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

```
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
```

only visible inside the feature body and its contracts
Local variables

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

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

OK: two different local variables in two routines
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 (";"
      end
      print (s)
      end

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

end
```

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

Remember that strings in Eiffel are mutable!
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
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
end

s : STRING
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
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 **Current**, it is omitted:
  \[ \text{Current}.f(a) \]
  \[ f(a) \]

- Such a call is **unqualified**

- Otherwise, if the target of a call is specified explicitly, the call is **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?

1) \( x.y \)  
   qualified

2) \( x \)  
   unqualified

3) \( f(x.a) \)  
   unqualified

4) \( x.y.z \)  
   qualified

5) \( x(y.f(a.b)) \)  
   unqualified

6) \( f(x.a).y(b) \)  
   qualified

7) Current.\( x \)  
   qualified
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 => class invariant violation (Example?)
  2. Guess! (Hint: uniform access principle)
Constant attributes

- 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
```

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.
Find 5 errors

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):

  1. **sequential composition**
     - $s_1 \rightarrow s_2$
  2. **conditional**
     - $\begin{cases} s_1 & \text{if } c = \text{True} \\ s_2 & \text{if } c = \text{False} \end{cases}$
  3. **loop**
     - $\begin{cases} s_1 & \text{if } c = \text{True} \\ s_2 & \text{if } c = \text{False} \end{cases}$

- 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 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

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

\[
\text{do}
\]

\[
\text{if } s.\text{is_equal}(\text{"Java"}) \text{ then}
\]

\[
\text{Result} := \text{"J**a"}
\]

\[
\text{else}
\]

\[
\text{if } s.\text{count} > \text{max} \text{ then}
\]

\[
\text{Result} := \text{"<an unreadable German word>"}
\]

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{end}
\]

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 time by one second inside class \text{TIME}

\begin{verbatim}
class \text{TIME}
    hour, minute, second : INTEGER

    second_forth
        do ... end

    ...

end
\end{verbatim}
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
    ...
  else
    ...
  else
    ...
else
  ...
else
  s_e
end
end
```
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

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

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

```plaintext
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 the initialization
- **until** followed by the exit condition
- **loop**
- **body**
- **end**
Compilation error? Runtime error?

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

Compilation error: integer expression instead of boolean

```
Correct
```

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

Correct, but non-terminating

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

```
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
```

In Eiffel we usually start counting from 1

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 inv**: Boolean expression
- **until exit_condition**: Boolean expression
- **loop**: Compound
- **body**: Optional
- **variant var**: Integer expression
- **end**: None
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

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

\[
\begin{aligned}
\quad & \text{-- Compute the sum of the numbers from 0 to \`n\'} \\
\quad & \text{require} \\
\quad & \quad 0 \leq n \\
\quad & \text{do} \\
\quad & \quad \text{from} \\
\quad & \quad \quad \text{Result} := 0 \\
\quad & \quad \quad i := 1 \\
\quad & \quad \text{invariant} \\
\quad & \quad \quad 1 \leq i \text{ and } i \leq n+1 \\
\quad & \quad \quad \text{Result} = (i \times (i - 1)) \div 2 \\
\quad & \quad \text{until} \\
\quad & \quad \quad i > n \\
\quad & \quad \text{loop} \\
\quad & \quad \quad \text{Result} := \text{Result} + i \\
\quad & \quad \quad i := i + 1 \\
\quad & \quad \text{variant} \\
\quad & \quad \quad n - i + 1 \\
\quad & \quad \text{end} \\
\quad & \text{ensure} \\
\quad & \quad \quad \text{Result} = (n \times (n + 1)) \div 2 \\
\quad & \text{end}
\end{aligned}
\]

What are the loop invariants and variants here?
What does this function do?

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

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

until

\[ i > n \]

loop

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

variant

\[ n - i + 2 \]

end

\[ \text{Result} = 6 = 3! \]
Implement a function that calculates Fibonacci numbers, using a loop

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

\[
\text{require} \\
\quad n_{\text{non-negative}} : n \geq 0
\]

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

end
Writing loops (solution)

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

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

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