Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 7
News (Reminder)

Mock exam next week!

- Attendance is **highly** recommended
- The week after we will discuss the results
Today

- Inheritance
- Genericity
Inheritance

Principle:
Describe a new class as extension or specialization of an existing class
(or several with *multiple* inheritance)

If $B$ inherits from $A$:

- **As modules:** all the services of $A$ are available in $B$
  (possibly with a different implementation)

- **As types:** whenever an instance of $A$ is required, an instance of $B$ will be acceptable
  ("is-a" relationship)
Let's play Lego!
Class *BRICK*

defered class *BRICK*

feature
  width: INTEGER
  depth: INTEGER
  height: INTEGER
  color: COLOR

  volume: INTEGER
    deferred
    end
end
defered end
defered end
Class **LEGO_BRICK**

Inherit all features of class **BRICK**.

New feature, number of nubs

Implementation of volume.

class **LEGO_BRICK**

inherit **BRICK**

feature

number_of_nubs: INTEGER

volume: INTEGER

do

Result := ...

end

done
The feature \textit{volume} is going to be redefined (=changed). The feature \textit{volume} comes from \textit{LEGO_BRICK}.

\textbf{Class LEGO\_BRICK\_SLANTED}

\begin{verbatim}
class LEGO\_BRICK\_SLANTED
    inherit LEGO\_BRICK
    redefine volume
    end

feature
    volume: INTEGER
        do
            Result := ...
        end
end
\end{verbatim}
Class LEGO_BRICK_WITH_HOLE

class LEGO_BRICK_WITH_HOLE

inherit LEGO_BRICK

redefine volume

end

feature

volume: INTEGER

do

Result := ...

done

end

end

The feature volume is going to be redefined (=changed). The feature volume comes from LEGO_BRICK
Inheritance Notation

Notation:
- Deferred *
- Effective +
- Redefinition ++

Diagram:
- BRICK
  - LEGO_BRICK
    - LEGO_BRICK_WITH_HOLE
    - LEGO_BRICK_SLANTED
Deferred

- Deferred classes can have deferred features.
- A class with at least one deferred feature must be declared as deferred.
- A deferred feature does not have an implementation yet.
- Deferred classes cannot be instantiated and hence cannot contain a create clause.

Can we have a deferred class with no deferred features?
Effective

- Effective classes do not have deferred features (the “standard case”).
- Effective routines have an implementation of their feature body.
If a feature was redefined, but you still wish to call the old one, use the **Precursor** keyword.

```plaintext
volume: INTEGER
  do
    Result := Precursor - ...
  end
```
A more general example of using Precursor

-- Class A
routine (a_arg1 : TYPE_A): TYPE_R
do ... end

-- Class C
routine (a_arg1 : TYPE_A): TYPE_R
local
l_loc : TYPE_R
do
  -- pre-process
  l_loc := Precursor {B} (a_arg1 )
  -- Not allowed: l_loc := Precursor {A} (a_arg1 )
  -- post-process
end
Today

- Inheritance
- Genericity
Assume we want to create a list class capable of storing objects of any type.

```plaintext
class
  LIST -- First attempt
end

feature

  put: (a_item: ANY)
    do
      -- Add item to the list
    end

  item: ANY
    do
      -- Return the first item in the list
    end

-- More feature for working with the list
```

We could choose ANY as the item type
Working with this list – first attempt

insert_strings (a_list_of_strings: LIST)
do
    a_list_of_strings.put("foo")
    a_list_of_strings.put(12);
    a_list_of_strings.put("foo")
end

print_strings (a_list_of_strings: LIST)
local
    l_printme: STRING
do
across a_list_of_strings as l loop
    l_printme := l.item
    io.put_string (l_printme)
end
end

Here we are inserting an INTEGER

Compile error: cannot assign ANY to STRING
Working with this list – the right way

```plaintext
insert_strings (a_list_of_strings: LIST)
    do
        a_list_of_strings.put("foo")
        a_list_of_strings.put(12);
        a_list_of_strings.put("foo")
    end

print_strings (a_list_of_strings: LIST)
    local
        l_current_item: ANY
    do
        across a_list_of_strings as l loop
            l_current_item := l.item
            if attached {STRING} l_current_item as itemstring then
                io.put_string (itemstring)
            else
                io.put_string ("The list contains a non-string item!")
            end
        end
    end
end
```

Still nobody detects this problem

This solution works, but wouldn't it be nice to detect this mistake at compile time?

Correct. This syntactical construct is called 'object test'.
Genericity

Genericity lets you parameterize a class. The parameters are types. A single class text may be reused for many different types.
A generic list

Class `LIST` feature

- `extend (x : G)` ...
- `last : G` ...

End

To use the class: obtain a **generic derivation**, e.g.

`cities : LIST [ CITY ]`
A generic list with constraints

class
    STORAGE [G] -> RESOURCE

inherit
    LIST [G]

feature
    consume_all
    do
        from start until after
        loop
            item.consume
            forth
        end
    end

end

The feature item is of type G. We cannot assume consume.

RESOURCE. We can assume this.
Type-safe containers

- Using genericity you can provide an implementation of type safe containers.

```plaintext
x: ANIMAL
animal_list: LINKED_LIST [ANIMAL]
a_rock: MINERAL

animal_list.put (a_rock) -- Does this rock?
```

Compile error!
Definition: Type

We use types to declare entities, as in

\[ x : \text{SOME\_TYPE} \]

With the mechanisms defined so far, a type is one of:

- A non-generic class  
  e.g.  \text{METRO\_STATION}

- A \textit{generic derivation}, i.e. the name of a class followed by a list of \textit{types}, the actual generic parameters, in brackets (also recursive)
  e.g.  \text{LIST[ARRAY[METRO\_STATION]]}
  \text{LIST[LIST[CITY]]}
  \text{TABLE[STRING, INTEGER]}
So, how many types can I possibly get?

Two answers, depending on what we are talking about:

- **Static types**
  
  Static types are the types that we use while writing Eiffel code to declare types for entities (arguments, locals, return values)

- **Dynamic types**
  
  Dynamic types on the other hand are created at runtime. Whenever a new object is created, it gets assigned to be of some type.
class EMPLOYEE
  feature
    name: STRING
    birthday: DATE
  end
end

class DEPARTMENT
  feature
    staff: LIST[EMPLOYEE]
  end
end

bound by the program text:
EMPLOYEE
STRING
DATE
DEPARTMENT
LIST[G]
becomes LIST[EMPLOYEE]
Object creation, static and dynamic types

class TEST_DYNAMIC_CREATION
feature
    ref_a: A; ref_b: B
    -- Suppose B, with creation feature make_b,
    -- inherits from A, with creation feature make_a

    do_something
        do
            create ref_a.make_a
            -- Static and dynamic type is A

            create {B} ref_a.make_b
            -- Static type is A, dynamic type is B

            create ref_b.make_b
            ref_a := ref_b
        end

end
Dynamic types: another example

class $SET[G]$ feature
  powerset: $SET[SET[G]]$ is
    do
      create Result
      -- More computation...
    end

$i\_th\_power\ (i: INTEGER): SET[ANY]\nrequire \ i \ >= \ 0$
local \ n: INTEGER
  do
    Result := Current
    from \ n := 1 \ until \ n > i \ loop
      Result := Result.powerset
      n := n + 1
    end
  end
end

Dynamic types from $i\_th\_power$:

$SET[ANY]$

$SET[SET[ANY]]$

$SET[SET[SET[ANY]]]$

...

From http://www.eiffelroom.com/article/fun_with_generics