Executing an Eiffel Class

Since the only top-level facility in Eiffel is a class we have to tell the compiling system which class we are using as the starting class or **root** class.

Let us consider a root class, TESTSQRT, for running the square root functions. This class usesd routines from the STD_FILES class for input and output. The STD_FILES class includes (among others) the following routines.

Input Procedures -- reading

The 'read...' routines are procedures and don't return a result; the result 'read...' is stored 'last...'.

These procedures read the input from standard input (window).

```
read_character      -- or readchar
read_integer        -- or readint
read_real           -- or readreal
read_word           -- or readword

read_line           -- or readline
-- inputs a full line and  puts result  in last_string

read_stream(nb:INTEGER)      -- or readstream(nb:INTEGER)
-- read a stream of at most nb characters from input
-- and make the result available in last_string

next_line  -- move to next line in input.
```

Last Functions

```
last_character      -- or  lastchar

last_integer        -- or  lastint

last_real           -- or lastreal

last_string         -- or laststring
```

These functions get their values from their respective 'read...' procedures.

Output procedures -- put…

put_character (c : CHARACTER)	-- or--	putchar (c : CHARACTER)
put_integer (n : INTEGER)	-- or--	putint (n : INTEGER),
put_real (r : REAL)	--or--	putreal (r : REAL)
put_string (s : STRING)	--or--	putstring (s : STRING),
new_line	--or--	put_new_line

-- output a linefeed, i.e. start a new line

Print

For convenience, Eiffel provides a routine, print, that can be used independently to print a string.

Also, an object, io:STD_FILES, is made available or supplied to all classes. (print = io.put_string)

In a string we can use %N (the linefeed character) to start a new line. e.g. print("%N") is the same as io.new_line

```
class
        TESTSQRT
creation
        make
feature

        make is
          local
                s  : SQRT
                x,r : REAL
                n : INTEGER
          do
                !!s
                print("Input a number :")
                io.read_real
                x := io.last_real
                print("%NBinary sqrt is : ")
                r := s.sqrt_r(x)
                io.put_real(r)
                io.put_new_line
          end -- make

end -- TESTSQRT
```

Naming Classes

In Eiffel, the name of the class is usually used to name the file the class is in but with the extension ‘.e’ added,

e.g. the class TESTSQRT is in the file “testsqrt.e”

All Eiffel class source files have the extension “.e”.

Also we have named the system “square_root” which is the same name as the .ace file, i.e. “square_root.ace” and also of the directory which contains the system.

This naming practice is a recommended convenience for Eiffel.

“testsqrt.e”

```
class TESTSQRT
  creation
        start  -- usually called ‘make’
  feature

        start is
          local
                s  : SQRT
                x,r : REAL
                n : INTEGER
          do
                !!s
                print("Input a number :")
                io.readreal
                x := io.last_real
                print("%NBinary sqrt is : ")
                r := s.sqrt_r(x)
                io.putreal(r)
                io.new_line
          end -- make
end -- TESTSQRT
```

“sqrt.e”

```
class
        SQRT
feature

sqrt_r(x:REAL):REAL is
        local
                y : REAL
        do
                from
                        y := 1
                until
                        y^2 > x
                loop
                        y := 2*y
                end
                result := bin_sqrt_r(0,y,0.0001,x)
        end -- sqrt_r
```

```
bin_sqrt_r (low,high:REAL;  eps:REAL; x:REAL):REAL is
        -- (Recursive version)
        require
                Within: low^2 <= x and x < high^2
        local
                mid:REAL
        do
                if low + eps < high then
                        mid := (low + high)/2
                        if mid^2 <= x then
                                        --mid^2 <= x and x < high^2
                                result := bin_sqrt_r(mid,high,eps,x)
                        else                    -- low^2 <= x and x < mid^2
                                result := bin_sqrt_r(low,mid,eps,x)
                        end
                else
                        result := low
                end
        ensure
                 result^2 <= x and x < (result+eps)^2
        end -- bin_sqrt_r

end --SQRT
```

Eiffel Classes

According to Bertrand Meyer, Eiffel is more a Class-Oriented language than an Object-Oriented one. An Eiffel programmer writes Classes; only classes appear in source code. Objects are generated at run or execution time.

A class in Eiffel can be regarded as a type or better, an Abstract Data Type (ADT). A class determines the properties that an instantiation of the class (i.e. an object of the class) will have at run time.

If B is a class we can write x:B in another class.

Client .v. Supplier

If in a class A we have the entity x:B, then B is a supplier to the client A. The class B makes available its properties to the entity x in A.
e.g.
Let the class POINT be the supplier class. This class may be used in some graphics class to define points in a window. A point on the plane relative to some co-ordinate system may be regarded as a 2-dim vector. In the client class we may have

```
p1, p2 : POINT
r,s : REAL
```

No objects are created at declaration time, except for the Basic classes:
INTEGER, REAL, CHARACTER and BOOLEAN.
For the non-basic classses one must create an object explicitly.
In Eiffel, this is done as

```
!!p1  -- "Bang Bang p1" or "Pling Pling p1"
```

This creates an object to which p1 refers,
i.e. p1 is a reference to a new object of type POINT.
Rather than call p1 a variable, p1 is called an entity.
Entities are, in effect, references to objects.

Since p1 is a POINT, certain routines may be performed on it. For example, a point may be translated a distance h horizontally and v vertically, or scaled by a factor s,

```
p1.translate(-3.5, 6)
p2.scale(3.0)
```

The point's co-ordinates are attributes of the class and may be accessed

```
r := p1.x  -- the x-coordiate of p1
s := p1.y -- the y coordinate of p1
```

We could also have a distance function associated with a point, that gets the distance from the current point ot some other point.

```
r := p1.distance(p2)
```

assigns the distance of p1 to p2 to r.

In general, the syntax is

entity.operation(arguments)

```
class
    POINT
feature
    x, y : REAL  -- the co-ordinates of the point

    scale (s : REAL) is    -- procedure to scale by factor s
        do
            x := s*x
            y := s*y
        end -- Scale

    translate (h, v : REAL) is
    -- move point horizontally h and vertically v
        do
            x := x + h
            y := y + v
        end -- Translate

    distance (p : POINT) : REAL is
        -- distance from current point to p
        do
            result := sqrt ((x - p.x)^2 + (y - p.y)^2)
        end -- Distance
```

```
        modulus : REAL is
                -- distance (of current) to the origin
                -- or the length of the vector/point current
                do
                        result := sqrt(x^2 + y^2)
                end – Modulus


end -- POINT
```

Any object from this class, POINT, will have the above properties.

In writing a class, keep in mind a typical object from the class. The distance function above gives the distance between p and the point created when the class is used (by a client). In a class we can refer to the typical object created as *current* and so in the above we could write 'current.x' instead of 'x'.

We could have implemented the function Modulus using the distance function:

```
                modulus : REAL is
                        -- Distance to origin
                        local
                                origin : POINT
                        do
                                !!origin
                                result := Distance (origin)
                        end -- Modulus
```

This function has the overhead or side-effect of creating a new object (the origin) each time it is called. In Eiffel, we should avoid having side-effects in functions.

Note that when an object of type POINT is created its attributes x and y will be automatically initialised to zero. We could rewrite the class POINT so as to allow a point to be initialised to any value when created. This is possible by using the facility *creation* in the class,
e.g.

```
class
        POINT
creation
                make
-- This  tells Eiffel that the creation procedure is 'make'
--- Also this procedure can be called at object creation.

feature
                make (x0, y0 : REAL) is
                        -- initialise current with co-ords x0, y0
                        do
                                x := x0
                                y := y0
                        end – make

                ...  the other routines as above
end -- POINT
```

It is usual practise to call the creation procedure 'make' but any other name is allowed.

e.g. In a client class of POINT we may have

```
                p: POINT

                ...

                !!p.make(2.0,3.0)
```

At run time a POINT object is created and initialised to default values and then the make procedure is called with args 2.0 and 3.0. After creation the POINT object will have coords 2.0 and 3.0.