Inheritance in Eiffel

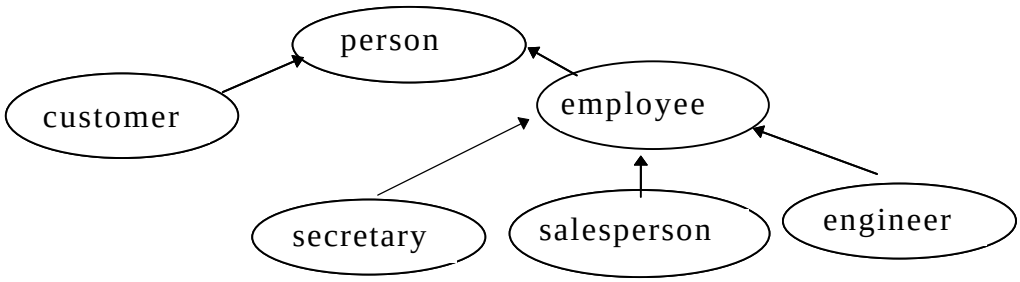
Eiffel supports multiple inheritance, i.e. a class C can inherit from more than one class. Inheritance in used in two major ways:

- **Specialization**
- **Generalization**

Specialization

A class C is a specialization of a class P, if C has all the properties (attributes & features) of P plus maybe a few more defined itself. The extra properties would distinguish an object from class C from one from class P.

In modelling a company we may have the inheritance classification. Thus a Salesperson is a specialization of an Employee, i.e. a Salesperson is an Employee with the duty of selling. In creating new class it is of great use to have similar classes already constructed and the new class can inherit all the properties of the similar class.



Assume the Employee class has a routine months_pay. In a non OO language a separate routine e.g. secretary_pay, engineer_pay, would be needed. In an OO language just the appropriate routine for months_pay would be called. The routine months_pay would be redefined in each class inheriting from employee. In particular, in a non OO language we could have (in Eiffel syntax)

```
inspect emp.type
        when secretary_type then
                pay := secretary_pay(emp)
        when salesperson_type then
                pay := salesperson_pay(emp)
        when engineer_type then
                pay := engineer_pay(emp)
end
```

In imperative languages, 'variant records' or 'tagged records' would be used to support this.
In Eiffel all we need is

```
pay := emp.months_pay
```

and the runtime would use the correct routine.
If another category, say Trainee, was added then all that would need to change is that a months_pay routine would be redefined for a trainee. A further case statement would be needed in a non OO language. (See Ch. 6 Switzer "Eiffel, an Introduction")

Generalisation

We may already have the classes Salesperson, Secretary etc, and decide that these are specializations of a class Employee; we generalise from the particular to the general. Common features in the particular class can be moved up the general class. In particular, the more general class may have 'deferred features and so it will be a 'deferred' class. The most general class in Eiffel is called GENERAL.

The Class Hierarchy in Eiffel

In Eiffel all classes are classified into an hierarchy, analogous to the "Linnean classification"of livings things. Every class inherits from the class ANY. We say, every class 'conforms' to ANY.

To make allowances for language changes and usage across platforms Eiffel has 2 other classes GENERAL and PLATFORM which are in effect the same as ANY but are are ancestors of ANY, ie ANY is heir to PLATFORM which in a heir of GENERAL. These classes are deferred class in that their features are defined in descendent classes.

Deferred classes and routines

These universal classes are so general that they have no implementation and so are "deferred" classes. The implementations are deferred until created by later conforming classes, e.g. in the class LIST_SET, it is possible to implement the 'deferred' routines e.g. 'copy' and 'is_equal'. Deferred classes are useful as design tools in that the classes with pre and post conditions can be created without having to be implemented. The implementation stage can be carried out after the design has been checked.

The class GENERAL (ANY, PLATFORM)

The class GENERAL (ANY) has, among others, the following features:

```
frozen  do_nothing -- Execute a null action.

frozen  void: NONE
```

'Frozen' features cannot be redefined and in effect are built into the Eiffel system but some routines have a frozen version and a version that is not which can be redefined in a conforming class.

The class NONE is a fictional or virtual class, there is no text in the class, which is a descendant of every class, i.e. it inherits from every class. In the class GENERAL there is a declaration

frozen void : NONE

but this is viewed as a dummy declaration.
If NONE inherits from every other class then it has the properties of every single class (even classes that have not been written yet?). In particular,we can regard void as being in every class.

```
frozen  deep_equal (some: GENERAL;  other: like  some): BOOLEAN
-- Are some and other either both void
-- or attached to isomorphic object structures?

frozen  equal (some: GENERAL;  other: like  some): BOOLEAN
-- Are some and other either both void or
-- attached to objects considered equal?
```

In using routines with parameters of type GENERAL, then any entity from a class which conforms to GENERAL, i.e. any class, can instantiate the parameter. So the routine equal **is in effect included in all classes. The routine** deep_equal **checks equality 'recursively' through it sub-components.**

```
is_equal (other: like  Current): BOOLEAN
        -- Is other attached to an object  equal to current?
        require
                other_not_void: other /= void
```

The routine is_equal can be redefined and with it the routine equal as Eiffel defines the frozen routine equal **in terms of** is_equal**.**

```
frozen  clone (other: GENERAL): like  other
-- For non-void other, clone calls copy; to change
--  copying/cloning semantics, redefine copy.
ensure
        equal:  equal (Result, other)

copy (other: like  Current)
-- Update current object using fields of object
-- attached to other, so as to yield equal objects.
```

The routine clone is defined in terms of copy, so that, in effect

```
a := clone(b)
```

becomes

```
!!a
a.copy(b)
```

Also included in the class GENERAL, are the 'print' routines that are available, if appropriate, in all classes.

```
io: STD_FILES -- Handle to standard file setup

out: STRING
-- New string containing terse printable  representation of current object

print (some: GENERAL)
-- Write representation of some on standard output
```

Eiffel Syntax for Inheritance

```
class Cname
inherit
        B1
                rename
                        f1 as g1
                        ...
                export  -- over-rides 'export rules'
                        { D,E}
                        f, g      -- f,g  exported only to D and E
                        ...
                redefine
                        h1, h2
                        ...
                end -- B1
        B2
                ...
creation
        etc
end
```

Visiblity and Exporting

The keyword "feature" can be annotated to indicate which features should be exported and to where. The effect applies to the routines before the next keyword "feature". Assume

```
class P
        ...
        feature
                routine0 -- visible to all client classes
        feature {NONE}
                routine1  -- hidden from all client classes
        feature {A,B}
                routine2 -- exported to A and B
        feature {P}
                routine3 -- exported just to class P
        ...
end -- class P
```

then, e.g.

```
class A
        ...
        x: P  -- class A is a client of P
        x.routine2  -- OK
but     x.routine3  not OK, routine3 is exported just to P
        ...
end -- A
```

Copy and Equal for LIST_SET

We can redefine the inherited 'deferred' routines for LIST_SET.

```
class  LIST_SET [G]
        inherit ANY
                redefine
                        copy, is_equal
                end
feature {NONE}
        first_node, cursor : NODE[G]
feature

        copy(other: like current) is -- see class GENERAL;
                require
                        other /= void
                local
                        prev, pres : NODE[G]
                do
                        if other.empty then
                                first_node := void
                                count := 0
                        else
                                from
                                        other.start
                                        !!pres
                                        first_node := pres
                                        count := other.count
                                until
                                        other.off
                                loop
                                        pres.set_item(other.item)
                                        if prev /= void then
                                                prev.set_next(pres)
                                        end
                                        prev := pres
                                        !!pres
                                        other.forth
                                end
                        end -- copy
```

Note:
The routine copy preserves the order of the original list_set, other.

```
is_equal(other: like current):BOOLEAN is
        local
                i, j : INTEGER
        do
                if count /= other.count then
                        result := false
                else
                        from
                                other.start
                                start
                                i := 0
                                j := count
                        until
                                i = j
                        loop
                                if equal(other.item, item) then
                                        other.forth
                                        forth
                                        i := i+1
                                else
                                        j := i
                                end
                        end
                        result :=  i = count
                end
        end -- is_equal
```

Note:
Two sets can be regarded as equal if they have the same elements but the notion of equals used here is stonger. Not only have the list_sets the same elements but the 'traversing' order is also the same. In particular, after a.copy(b), we have that a.is_equal(b). But two sets A and B may have the same elements but would not be regarded as being equal using is_equal due to different traversing orders in A and B.

Exercise: **Write a routine, set_equal, such that**
x.set_equal(y) ≡ x and y have the same elements.