Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 7
Mock exam next week!

- Monday exercise groups: November 7
- Tuesday exercise groups: November 8
- You have to be present
- The week after we will discuss the results
- Assignment 7 due on November 15
Today

- Inheritance
- Genericity
- Exercise: practicing contracts
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
defered
defered
end
don't include
Inherit all features of class BRICK.

New feature, calculate all nubs

Implementation of volume.
The feature `volume` is going to be redefined (=changed). The feature `volume` comes from `LEGO_BRICK`.

```
class
  LEGO_BRICK_SLANTED
inhibit
  LEGO_BRICK
redefine
  volume
end

feature
  volume: INTEGER
do
    Result := ...
end
end
```
Class **LEGO_BRICK_WITH_HOLE**

```plaintext
class LEGO_BRICK_WITH_HOLE
   inherit LEGO_BRICK
   redefine volume
end

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

```
Inheritance Notation

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

BRICK

volume*

LEGO_BRICK

volume+

LEGO_BRICK_WITH_HOLE

volume++

LEGO_BRICK_SLANTED

volume++
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.
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

\[
\text{routine } (a_{-arg1} : \text{TYPE}_A): \text{TYPE}_R \\
do \ldots \text{end}
\]

-- Class C

\[
\text{routine } (a_{-arg1} : \text{TYPE}_A): \text{TYPE}_R \\
\text{local} \\
\quad _\text{loc} : \text{TYPE}_R \\
do \\
\quad \text{-- pre-process} \\
\quad _\text{loc} := \textbf{Precursor } \{B\} (a_{-arg1}) \\
\text{-- Not allowed: } _\text{loc} := \textbf{Precursor } \{A\} (a_{-arg1}) \\
\quad \text{-- post-process} \\
\text{end}
\]
Today

- Inheritance
- Genericity
- Exercise: practicing contracts
Genericity lets you parameterize a class. The parameters are types. A single class text may be reused for many different types.
**Genericity**

Inheritance

Abstraction

Type parameterization

Genericity

Specialization

**Type parameterization**

- `LIST_OF_CARS`
- `LIST_OF_CITIES`
- `LINKED_LIST_OF_CARS`
- `SET_OF_CARS`
- `LIST_OF_PERSONS`
A generic list

class LIST [G] 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

The feature item is of type G. We cannot assume consume.
Type-safe containers

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

\[ x: \text{ANIMAL} \]
\[ \text{animal\_list: LINKED\_LIST [ANIMAL]} \]
\[ \text{a\_rock: MINERAL} \]

\[ \text{animal\_list.put (a\_rock) -- Does this rock?} \]
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 **generic derivation**, i.e. the name of a class followed by a list of **types**, the actual generic parameters, in brackets (also recursive)  
  e.g. \( \text{LIST[ARRAY[\text{METRO\_STATION}]]} \), \( \text{LIST[LIST[CITY]]} \), \( \text{TABLE[STRING, INTEGER]} \)
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.
Static types

class EMPLOYEE

feature

    name: STRING
    birthday: DATE

end

class DEPARTMENT

feature

    staff: LIST[EMPLOYEE]

end

bound by the program text:

EMPLOYEE
STRING
DATE
DEPARTMENT
LIST[G]

becomes LIST[EMPLOYEE]
Object creation, static and dynamic types

```plaintext
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
        end
end
```
Dynamic types: another example

class \textit{SET}[G] \ feature
\hspace{1cm} \textit{powerset}: \ \textit{SET}[\textit{SET}[G]] \ \textit{is}
\hspace{1cm} \textbf{do}
\hspace{1.5cm} \textbf{create Result}
\hspace{2cm} \text{-- More computation...}
\hspace{1cm} \textbf{end}
\hspace{1cm} \textbf{end}

\textit{i\_th\_power} (\textit{i}: \textit{INTEGER}): \ \textit{SET}[\textit{ANY}]
\hspace{1cm} \textbf{require} \ \textit{i} \geq 0
\hspace{1cm} \textbf{local} \ \textit{n}: \textit{INTEGER}
\hspace{1cm} \textbf{do}
\hspace{1.5cm} \textit{Result} := \textit{Current}
\hspace{1.5cm} \textbf{from} \ \textit{n} := 1 \ \textbf{until} \ \textit{n} > \textit{i} \ \textbf{loop}
\hspace{2cm} \textit{Result} := \textit{Result}.\textit{powerset}
\hspace{2.5cm} \textit{n} := \textit{n} + 1
\hspace{1.5cm} \textbf{end}
\hspace{1cm} \textbf{end}
\hspace{1cm} \textbf{end}

From http://www.eiffelroom.com/article/fun_with Generics
Today

- Inheritance
- Genericity
- Exercise: practicing contracts
Specification of a card game

A deck is initially made of 36 cards

Every card in the deck represents a value in the range 2..10

Every card also represents 1 out of 4 possible colors

The colors represented in the game cards are: red ('R'), white ('W'), green ('G') and blue ('B')

As long as there are cards in the deck, the players can look at the top card and remove it from the deck
Class CARD create make

make (a_color: CHARACTER, a_value: INTEGER)
    -- Create a card given a color and a value.

require
    ...

ensure
    ...

color: CHARACTER
    -- The card color.

value: INTEGER
    -- The card value.
Class CARD: which colors are valid?

is_valid_color (a_color: CHARACTER): BOOLEAN

-- Is `a_color' a valid color?

require

... 

ensure

...
is_valid_range (n: INTEGER): BOOLEAN

    -- Is `n` in the acceptable range?

    require

    ...  

    ensure

    ...

    invariant

    ...

Class CARD: which ranges are valid?
Class CARD create make: reloaded

make (a_color: CHARACTER, a_value: INTEGER)
    -- Create a card given a color and a value.

require
    ...

ensure
    ...

color: CHARACTER
    -- The card color.

value: INTEGER
    -- The card value.
Class DECK create make

make

-- Create a deck with random cards.

require

...

ensure

...

feature {NONE} -- Implementation

card_list: LINKED_LIST [CARD]

-- Deck as a linked list of cards.
Class DECK queries

- **top_card**: \texttt{CARD}
  -- The deck's top card.

- **is_empty**: \texttt{BOOLEAN}
  -- Is Current deck empty?
  \begin{verbatim}
  do ...
  end
  \end{verbatim}

- **count**: \texttt{INTEGER}
  -- Number of remaining cards in the deck.
  \begin{verbatim}
  do ...
  end
  \end{verbatim}
Removing the top card from DECK

remove_top_card

-- Remove the top card from the deck.

require

...

ensure

...
The class invariant

invariant

...