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 inside a feature clause, but outside other features

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:

```
feature
    f (arg1 : C1 ; ... ; argn : CN)
        require ...
        local ...
        do ...
        ensure ...
    end
```

Visible only inside the feature body and its contracts
Local variables

Some variables are only used by one routine. Declare them as local:

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

Visible only inside the feature body
```
Summary: The scope of names

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

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

Local variables:
- declared in a local clause inside the feature
- visible only 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

```eiffel
class PERSON

feature

... name : STRING

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

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

end
```

Now the semicolon sticks to the attribute. This is called side effect.

Remember that strings in Eiffel are mutable by default!
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
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?

When is it better to use a local variable instead of an attribute (and vice versa)?
You can use the predefined local variable `Result` inside (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 a routine’s body, `Result` (and the local variables) is initialized with the default value of its type

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

It’s the function’s return type!
Compilation error? (5)

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
In object-oriented computation each routine call is performed on a certain object.
From inside a routine we can access this object using the predefined entity \textit{Current}.

What is the type of \textit{Current}?
Revisiting qualified vs. unqualified feature calls

- If the target of a feature call is \textit{Current}, it is omitted:
  \[
  \text{Current}.f(a) \\
  f(a)
  \]

- Such a call is \textit{unqualified}

- Otherwise, if the target of a call is specified explicitly, the call is \textit{qualified}
  \[
  x.f(a)
  \]
Qualified or unqualified?

Are the following feature calls, with their feature names underlined, qualified or unqualified? What are the targets of these calls?

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1) $x.y$</td>
<td>qualified</td>
</tr>
<tr>
<td>2) $x$</td>
<td>unqualified</td>
</tr>
<tr>
<td>3) $f(x,a)$</td>
<td>unqualified</td>
</tr>
<tr>
<td>4) $x.y.z$</td>
<td>qualified</td>
</tr>
<tr>
<td>5) $x(y.f(a,b))$</td>
<td>unqualified</td>
</tr>
<tr>
<td>6) $f(x.a).y(b)$</td>
<td>qualified</td>
</tr>
<tr>
<td>7) <strong>Current.$x$</strong></td>
<td>qualified</td>
</tr>
</tbody>
</table>
Assignment to attributes

Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way:

\[ y := 5 \]  \hspace{1cm} \text{OK}

\[ x.y := 5 \]  \hspace{1cm} \text{Error}

\[ \text{Current}.y := 5 \]  \hspace{1cm} \text{Error}

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 \( \Rightarrow \) class invariant violation (Example?)

2. Guess! (Hint: uniform access principle)
It is possible to declare constant attributes, that is, attributes having a fixed value that cannot change during the program execution.

```plaintext
class CAR
feature
  ...
  ...
  number_of_gears: INTEGER = 5
  ...
  set_number_of_gears (new_number: INTEGER)
    do
      number_of_gears := new_number
    end
end
```

Constant attribute

Error: constant attributes are readonly
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
end

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 cannot be assigned to
other is a formal argument (not a variable) and thus cannot be used in creation
other.x is a qualified attribute call (not a variable) and thus cannot be assigned to
the same reason for other.y
Current is not a variable and thus cannot be used in creation
In structured programming instructions can be combined only in three ways (constructs):

- Sequential composition
- Conditional
- Loop

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

- **Basic syntax:**
  
  $\text{if } c \text{ then } s_1 \text{ end}$

  $\text{else } s_2 \text{ end}$

- **Could $c$ be an integer expression?**
  - 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

\[
f(\text{max} : \text{INTEGER} ; s : \text{STRING}) : \text{STRING}
\]
do
  if \(s\.is\_equal(\text{"Java"})\) then
    Result := \text{"J**a"}
  else
    if \(s\.count > \text{max}\) then
      Result := \text{"<an unreadable German word>"}
    end
  end
end

Calculate the value of:

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

- ... that computes the maximum of two integers

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

- ... that increases the time by one second inside class \textit{TIME}

```
class \textit{TIME}

  \textit{hour, minute, second} : \text{INTEGER}

  \textit{second} \_\textit{forth}

  \texttt{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
  \( s_2 \)
  ...
elseif \( c_n \) 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
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

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

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

Syntax:

```
from initialization

until exit_condition

loop

body

end
```

- `from` followed by an `initialization` which can be a compound expression.
- `until` followed by an `exit_condition` which is a Boolean expression.
- `loop` followed by a `body` which can be a compound expression.
- `end` to close the loop.
Compilation error? Runtime error?

$$f(x, y : \text{INTEGER}) : \text{INTEGER}$$

```plaintext
f(x, y : INTEGER): INTEGER
  do
  from
  until \((x \div y)\)
  loop
    "Print me!"
  end
end
```

**Compilation error:**
- integer expression instead of boolean
- expression instead of instruction

```
f(x, y : INTEGER): INTEGER
  local
    i : INTEGER
  do
    from \(i := 1\) until (True)
    loop
      \(i := i \times x \times y\)
    end
  end
end
```

**Correct, but non-terminating**

```
f(x, y : INTEGER): INTEGER
  do
    from
    until False
    loop
    end
  end
```

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

```eiffel
i : INTEGER
...
from i := 1
until i > 10
loop
  print ("I will not say bad things about assistants")
  i := i + 1
end
...
from i := 10
until i < 1
loop
  print ("I will not say bad things about assistants")
end
```

Caution! Loops can be infinite!
Loop: More general form

Syntax:

```
from    
initialization    

invariant
inv

until
exit_condition

loop
body

variant
var

end
```

- `from` (Compound)
- `initialization` (Optional)
- `invariant` (Optional)
- `inv` (Optional)
- `until` (Optional)
- `exit_condition` (Optional)
- `loop` (Compound)
- `body` (Compound)
- `variant` (Optional)
- `var` (Optional)
- `end` (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?)
Example – sum of the first n integers

\textit{sum (n: INTEGER): INTEGER}

-- Compute the sum of the numbers from 0 to `n`

\textbf{require} \hspace{1em} 0 <= n

\textbf{local} \hspace{1em} i: INTEGER

\textbf{do}

\textbf{from}

\hspace{1em} Result := 0\hspace{1em} i := 1

\textbf{invariant}

\hspace{1em} 1 <= i \textit{ and } i <= n+1

\hspace{1em} Result = (i * (i - 1)) \hspace{1em} // \hspace{1em} 2

\textbf{until}

\hspace{1em} i > n

\textbf{loop}

\hspace{1em} Result := Result + i\hspace{1em} i := i + 1

\textbf{variant}

\hspace{1em} n - i + 1

\textbf{end}

\textbf{ensure}

\hspace{1em} Result = (n * (n + 1)) \hspace{1em} // \hspace{1em} 2

\textbf{end}
What does this function do?

```plaintext
factorial (n : INTEGER) : INTEGER
require
  n >= 0
local
  i : INTEGER
do
  from
    i := 2
  Result := 1
until
  i > n
loop
  Result := Result * i
  i := i + 1
end
end
```
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

Result = 6 = 3!
Writing loops

Implement a function that calculates Fibonacci numbers, using a loop

\[
\text{fibonacci} \ (n : \text{INTEGER}) : \text{INTEGER} \\
\quad \quad \quad \text{-- n-th Fibonacci number}
\]

\[
\text{require} \\
\quad \quad n_\text{non_negative} : n \geq 0
\]

\[
\text{ensure} \\
\quad \quad \text{first_is_zero} : n = 0 \text{ implies Result = 0} \\
\quad \quad \text{second_is_one} : n = 1 \text{ implies Result = 1} \\
\quad \quad \text{other_correct} : n > 1 \text{ implies Result = } \text{fibonacci} \ (n - 1) + \\
\quad \quad \quad \text{fibonacci} \ (n - 2)
\]

end
Writing loops (solution)

\[
\text{fibonacci}(n : \text{INTEGER}) : \text{INTEGER}
\]

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

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