Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 5
Today

- Attributes, formal arguments, and local variables
- Control structures
Attributes

Declared anywhere inside a feature clause, but outside other features

```plaintext
class C
feature
    attr1 : CA1
    f (arg1 : A ...) do
        ...
    end
...
end
```

Visible anywhere inside the class
visible outside the class (depending on their visibility)
Formal arguments

Declared after the feature name, in parenthesis:

\[
\text{feature } \quad \text{f (arg1 : C1; \ldots; argn : CN)}
\]

require ...
local ...
do ...
ensure ...
end

only visible inside the feature body and its contracts
Some variables are only used by a certain routine. Declare them as local:

```plaintext
feature
  f (arg1: A ...)
  require ...
  local x, y: B
  z: C
  do ...
  ensure ...
end
```

only visible inside the feature body
Summary: the scope of names

Attributes:
- declared anywhere inside a feature clause, but outside other features
- visible anywhere inside the class
- visible outside the class (depending on their visibility)

Formal arguments:
- declared after the feature name, in parenthesis
- only visible inside the feature body and its contracts

Local variables:
- declared in a local clause inside the feature declaration
- only visible inside the feature body
class PERSON
feature
  name : STRING

  set_name (a_name : STRING)
    do
      name := a_name
    end

  exchange_names (other : PERSON)
    local
      s : STRING
    do
      s := other.name
      other.set_name (name)
      set_name (s)
    end

  print_with_semicolon
    do
      create s.make_from_string (name)
      s.append (";")
      print (s)
    end
end

Error: this variable was not declared
class PERSON
feature
    -- name and set_name as before
    exchange_names (other : PERSON)
    local
        s : STRING
    do
        s := other.name
        other.set_name (name)
        set_name (s)
    end

print_with_semicolon
local
    s : STRING
    do
        create s.make_from_string (name)
        s.append (";"
        print (s)
    end
end
An example of side effects

```ruby
class PERSON
  feature
  ...
  name : STRING
  
  print_with_semicolon
    local
      s : STRING
      do
        create s.make_from_string(name)
        s.append(";")
        print(s)
      end
  end

  print_with_sticky_semicolon
    do
      name.append(";")
      print(name)
    end
end
```

Now the semicolon sticks to the attribute.
This is called side effect
Compilation error? (3)

```plaintext
class PERSON
feature
    ... -- name and set_name as before
    s: STRING

exchange_names (other: PERSON)
    do
        s := other.name
        other.set_name (name)
        set_name (s)
    end

s: STRING

print_with_semicolon
    do
        create s.make_from_string (name)
        s.append (";"
        print (s)
    end
end
```

Error: an attribute with the same name was already defined
class PERSON
feature
…    -- name and set_name as before
exchange_names (other: PERSON)
do
    s := other.name
    other.set_name (name)
    set_name (s)
end

print_with_semicolon
do
    create s.make_from_string (name)
    s.append (';')
    print (s)
end

s: STRING
end

OK: a single attribute used in both routines
Local variables vs. attributes

- Which one of the two correct versions (2 and 4) do you like more? Why?

- Describe the conditions under which it is better to use a local variable instead of an attribute and vice versa
Inside every function you can use the predefined local variable **Result** (you needn’t and shouldn’t declare it)

The return value of a function is whatever value the **Result** variable has at the end of the function execution

At the beginning of routine’s body **Result** (as well as regular local variables) is initialized with the default value of its type

Every regular local variable is declared with some type; and what is the type of **Result**?

It’s the function return type!
class PERSON
feature
    ... -- name and set_name as before
    exchange_names (other : PERSON)
        do
            Result := other.name
            other.set_name (name)
            set_name (Result)
        end

    name_with_semicolon : STRING
        do
            create Result.make_from_string (name)
            Result.append (';')
            print (Result)
        end
end
Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

\[ y := 5 \]
\[ x.y := 5 \]
\[ \text{Current}.y := 5 \]

There are two main reasons for this rule:

1. A client may not be aware of the restrictions on the attribute value and interdependencies with other attributes => class invariant violation (Example?)
2. Guess! (Hint: uniform access principle)
An entity in program text is a “name” that directly denotes an object. More precisely: it is one of

- attribute name
- variable attribute
- constant attribute
- formal argument name
- local variable name
- Result
- Current

Only a variable can be used in a creation instruction and in the left part of an assignment
class VECTOR
feature
  x, y : REAL

  copy_from (other : VECTOR)
  do
    Current := other
  end

  copy_to (other : VECTOR)
  do
    create other
    other.x := x
    other.y := y
  end

  reset
  do
    create Current
  end
end

Current is not a variable and can not be assigned to
other is a formal argument (not a variable) and thus can not be used in creation
other.x is a qualified attribute call (not a variable) and thus can not be assigned to
the same reason for other.y
Current is not a variable and thus can not be used in creation
Structured programming

- In structured programming instructions can be combined only in three ways (constructs):
  - **Sequential composition**
    - Compound
    - *s₁* → *s₂*
  - **Conditional**
    - Condition
    - *c* → True
    - True → *s₁* → False
    - False → *s₂*
  - **Loop**
    - *c* → True
    - True → *s*
    - False → *c*

- Each of these blocks has a single entry and exit and is itself a (possibly empty) compound
### Conditional

- **Basic syntax:**
  
  ```
  if c then
    s_1
  else
    s_2
  end
  ```

- **Could** `c` **be an integral expressions?**
  
  - No. `c` is a boolean expression (e.g., entity, query call of type `BOOLEAN`)

- **Are these valid conditionals?**
  
  - Yes, `else` is optional
  - Yes, `s_1` could be empty.
  - Yes, `s_1` and `s_2` could be both empty.
Calculating function’s value

\[ f(max: INTEGER; s: STRING): STRING \]

\[
\begin{align*}
do &
\text{if } s.is\_equal(“Java”) \text{ then} \\
& \quad \text{Result} := “J**a”
\text{else} \\
& \quad \text{if } s.count > max \text{ then} \\
& \quad \quad \text{Result} := “<an unreadable German word>”
\end{align*}
\]

Calculate the value of:

- \( f(3, “Java”) \rightarrow “J**a” \)
- \( f(20, “Immatrifikationsbestätigung”) \rightarrow “<an unreadable German word>” \)
- \( f(6, “Eiffel”) \rightarrow \text{Void} \)
Write a routine...

... that computes the maximum of two integers

\[ \text{max}(a, b : \text{INTEGER}) : \text{INTEGER} \]

... that increases time by one second inside class TIME

class TIME

    hour, minute, second : INTEGER

    second_forth
        do ... end
    ...
end
Comb-like conditional

If there are more than two alternatives, you can use the syntax:

```plaintext
if c_1 then
  s_1
elseif c_2 then
  s_2
...
elif c_n then
  s_n
else
  s_e
end
```

instead of:

```plaintext
if c_1 then
  s_1
else
  if c_2 then
    s_2
  else
    ...
  if c_n then
    s_n
  else
    s_e
end
...
end
```
Multiple choice

If all the conditions have a specific structure, you can use the syntax:

inspect expression
when const_1 then
  s_1
when const_2 then
  s_2
...
when const_n1 .. const_n2 then
  s_n
else
  s_e
end
Lost in conditions

Rewrite the following multiple choice:

- using a comb-like conditional
- using nested conditionals

inspect user_choice
when 0 then
  print ("Hamburger")
when 1 then
  print ("Coke")
else
  print ("Not on the menu!")
end

if user_choice = 0 then
  print ("Hamburger")
elseif user_choice = 1 then
  print ("Coke")
else
  print ("Not on the menu!")
end

if user_choice = 0 then
  print ("Hamburger")
else
  if user_choice = 1 then
    print ("Coke")
  else
    print ("Not on the menu!")
  end
end
Loop: Basic form

Syntax:

```
from initialization until exit_condition loop body end
```

- `from` followed by initialization
- `until` followed by exit condition
- `loop` followed by body
- `end`
Compilation error? Runtime error?

\[
f(x, y: \text{INTEGER}): \text{INTEGER}
\]
\[
do
\from i := 1 \until (\text{True})
\loop
\begingroup
i := i \times x \times y
\endgroup
\end
\end
Simple loop

How many times will the body of the following loop be executed?

\[
i : \text{INTEGER} \\
\ldots \\
\text{from} \\
i := 1 \\
\text{until} \\
i > 10 \\
\text{loop} \\
\text{print ("I will not say bad things about assistants");} \\
i := i + 1 \\
\text{end} \\
\ldots \\
\text{from} \\
i := 10 \\
\text{until} \\
i < 1 \\
\text{loop} \\
\text{print ("I will not say bad things about assistants");} \\
\text{end}
\]

In Eiffel we usually start counting from 1

Caution! Loops can be infinite!
What does this function do?

factorial (n: INTEGER): INTEGER
require
  n >= 0
local
  i: INTEGER
do
d from
  i := 2
  Result := 1
until
  i > n
loop
  Result := Result * i
  i := i + 1
end
end
Loop: More general form

Syntax:

from

initialization

invariant

inv

until

exit_condition

loop

body

variant

var

end

Compound

Optional

Boolean expression

Boolean expression

Compound

Optional

Integer expression
Invariant and variant

Loop invariant (do not confuse with class invariant)
- holds before and after the execution of loop body
- captures how the loop iteratively solves the problem: e.g. “to calculate the sum of all \( n \) elements in a list, on each iteration \( i (i = 1..n) \) the sum of first \( i \) elements is obtained”

Loop variant
- integer expression that is nonnegative after execution of from clause and after each execution of loop clause and strictly decreases with each iteration
- a loop with a correct variant can not be infinite (why?)
What are the invariant and variant of the “factorial” loop?

**from**

\[ i := 2 \]
\[ \text{Result} := 1 \]

**invariant**

\[ \text{Result} = \text{factorial}(i - 1) \]

**until**

\[ i > n \]

**loop**

\[ \text{Result} := \text{Result} \times i \]
\[ i := i + 1 \]

**variant**

\[ n - i + 2 \]

**end**
Implement a function that calculates Fibonacci numbers, using a loop

```
fibonacci (n : INTEGER) : INTEGER
    -- n-th Fibonacci number
    require
        n_non_negative : n >= 0
    ensure
        first_is_zero : n = 0 implies Result = 0
        second_is_one : n = 1 implies Result = 1
        other_correct : n > 1 implies Result = fibonacci (n - 1) + fibonacci (n - 2)
    end
```
Writing loops (solution)

\[
\text{fibonacci}(n : \text{INTEGER}) : \text{INTEGER} \\
\text{local} \\
\quad a, b, i : \text{INTEGER} \\
\text{do} \\
\quad \text{if } n \leq 1 \text{ then} \\
\quad \quad \text{Result} := n \\
\quad \text{else} \\
\quad \quad \text{from} \\
\quad \quad \quad a := 0 \\
\quad \quad \quad b := 1 \\
\quad \quad \quad i := 1 \\
\quad \quad \text{invariant} \\
\quad \quad \quad a = \text{fibonacci}(i - 1) \\
\quad \quad \quad b = \text{fibonacci}(i) \\
\quad \quad \text{until} \\
\quad \quad \quad i = n \\
\quad \text{loop} \\
\quad \quad \quad \text{Result} := a + b \\
\quad \quad \quad a := b \\
\quad \quad \quad b := \text{Result} \\
\quad \quad \quad i := i + 1 \\
\quad \text{variant} \\
\quad \quad \quad n - i \\
\quad \text{end} \\
\text{end} \\
\text{end}
\]
Summary

- Attributes, formal arguments, and local variables
  - Scope
- Control structures