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:

```
feature
  f (arg1: A ...) is ...
  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
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

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

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
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
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 \texttt{Current}.

What is the type of \texttt{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}
  \[
  x. f(a)
  \]
Are the following feature calls, with their feature names underlined, qualified or unqualified? What are the targets of these calls?

1) _x_.y
2) _x_
3) _f_ (x.a)
4) _x_.y.z
5) _x_ (y.f (a.b))
6) f (x.a).y (b)
7) **Current**.x

Qualified or unqualified?

Qualified

Qualified

Unqualified

Unqualified

Qualified

Qualified

Qualified
Assignment to attributes

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

  - \( y := 5 \) \( \text{OK} \)
  - \( x.y := 5 \) \( \text{Error} \)
  - \( \text{Current.y} := 5 \) \( \text{Error} \)

- 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)
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.
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
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:
  
  \[
  \text{if } c \text{ then } s_1 \text{ else } s_2 \text{ end}
  \]

- Could \( c \) be an integral expression?

  No. \( c \) is a boolean expression (e.g., entity, query call of type \text{BOOLEAN} \)

- Are these valid conditionals?

  Yes, \text{else} is optional
  
  Yes, \( s_1 \) could be empty.
  
  Yes, \( s_1 \) and \( s_2 \) could be both empty.
Calculating function’s value

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

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

Calculate the value of:

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

```plaintext
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
... 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
```
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”)
else if user_choice = 1 then
    print (“Coke”)
else
    print (“Not on the menu!”)
end

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` which is a `Compound`
- `until` followed by `exit_condition` which is a `Boolean expression`
- `loop` followed by `body` which is a `Compound`
 Compilation error? Runtime error?

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

\[
\begin{align*}
\text{local} \\
i : \text{INTEGER} \\
\text{do} \\
\text{from } i := 1 \text{ until (True)} \\
\text{loop} \\
i := i \times x \times y \\
\text{end}
\end{align*}
\]

Correct

Hands-On

Correct, but non-terminating
How many times will the body of the following loop be executed?

\[ i : \text{INTEGER} \]

\[ \text{... from } i := 1 \text{ until } i > 10 \text{ loop} \]

\[ \text{print ("I will not say bad things about assistants"}) \]

\[ i := i + 1 \]

\[ \text{end} \]

\[ \text{... from } i := 10 \text{ until } i < 1 \text{ loop} \]

\[ \text{print ("I will not say bad things about assistants"}) \]

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

Syntax:

```
from initialization
invariant inv
until exit_condition
loop body
variant var
end
```

Options:
- Compound
- Optional
- Boolean expression
- 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

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

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

What are the invariant and variant of the “factorial” loop?

from

\[ i := 2 \]
\[ Result := 1 \]

invariant

\[ Result = factorial (i - 1) \]

until

\[ i > n \]

loop

\[ Result := Result \ast i \]
\[ i := i + 1 \]

variant

\[ n - i + 2 \]

end

Result = 6 = 3!
Implement a function that calculates Fibonacci numbers, using a loop

\[
\text{fibonacci} \ (n : \text{INTEGER}) : \text{INTEGER} \\
\quad \quad \text{-- n-th Fibonacci number} \\
\text{require} \\
\quad \quad \text{n\_non\_negative : } n \geq 0 \\
\text{ensure} \\
\quad \quad \text{first\_is\_zero : } n = 0 \text{ implies } \text{Result} = 0 \\
\quad \quad \text{second\_is\_one : } n = 1 \text{ implies } \text{Result} = 1 \\
\quad \quad \text{other\_correct : } n > 1 \text{ implies } \text{Result} = \text{fibonacci} \ (n - 1) + \\
\quad \quad \text{fibonacci} \ (n - 2) \\
\quad \quad \text{end}
\]
Writing loops (solution)

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

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