



Einführung in die Programmierung Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 5



- Reference types vs. expanded types
- Assignment
- Basic types
- Local variables
- Qualified and unqualified calls
- Entities and variables: summary

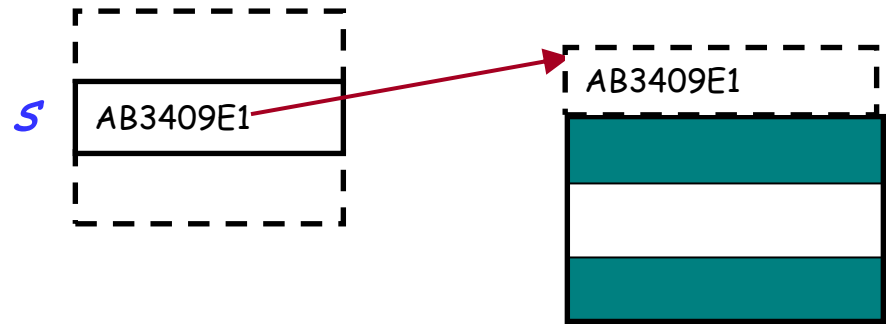
What are reference and expanded types?



Reference types: *s* contains the address (reference, or location), of the object.

Example:

s: *STATION*

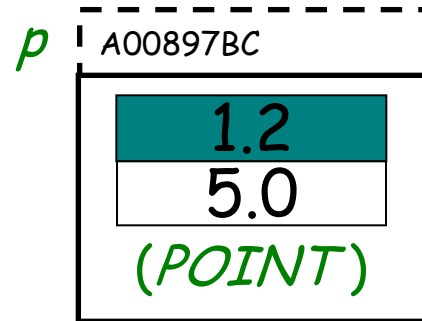


(*STATION*)

Expanded types: *p* points directly to the object.

Example:

p: *POINT*



Declaration of reference and expanded types



Objects of **reference** types: they don't exist when we declare them (they are initially *Void*).

s: STATION

We need to explicitly create them with a create instruction.

create s

Objects of **expanded** types: they exist by just declaring them (they are never *Void*)

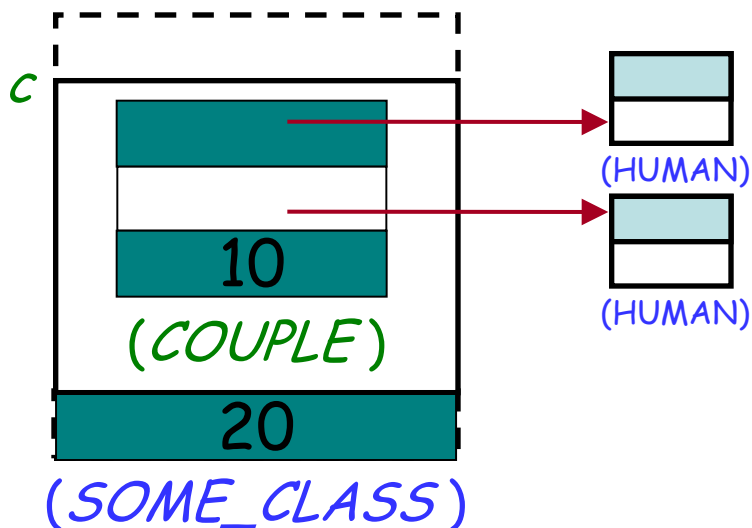
p: POINT

Feature *default_create* from *ANY* is implicitly invoked on them

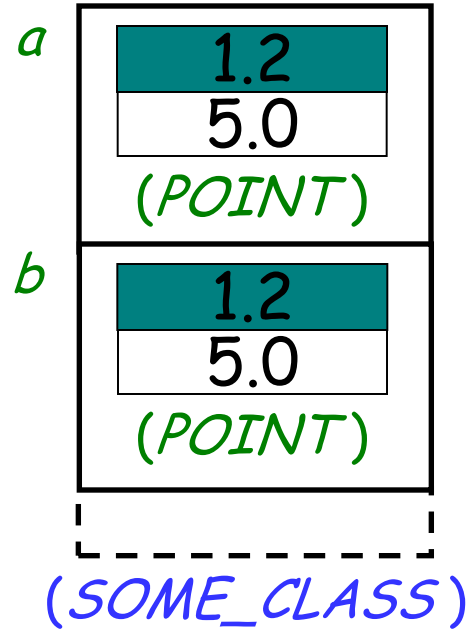
Can expanded types contain reference types?



Expanded types can contain reference types, and vice versa.



Expanded entities equality



$a = b?$

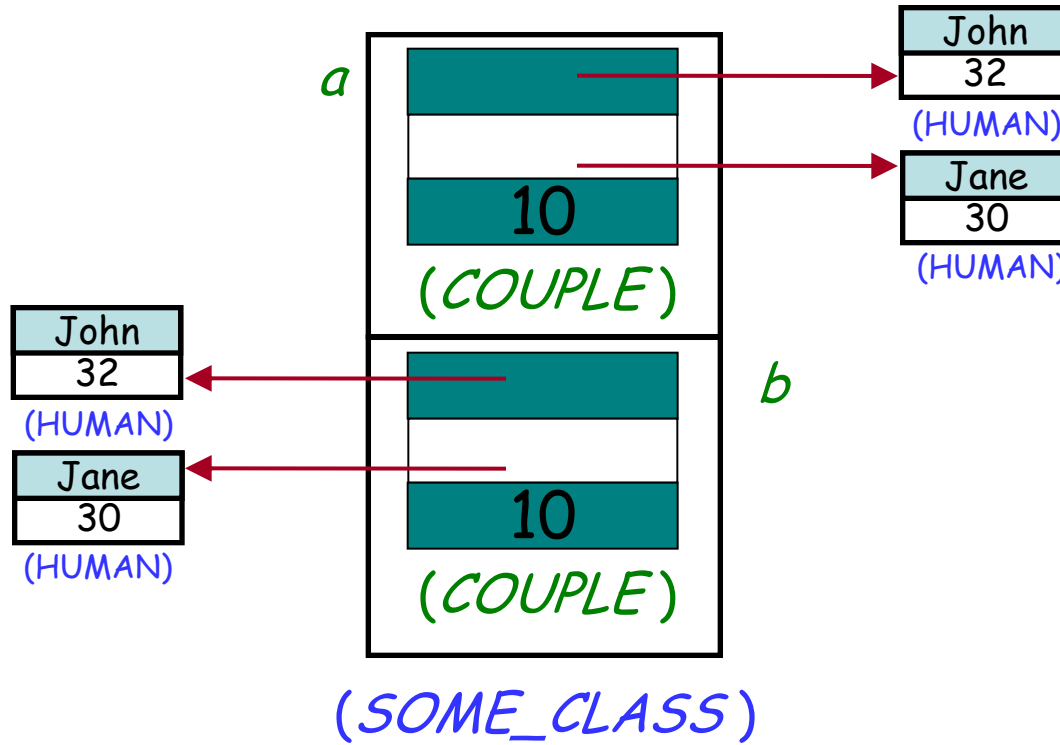
True

Entities of expanded types are compared by value!

Expanded entities equality



Hands-On



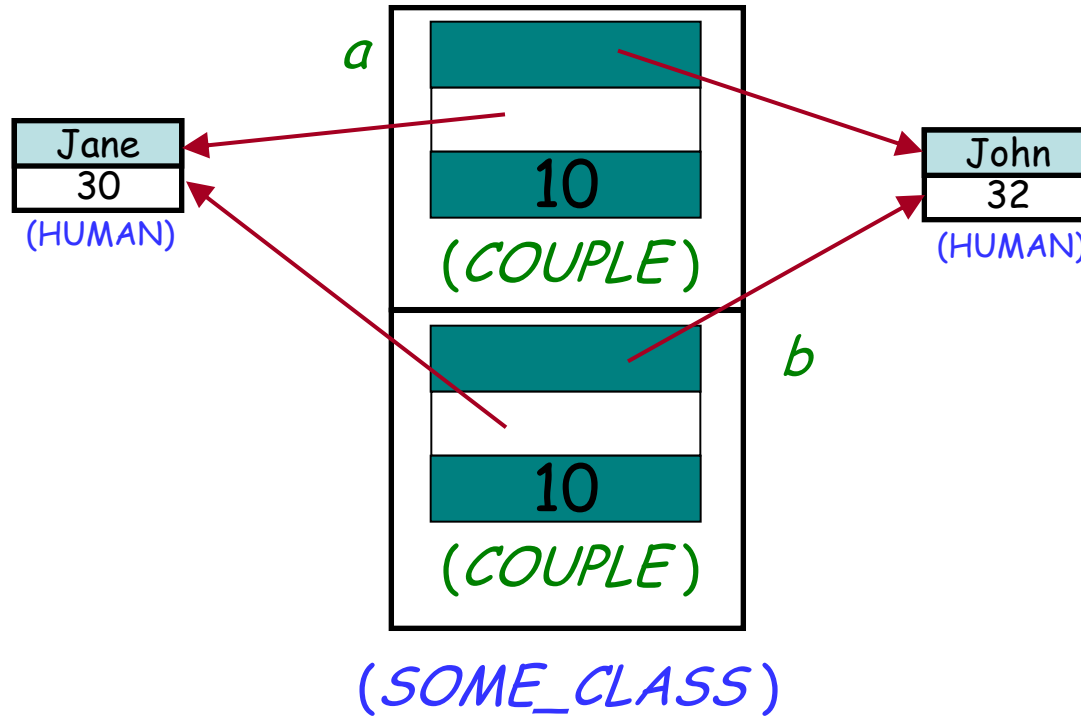
$a = b?$

False

Expanded entities equality



Hands-On



$a = b?$

True

Why expanded types?



- Representing basic types (*INTEGER, REAL,...*)
- Modeling external world objects realistically, i.e. describing objects that have sub-objects (and no sharing), for example a class *WORKSTATION* and its *CPU*.
- Possible efficiency gain.
- Interface with other languages.



➤ **Assignment** is an instruction (What other instructions do you know?)

➤ **Syntax:**

$$a := b$$

➤ where a is a variable (e.g., attribute) and b is an expression (e.g. argument, query call);

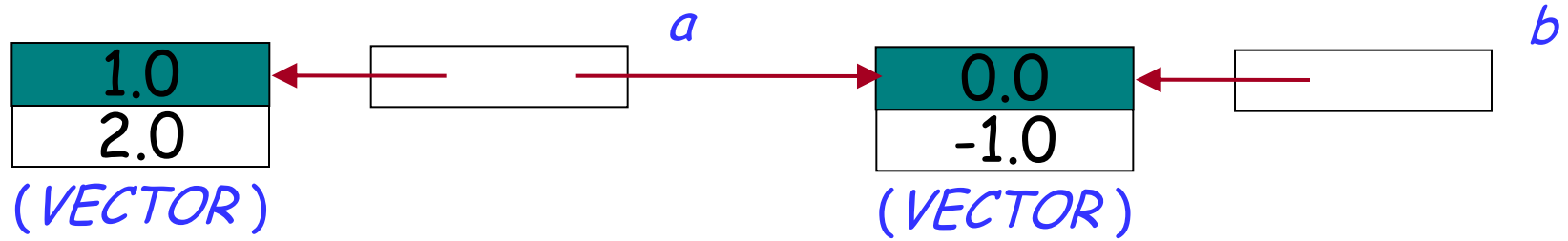
➤ a is called the **target** of the assignment and b the **source**.

➤ **Semantics:**

➤ after the assignment a equals b ($a = b$);

➤ the value of b is not changed by the assignment.

Reference assignment

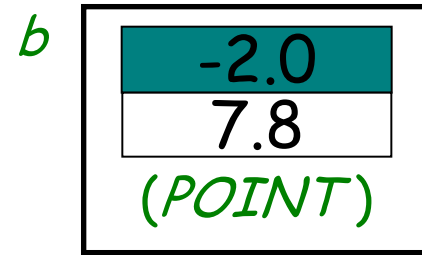
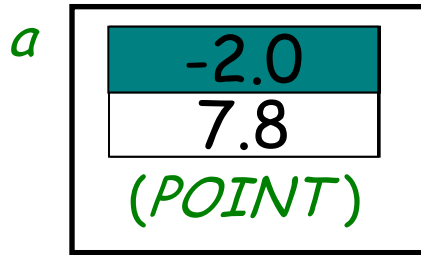


a := b

a references the same object as *b*:

a = b

Expanded assignment



a := b

The value of *b* is copied to *a*, but again:

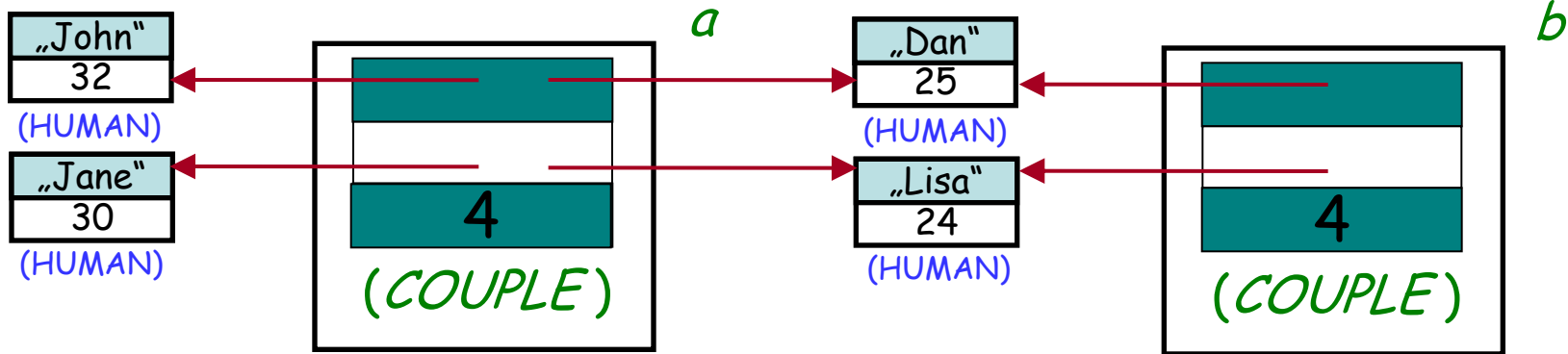
a = b

Assignment



Hands-On

Explain graphically the effect of an assignment:



$a := b$

Here **COUPLE** is an expanded class, **HUMAN** is a reference class

- More general term than assignment
- Includes:
 - Assignment

a := b

- Passing arguments to a routine

f(a: SOME_TYPE)

do ... end

f(b)

- Same semantics

Dynamic aliasing

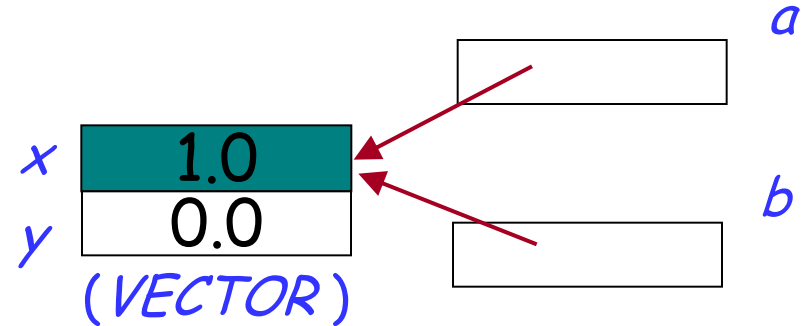


a, b: VECTOR

...

create *b.make* (1.0, 0.0)

a := b



- now *a* and *b* reference the same object (they are two names or aliases of the same object)
- any change to the object attached to *a* will be reflected when accessing it using *b*
- any change to the object attached to *b* will be reflected when accessing it using *a*

Dynamic aliasing



Hands-On

What are the values of $a.x$, $a.y$, $b.x$ and $b.y$ after executing instructions 1-4?

a, b : VECTOR

...

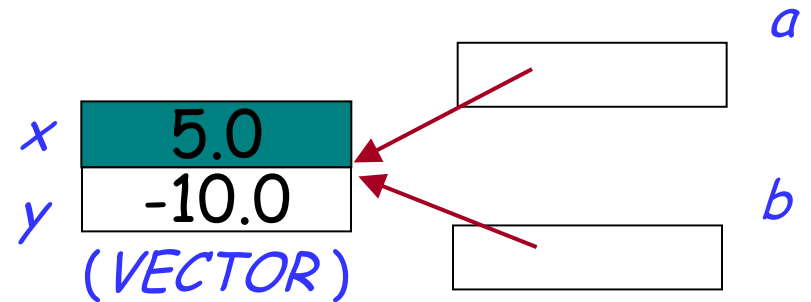
create $a.make(-1.0, 2.0)$

1 create $b.make(1.0, 0.0)$

2 $a := b$

3 $b.set_x(5.0)$

4 $a.set_y(-10.0)$



How to declare an expanded type



To get an expanded type, declare a class with keyword **expanded**:

expanded class *COUPLE*

feature -- *Access*

man, woman: HUMAN

Reference

years_together: INTEGER

?

end

Now all the entities of type *COUPLE* will automatically become expanded:

pitt_and_jolie: COUPLE

Expanded

Basic types



Their only privilege is to use **manifest constants** to construct their instances:

b: BOOLEAN

x: INTEGER

c: CHARACTER

s: STRING

...

b := True

x := 5 **-- instead of create *x.make_five***

c := 'c'

s := "I love Eiffel"

Basic types



- Some basic types (*BOOLEAN, INTEGER, NATURAL, REAL, CHARACTER*) are expanded...

$a := b$

a  b 

a  b 

- ... and immutable (they do not contain commands to change the state of their instances)...

$a := a.plus(b)$ instead of $a.add(b)$
 $a + b$

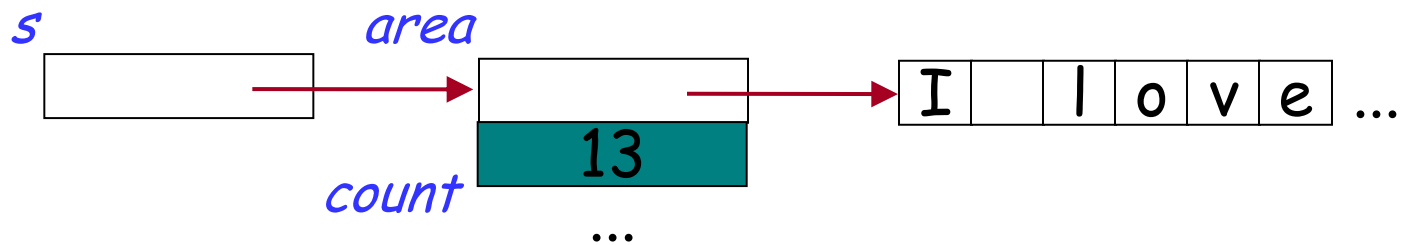
Alias for add

Strings are a bit different



Strings in Eiffel are **not** expanded...

s: *STRING*



... and **not** immutable

s := "I love Eiffel"

s.append(" very much!")

String comparison: `=` versus `is_equal`



```
s1: STRING = "Teddy"
```

```
s2: STRING = "Teddy"
```

```
...
```

```
s1 = s2 -- False: reference comparison on different objects
```

```
s1.is_equal (s2) - True
```

```
...
```

Now you know what to do if interested in comparing the content of two strings

Initialization

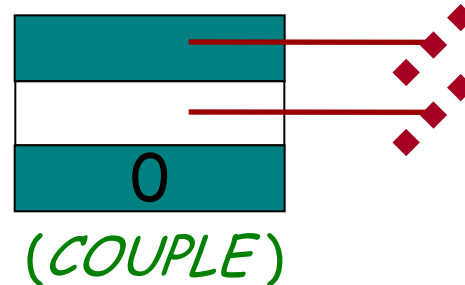


Default value of any **reference** type is **Void**

Default values of **basic expanded** types are:

- **False** for **BOOLEAN**
- 0 for numeric types (**INTEGER, NATURAL, REAL**)
- "null" character (its **code** is 0) for **CHARACTER**

Default value of a **non-basic expanded** type is an object, whose fields have default values of their types



Initialization



Hands-On

What is the default value for the following classes?

expanded class *POINT*
feature *x, y. REAL* end

<i>x</i>	0.0
<i>y</i>	0.0

(*POINT*)

class *VECTOR*
feature *x, y. REAL* end

Void

STRING

Void

Custom initialization for expanded types



- Expanded classes are not creatable using a creation feature of your choice

expanded class *POINT*

create *make*

feature *make* do *x := 5.0; y := 5.0* end

...

end

- But you can use (and possibly redefine) `default_create`

expanded class *POINT*

inherit *ANY*

redefine default_create

feature

default_create

 do

x := 5.0; y := 5.0

 end

end

Incorrect

Local variables



Some variables are only used by a certain routine.

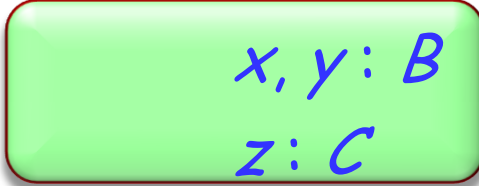
Declare them as local:

feature

f (arg1: A ...)

require ...

local



x, y: B
z: C

do ...

ensure ...

end

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

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

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 routine



- 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

Hands-On

- 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
```

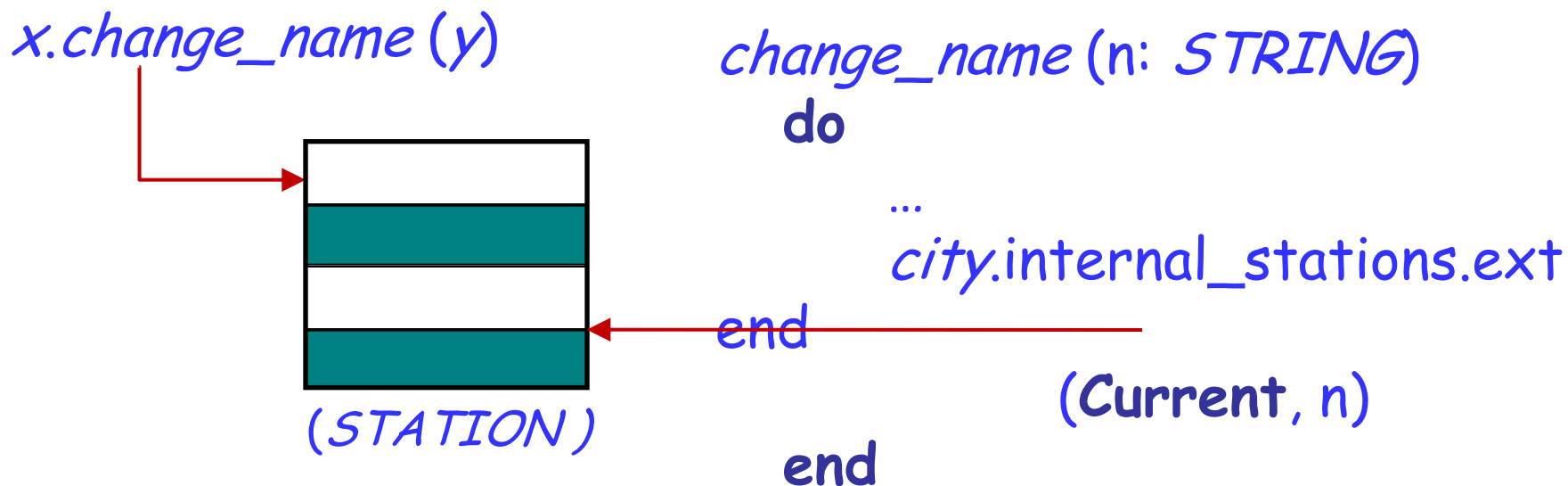
Error: Result can not be used in a procedure

```
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 **Current**



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



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

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) $x.\underline{y}$

qualified

2) \underline{x}

unqualified

3) $\underline{f}(x.a)$

unqualified

4) $x.\underline{y}.z$

qualified

5) $\underline{x}(y.f(a.b))$

unqualified

6) $f(x.a).\underline{y}(b)$

qualified

7) $\text{Current}.\underline{x}$

qualified

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

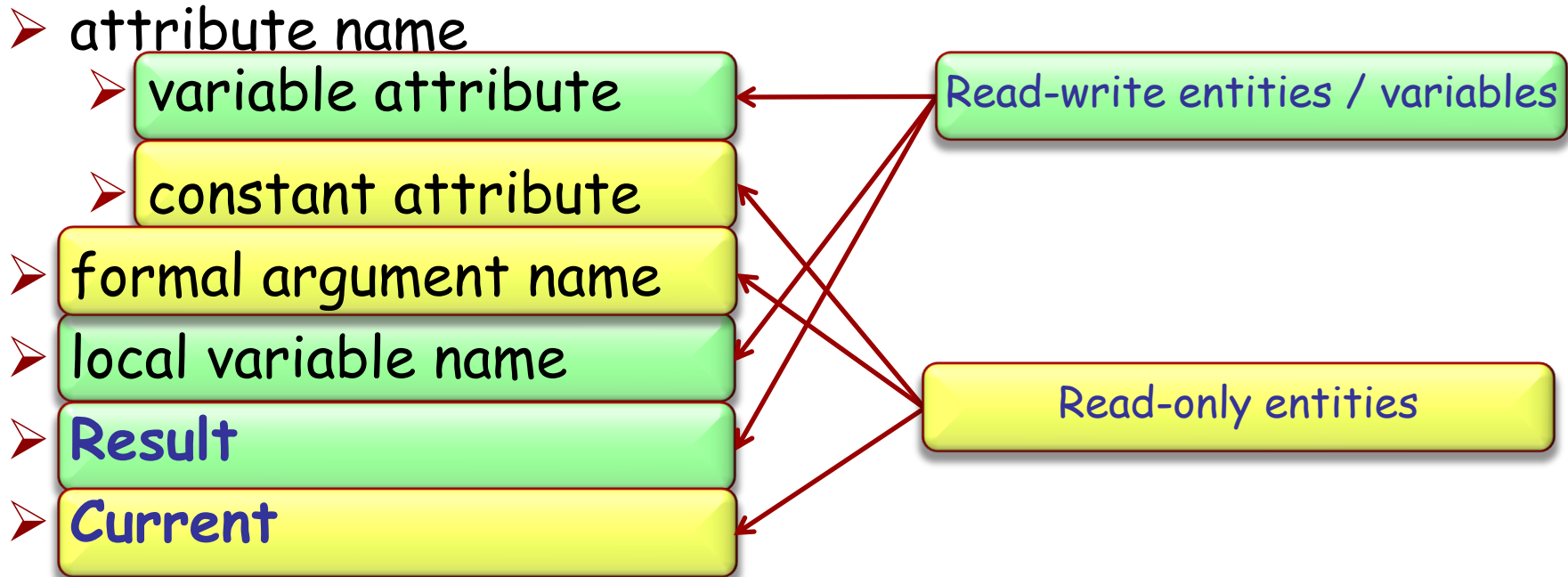
<code>y := 5</code>	OK
<code>x.y := 5</code>	Error
<code>Current.y := 5</code>	Error

- 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)

Entity: the final definition

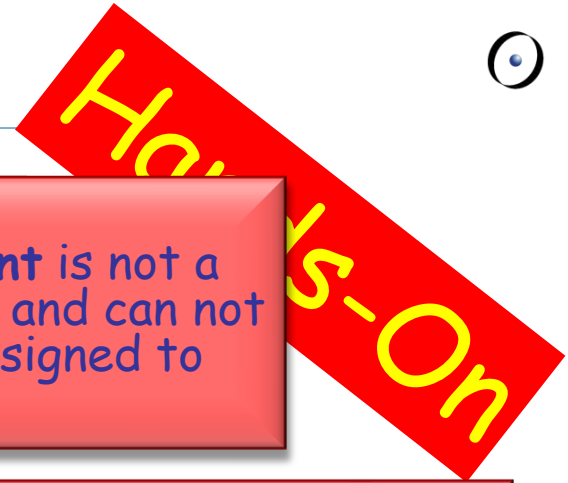


An **entity** in program text is a "name" that *directly* denotes an object. More precisely: it is one of



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
  end

  copy_to (other: VECTOR)
  do
  end

  reset
  do
  end

end
```

```
Current := other
```

```
create other
other.x := x
other.y := y
```

```
create Current
```

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