ETH Zürich

Chair of Software Engineering

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 ...)
                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

Compilation error? (1)

Hands-On

```
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

Compilation error? (2)

Hands-On

```
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

Hands-On

```
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!

Compilation error? (3)

Hands-On

```
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

Compilation error? (4)

Hands-On

```
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

Hands-On

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

```
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
```

```
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
```

- Describe the conditions under which it is better to use a local variable instead of an attribute and vice versa

Result

- 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!

Compilation error? (5)

Hands-On

```
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
```

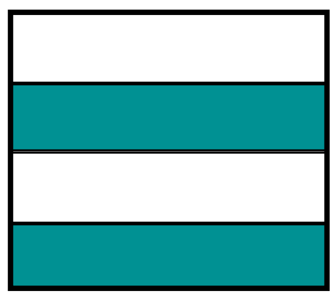
Error: Result can not be used in a procedure

Current

- 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 **Current**

x.change_name (y)

```
change_name (n: STRING)
  do
    ...
    city.internal_stations.extend (Current, n)
  end
```



(STATION)

- What is the type of **Current**?

Revisiting qualified vs. unqualified feature calls

- If the target of a feature call is **Current**, it is omitted:

$$\textbf{Current}.f\,(a)$$
$$f\,(a)$$

- 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?

Hands-On

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

1) *x*.<u>*y*</u>	qualified
2) <u>*x*</u>	unqualified
3) <u>*f*</u> (*x*.*a*)	unqualified
4) *x*.<u>*y*</u>.*z*	qualified
5) <u>*x*</u> (*y*.*f* (*a*.*b*))	unqualified
6) *f* (*x*.*a*).<u>*y*</u> (*b*)	qualified
7) **Current**.<u>*x*</u>	qualified

Assignment to attributes

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

y := 5	OK
x.y := 5	Error
Current.*y* := 5	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.

```
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

Entity: the final definition

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**

Read-write entities / variables

Read-only entities

Only a variable can be used in a creation instruction and in the left part of an assignment

Find 5 errors

Hands-On

```
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

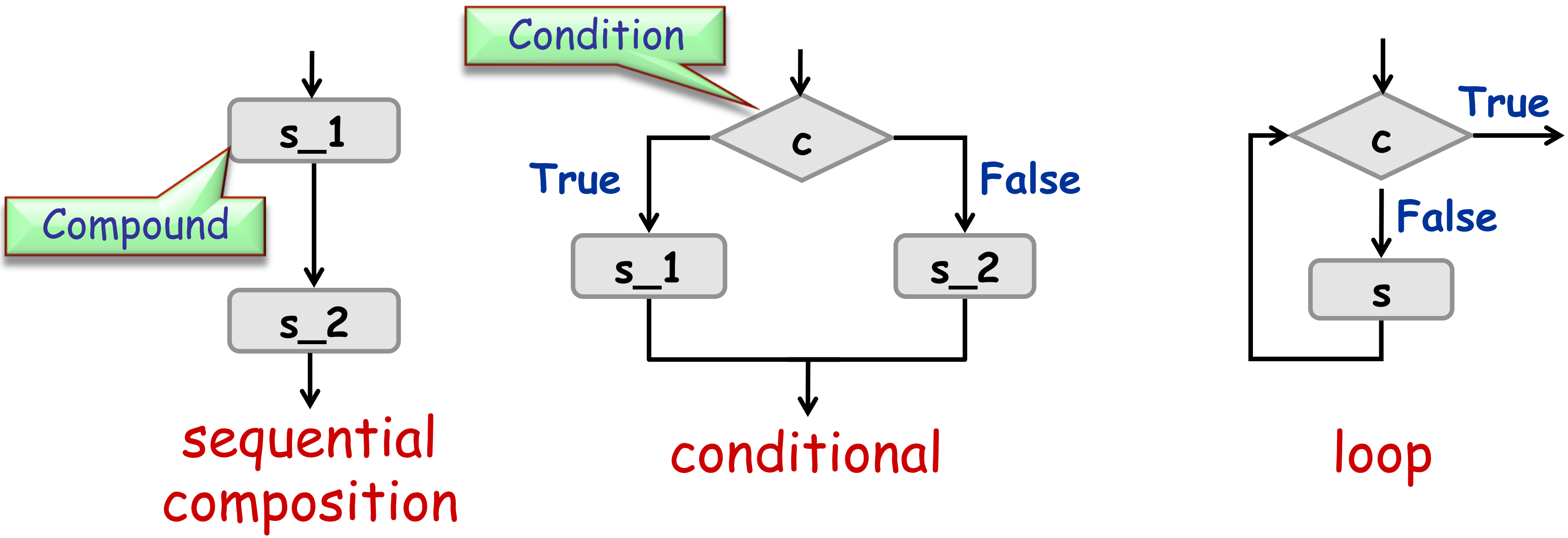
other.x is a qualified attribute call (not a variable) and thus can not be assigned to

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):



- 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
```

Condition

Compound

Compound

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

```
if c then
 s_1
end
```

```
if c then

end
```

```
if c then
else
end
```

Yes, **else** is optional

Yes, *s_1* could be empty.

Yes, *s_1* and *s_2* could be both empty.

Calculating function’s value

Hands-On

```
f (max : INTEGER ; s : STRING): STRING
    do
        if s.is_equal ("Java") then
            Result := "J**a"
        else
            if s.count > max then
                Result := "<an unreadable German word>"
            end
        end
    end
```

Calculate the value of:

- *f* (3, “Java”) → “J**a”
- *f* (20, “Immatrikulationsbestätigung”) → “<an unreadable German word>”
- *f* (6, “Eiffel”) → **Void**

Write a routine...

Hands-On

- ... that computes the maximum of two integers

max (a, b : INTEGER) : 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:

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

Condition

Compound

One word!

instead of:

```
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
```

Integer or character expression

Integer or character constant

Compound

Interval

Lost in conditions

Hands-On

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 Compound

until

exit_condition Boolean expression

loop

body Compound

end

Compilation error? Runtime error?

Hands-On

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

Compilation error: integer expression instead of boolean

Compilation error: expression instead of instruction

```
f
    do
        from
        until False
        loop

        end
    end
```

Correct, but non-terminating

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

Correct

Simple loop

Hands-On

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
```

In Eiffel we usually start counting from 1

10

```
...
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              Compound
invariant                           Optional
        inv                         Boolean expression
until
        exit_condition              Boolean expression
loop
        body                        Compound
variant                             Optional
        var                         Integer expression
end
```

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

```
sum (n: INTEGER): INTEGER
        -- Compute the sum of the numbers from 0 to `n'
    require
        0 <= n
    do
        from
            Result := 0
            i := 1
        invariant
            1 <= i and i <= n+1
            Result = (i * (i – 1)) // 2
        until
            i > n
        loop
            Result := Result + i
            i := i + 1
        variant
            n – i + 1
        end
    ensure
        Result = (n * (n + 1)) // 2
    end
```

What are the loop invariants and variants here?

What does this function do?

Hands-On

```
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

Hands-On

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 * i
        i := i + 1
variant
        n – i + 2
end
```

Result = 6 = 3!

Writing loops

Hands-On

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)

```
fibonacci (n : INTEGER) : INTEGER
    local
        a, b, i : INTEGER
    do
        if n <= 1 then
            Result := n
        else
            from
                a := 0
                b := 1
                i := 1
            invariant
                a = fibonacci (i - 1)
                b = fibonacci (i )
            until
                i = n
            loop
                Result := a + b
                a := b
                b := Result
                i := i + 1
            variant
                n - i
            end
        end
    end
```

Hands-On

Summary

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