
  end
end -- class PM2
```

```
class PM3
inherit
 PM1
  rename make as pm1_make
  redefine to_string
  end
 PM2
  rename make as pm2_make
  redefine to_string
  end
create
 make
feature
 make
  do
    pm1_make
    pm2_make
  end
 to_string: STRING
  do
    Result := {PM1}Precursor
    Result.append ("%Nand%N")
    Result.append ({PM2}Precursor)
  end
end -- class PM2
```

Covariance

Covariance has been mentioned a few times already. Maybe an explanation of it would help.

Inheritance is arguably the most powerful characteristic of Object Orientation. Eiffel's typing and assertion support is *covariant* with respect to inheritance.

Covariance is not so strange a concept as it sounds. Simply put, it means that a child class must do as well as or better than its parent. More specifically, the features inherited by a child class must each do as well as or better than the corresponding features in the parent class.

When a programmer redefines a feature in a descendent, the programmer must make certain that the signature is consistent (type, position and count) with the parent.

> ***The classes of elements that make up the signature of a redefined routine must be either the same as those in the parent, or proper descendants of those in the parent.***

The result type of a redefined function then must adhere to this rule. Similarly, any arguments to a redefined routine must adhere to this rule.

The programmer must also make certain that the assertions that define the contract for that routine (or class invariant) are consistent as well, according to covariance. Specifically,

> ***A descendent routine's <u>preconditions</u> must be the same as or <u>weaker</u> than the parent's***
> ***A descendent routine's <u>postconditions</u> must be the same as or <u>stronger</u> than the parent's***

In other words, the child must at least live up to the contract defined by the parent, but can "do a better job" by making it easier on the client (weaker preconditions) or more beneficial to the client (stronger postconditions).

Select

The select clause allows the programmer to select from a parent one of a set of repeatedly inherited features that have been renamed and therefor conflict. They are still the same features, but they now go by different names. The syntax of the select clause is:

```
select
   <a_feature>[,
   <another_feature>]
```

Where <a_feature> is the name of a feature either as it appears in the parent class (if not renamed), or if renamed in this class, the new name. When multiple features are selected, they must be separated by commas. They need not be on different lines.

For our example we revisit the undefine example, with classes `UD1`, `UD2`, `UD3`, and `UD4`. Recall that we had several options for resolving the conflict, and that we chose undefine. Now we will use *rename* and *select*. To regenerate the problem, we roll back our implementation of `UD4` to look like this.

```
class UD4
inherit
  UD2
  UD3
create
 make
feature
end -- class UD4
```

We again get the error message shown in the undefine example above. Now to creep up on the proper use of select, we change `UD4` to rename the `to_string` feature from `UD2`.

```
class UD4 inherit
  UD2
   rename to_string as ud2_to_string
  end
  UD3
create
 make
end -- class UD4
```

This results in the following error from the compiler.

```
Error code: VMRC(2)
    Error: conflict between versions of a repeatedly inherited feature.
    What to do: list one of the versions in exactly one Select clause.

    Class: UD4
    In parent UD2: ud2_to_string: STRING
    In parent UD3: to_string: STRING

-------------------------------------------------------------------------------
Degree: 4 Processed: 1 To go: 0 Total: 1
```

Being the cooperative folks that we are, we do as the message suggests and list exactly one of the versions in a **select** clause (we opt for the one from UD3, as before), and now the compiler is happy once more. You can think of Select as choosing the *name* by which you want the feature to be known (from the names available).

```
class UD4 inherit
  UD2
   rename to_string as ud2_to_string
  end
  UD3
   select
     to_string
  end
create
 make
end -- class UD4
```

Deferred Features

Creating classes that could someday be inherited by others is an interesting exercise (to say the least). Sometimes it is necessary and beneficial to identify a capability or attribute, at least in a general sense, in a parent, but leave the specifics to a child. Eiffel helps in this area by providing explicit support for this deferral.

Any class can have one or more features whose existence is assured, but whose implementation is deferred (class hierarchy wise). Such a class is called a *deferred* class. Deferring is not a function of time, but of space. This concept is sometimes difficult for programmers new to Eiffel and Object Orientation. The programmer can defer implementation decisions to a space lower in the class hierarchy (a.k.a. a descendent). These are decisions about structure, and algorithm, and sometimes about specification of type.

Here is a simple example of two classes, one a deferred class (the parent), and the other a simple class that inherits the deferred class and *implements* for that deferred class the deferred feature.

```
deferred class D_PARENT
feature
my_message: STRING
  deferred
  end
end -- class D_PARENT
```

Here is the child class.

```
class D_CHILD
inherit
  D_PARENT

feature
my_message: STRING
  do
    Result := "Hello World%N"
  end
end -- class D_CHILD
```

The child class does not declare that it is going to implement the deferred feature (see *The Inherit Clause*, for inheritance syntax), but the child's implementation must obey the consistency rules.

A deferred class is incomplete, and cannot be created. Therefor there is no such thing as a deferred *object*. Attempts to create an object of a deferred class fail at compile time.

The ANY Class

Paraphrasing, only slightly, the Eiffel language reference manual:

> ***"Any class other than ANY which does not include an explicitly written Inheritance clause is considered to have an implicit clause of the form***
> *inherit* ANY***"***

This seems innocuous enough, but can confuse the novice when confronted with error messages during compilation. These are usually just the result of inadvertent repeated inheritance. An example is in order. Imagine a simple class (you've seen it before in many forms).

```
class ANY_TEST
 create
  make
feature
 make
  do
    print ("Hello World%N")
  end
end -- class ANY_TEST
```

This compiles and runs and all is right with the world. Now let's inherit a couple of otherwise harmless looking classes that have only constants (for simplicity). Here are the bodies of those classes.

```
class MY_CONSTANTS
feature
 K_some_dumb_constant: INTEGER = 1
end -- class MY_CONSTANTS
```

```
class MY_OTHER_CONSTANTS
feature
 K_some_other_dumb_constant: INTEGER = 2
end -- class MY_OTHER_ONSTANTS
```

Our test class now looks like this.

```
class ANY_TEST
inherit
  MY_CONSTANTS
  MY_OTHER_CONSTANTS
create
  make
feature
 make
  do
    print ("Hello World%N")
  end
end -- class ANY_TEST
```

So far so good. But what if one of the inherited classes has redefined a feature from ANY? Now you have trouble.

```
class MY_CONSTANTS
inherit
  ANY
    redefine
      copy
  end
feature
 copy (v: like Current)
  do
  end
 K_some_dumb_constant: INTEGER = 1
end -- class MY_CONSTANTS
```

This creates a conflict because now MY_CONSTANTS has changed the implementation of **copy** and so the two identical versions we inherited before are now two different ones and so cannot be merged invisibly by the compiler. The programmer must step in and resolve the conflict.

Now when you try to compile, you get an error message from an inheritance you never thought you had.

```
Error code: VMFN
    Error: two or more features have same name.
    What to do: if they must indeed be different features, choose different
    names or use renaming; if not, arrange for a join (between deferred
    features), an effecting (of deferred by effective), or a redefinition.

    Class: ANY_TEST
    Feature: copy (other: (like Current) MY_OTHER_CONSTANTS)
      inherited from: MY_OTHER_CONSTANTS
      Version from: GENERAL
    Feature: copy (other: (like Current) MY_CONSTANTS)
      inherited from: MY_CONSTANTS
      Version from: MY_CONSTANTS
-------------------------------------------------------------------------------
Degree: 4 Processed: 2 To go: 0 Total: 2
```

This results from the rule stated at the beginning of this section, that any class lacking an inheritance clause is assumed to have an implicit one that inherits ANY.

In this case, the class MY_OTHER_CONSTANTS implicitly inherits ANY, and so its version of **copy** conflicts with the one redefined in MY_CONSTANTS.

What is the best way to handle this? You could simply rename all of the features that are conflicting from the constants class (you won't be using them anyway). You could otherwise undefine each of them in your constants class, by inheriting ANY and declaring the constants class to be deferred. The rename approach is the more common.

Resolving Inheritance Conflicts

Rename *versus* Redefine *versus* Undefine

If you exploit multiple inheritance, then you will encounter inheritance conflicts. There are choices to be made about the means by which to resolve these conflicts. The trick is to figure out which choice is the correct one for that situation.

The answer is found in another question. *Which capability or attribute do you want?*

If only one, then make certain you get that one. If you want more than one, that is, you want to use (or modify) the capabilities provided by more than one ancestor's version, then you have to rename at least one of them (then you have two features and no more conflict.)

It is important to understand the difference between renaming, un-defining and redefining.

Rename merely changes the name of the feature in the inheriting class, it does not remove it or reimplement it

Redefine changes the implementation of a feature in the inheriting class

Undefine eliminates the implementation of a feature from the inheriting class

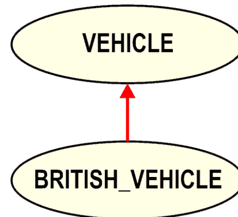
Single inheritance does not avoid these conflicts, but repeated inheritance gives us more opportunity for conflict resolution (and a greater chance of having conflicts).

As a rule, the correct choice of resolution often follows the logic in this table.

Feature Name	Feature Body	Resolution
Same	Same	Compiler handles repeated inheritance automatically
Same	Different	Rename one to retain two distinct features **OR** Undefine one of them to yield a single feature
Different	Same	Rename one to make them the same name **OR** Select one by name in a select clause
Different	Different (redefined)	Select one by name in a select clause
Different	Different (unique)	Do Nothing – Not a repeated inheritance

Which Feature am I Calling?

A common question is "*When I redefine a feature, how do I know which feature I will get at run-time?*" The answer again, is that *the object knows*. Remember that the compiler generates code based on the static class definitions and the *handles* (instances at compile time) associated with them. At run-time, you are dealing with objects. Each object is created at a level of its class hierarchy and so the features of that object come from that level, regardless of the type of the handle used to access it. For illustration, let's revisit the BRITISH_VEHICLE example, expanding our silly classes to include some redefinition as well, **align_in_roadway**.



```
class VEHICLE
feature
 gas_tank: CAR_PART
 trunk: CAR_PART
 hood: CAR_PART
 windshield: CAR_PART
 engine: CAR_PART

 align_in_roadway
  do
   print ("On the right side of the road%N")
  end
end -- class VEHICLE
```

```
class BRITISH_VEHICLE
inherit
  VEHICLE
    rename
      gas_tank as petrol_tank,
      trunk as boot,
      hood as bonnet,
      windshield as windscreen,
      engine as motor
    redefine
      align_in_roadway
  end
feature
 align_in_roadway
  do
   print ("On the left side of the road, actually%N")
  end
end -- class BRITISH_VEHICLE
```

The code in the client, to exercise the redefinition, and to prove the point, would be as in the following fragment.

```
 1 foo
 2  local
 3    tv: VEHICLE
 4    tbv: BRITISH_VEHICLE
 5    inventory: LINKED_LIST [VEHICLE]
 6  do
 7    create inventory.make
 8    create tv.make
 9    inventory.extend (tv)
10    create tbv.make
11    inventory.extend (tbv)
12
13    show_alignment (inventory.first)
14    show_alignment (inventory.last)
15  end
16 show_alignment (v: VEHICLE)
17  -- I do not know whether this is a simple VEHICLE or not,
18  -- but then I don't care - the object knows
19  do
20    v.align_in_roadway
21  end
```

The output of this code fragment would be:

```
On the right side of the road
On the left side of the road, actually
```

Why is this so? Let's look at the mechanics. The compiler sees that, on line 8, you want to create an object of type VEHICLE, and so it generates the appropriate code to do so. On line 10, you ask to create an object of type BRITISH_VEHICLE and the compiler obliges by generating the code necessary to accomplish that request.

At run-time, the generated code creates a LINKED_LIST whose elements are expected to be at least of type VEHICLE, if not of a specialized descendent. The compiler has already validated the typing rules. Line 8, when executed, creates a VEHICLE object, whereas line 10 creates a BRITISH_VEHICLE object. Each is added to the list already created.

Now when the code for lines 13 and 14 are executed, the **show_alignment** routine is called first with a VEHICLE object (the first item in the **inventory** list) then with a BRITISH_VEHICLE object (the second and last item in the list).

The **show_alignment** routine hasn't a clue about the type of the object being passed. Remember that the compiler already made certain that it would be at least a VEHICLE. When line 20 if finally executed, it is done so with the object passed to the routine. The first time, the object is a VEHICLE, and the next time it is a BRITISH_VEHICLE. Each time, it is the object's already *bound* **align_in_roadway** routine that is called. No further type checking or magic is required.

Why doesn't the call in **show_alignment** result in the string "On the right side of the road" both times? -- Because the *object* is not a simple VEHICLE both times. The second time it is a BRITISH_VEHICLE. The routine doesn't know this; the *object* does.

Run-Time Typing

In the previous examples, we have seen a lot of static binding. That is part of Eiffel's strong typing. Performing whatever bindings are possible at compile time makes development quite a bit saner. Still there is sometimes the need to determine the type of an object at run-time. We saw a glimpse of this in our example of the BRITISH_VEHICLE when covering the **rename** clause. If you recall, the code fragment looked like this.

```
14a     if attached {BRITISH_VEHICLE} inventory.last as bv then
14b       print (bv.bonnet)
14c     elseif attached {VEHICLE} inventory.last as uv then
14d       print(uv.hood)
14e     else
14f       print ("We're totally screwed")
14g     end
```

Line 14a contains an attachment test. The type of **inventory.last** is VEHICLE, and may be VEHICLE or BRITISH_VEHICLE. The attachment test, checking for **inventory.last** being of type BRITISH_VEHICLE, adds a temporary handle '**bv**' whose scope is limited to the attachment test. If indeed, the object at **inventory.last** is of type BRITISH_VEHICLE, then the temporary handle '**bv**' will be non-Void, guaranteed.

At compile time, you can have pretty much any combination of types in the attachment test. It is after all a test only tried at run-time. For example, the following code would compile just fine, though it would be useless.

```
 foo
  local
    bogus: SOME_UTTERLY_UNRELATED_CLASS
  do
    create bogus.make
    if attached {BRITISH_VEHICLE} bogus as bv then
      print ("How did this happen?  It never would")
    end
  end
```

There is another option that can be used to verify attachment at run-time when you absolutely, positively know that the test will succeed. That option uses the exception mechanism in place of basic control flow. Our now-famous 14a example could be rewritten as something like this.

```
14a     check attached {BRITISH_VEHICLE} inventory.last as bv then
14b       print (bv.bonnet)
14c     end
```

It appears that the assignment test will fail at run-time (base solely on our knowledge, as programmers, of the current class taxonomy.) Can you be certain? The assignment test is not intended to introduce chaotic and misguided coding practices. It is however, a very nice mechanism for eliminating from our code any need for tags, flags or other primitive mechanisms for determining the types or our objects. Prudent use of the assignment test can help reduce the size and complexity of our code.

Conclusion

Newcomers to Eiffel are sometimes “frightened and confused” when they encounter inheritance conflicts, or even non-conflicting inheritance adaptations. The mechanisms are very powerful, but they are, as is the rest of Eiffel, consistent and rational. Once one understands the relationships between compile-time rules and run-time realities, these mechanisms become quite clear.

Eiffel provides a rich and expressive inheritance mechanism, while keeping the syntax clean, regular and fairly simple. With a little practice, even a newcomer to Eiffel should become comfortable with these mechanisms, and the power they bring to problem solving.