Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 8: Riferimenti

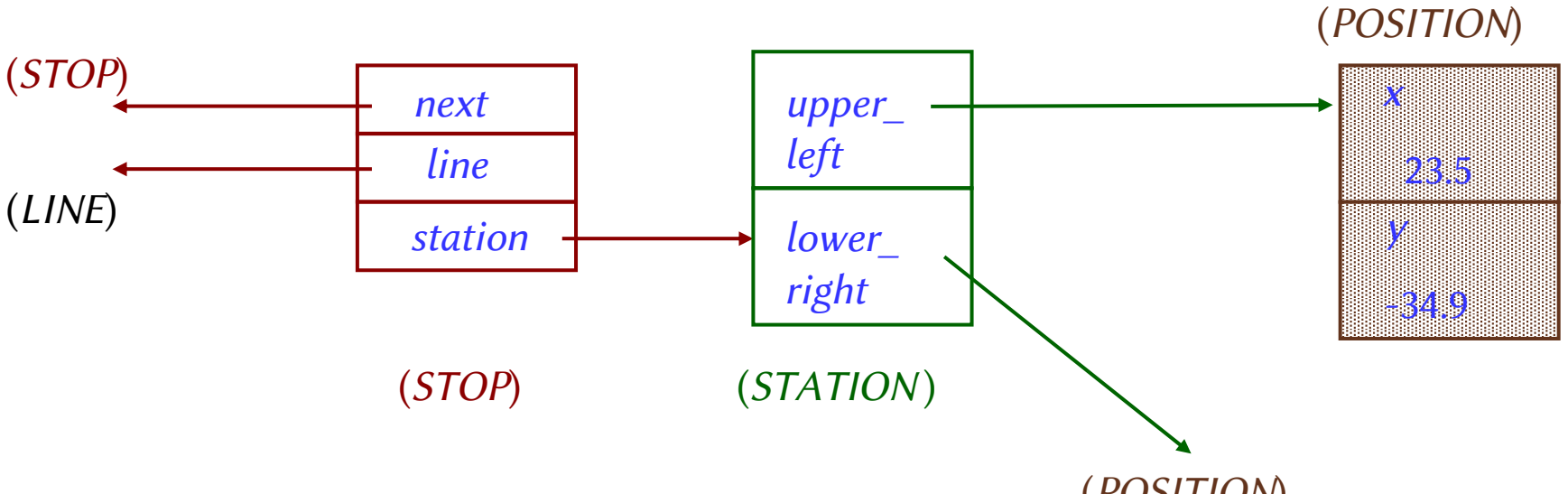
Object structure (static property)

An object is made of fields

Each field is a value, which is either:

A reference to another object

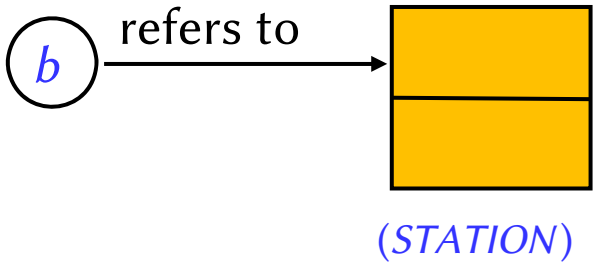
A basic value: integer, character, "real" number...
(known as an ***expanded*** value)



There are two types of values

Reference types: the value is a reference.

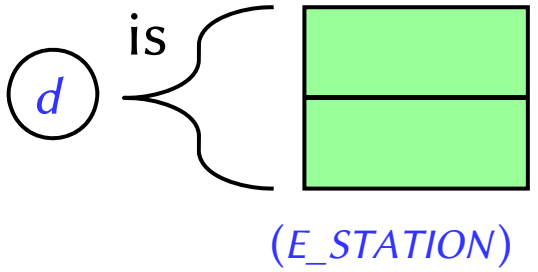
b: *STATION*



A car *has* a brand

Expanded types: the value is an object.

d: *E_STATION*



A car *has* an engine

Expanded classes

A class may be declared as

expanded class *E_STATION*

... The rest as in *STATION* ...

Then any entity declared

d: *E_STATION*

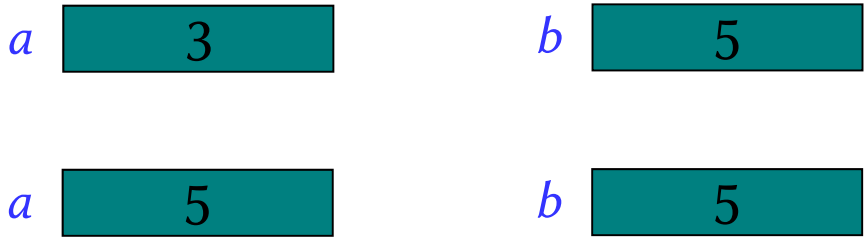
has the expanded semantics just described:

its value is an object.

Basic types

- So called basic types (*BOOLEAN*, *INTEGER*, *NATURAL*, REAL, *CHARACTER*, *STRING*) in Eiffel are classes – just like all other types
- ***Most*** of them are expanded...

$a := b$



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

The variable is re-assigned

A status changing command

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

Another name for the same feature

Basic types as expanded classes

expanded class *INTEGER* ...

(internally: *INTEGER_32*, *INTEGER_64* etc.)

expanded class *BOOLEAN* ...

expanded class *CHARACTER* ...

expanded class *REAL* ...

(internally: *REAL_32*, *REAL_64* etc.)

n : *INTEGER*

c : *CHARACTER*

s : *BOOLEAN*

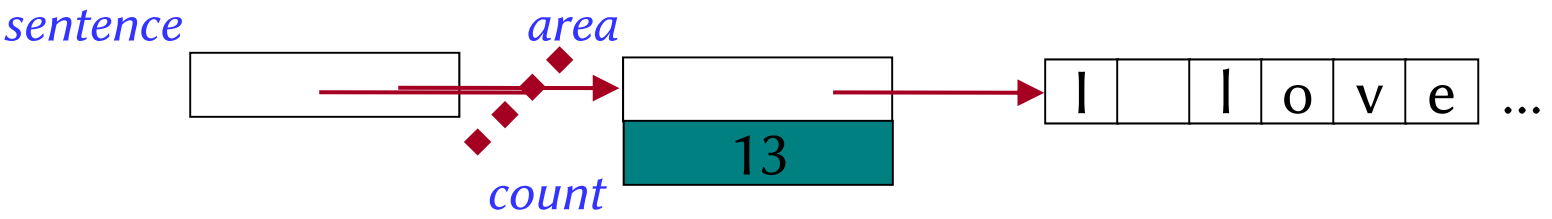
What about strings?

Strings are a bit different

Strings in Eiffel are **not** expanded:

sentence : *STRING*

is a reference to an object



... and not immutable

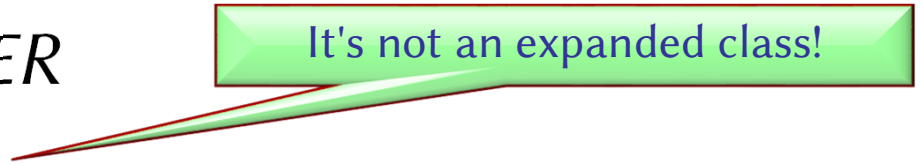
sentence := “I love Eiffel”

sentence.append (“ very much!”) [compare with *a* := *a*+*b*]

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"

Initialization

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

Default values of basic expanded types are:

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

Default value of an expanded type is an object, whose fields have default values of their types

These rules apply to:

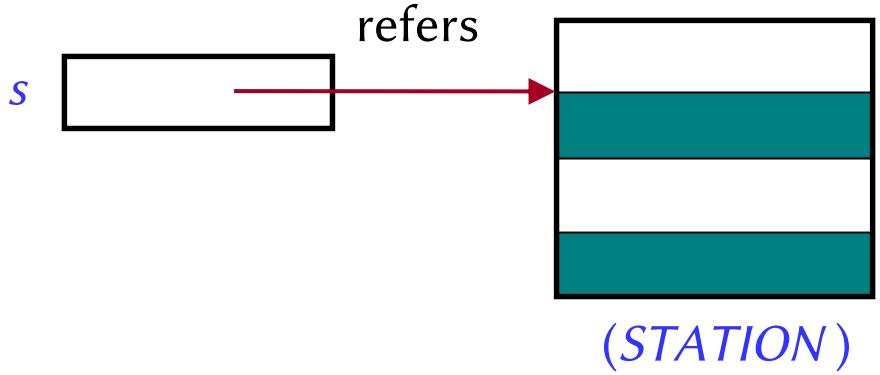
- Fields (from class attributes), on object creation
- Local variables, on start of routine execution (includes **Result**)

Two kinds of types

Reference types: value of any entity is a reference.

Example:

s : *STATION*



Expanded types: value of an entity is an object.

Example:

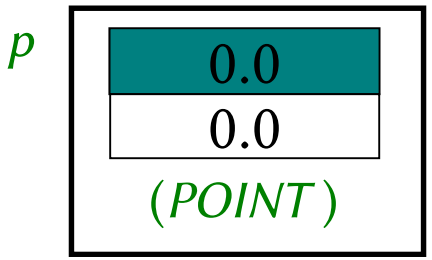
p : *POINT*

… … …

expanded class *POINT*

feature

x, y : *REAL*



Objects of reference or 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*

No need to use a **create** instruction

Feature *default_create* from *ANY* is implicitly invoked on them when creating the containing instance

How to declare an expanded type

To create an expanded type, declare the class with keyword **expanded**:

```
expanded class COUPLE
feature -- Access
        man, woman : HUMAN
        years_together : INTEGER
end
```

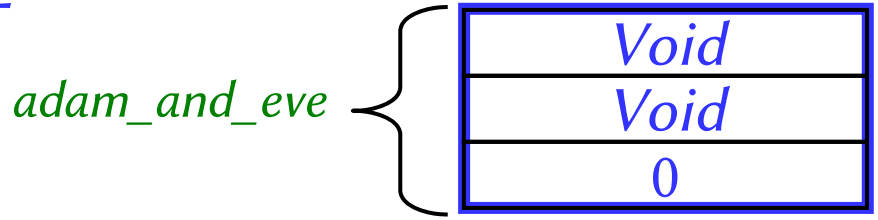
An expanded class

Reference

Expanded

Any entity of type *COUPLE* is expanded:

```
adam_and_eve: COUPLE
```



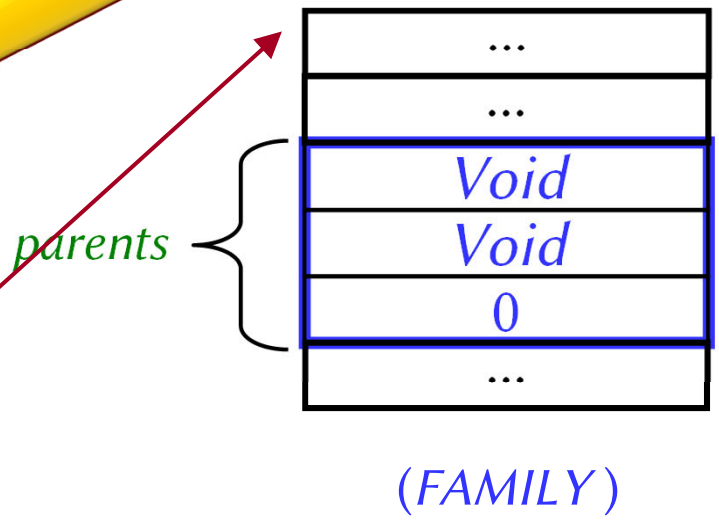
Object creation

A standard reference class

An expanded class

```
class FAMILY
feature
        parents : COUPLE
        home :  HOUSE
        phone : STRING
        ...
class CLAN
feature
        main_family : FAMILY
        ...
        create main_family
```

The instance referred to by *main_family* is created only when this instruction is executed...

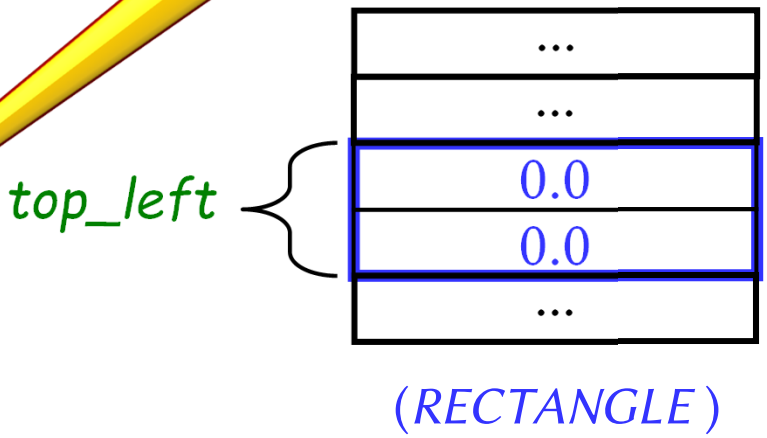


Object creation

```
class RECTANGLE
feature
        top_left : POINT
        bottom_right :  POINT
        cost : INTEGER
        color : STRING
        ...
class DRAWING
feature
        rect : RECTANGLE
        ...
        create rect
```

A reference class

When this is executed...



Initialization

Hands-On

```
class SOME_CLASS
feature p : POINT   v: detachable VECTOR
        s: detachable STRING
end
```

When creating an instance of *SOME_CLASS* which is the default initial value given to its attributes?

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

x	0.0
y	0.0

(*POINT*)

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

Void

STRING

Void

Is this correct?

```
expanded class POINT
create make
feature x, y : REAL
feature make
        do
                x := 5.0
                y := 5.0
        end
        ...
end
```

➢REMEMBER:

- Instances of expanded classes are automatically created when the object containing them is created

  For *x* : *POINT*

  There is no need of a **create** *x*

Custom initialization for expanded types?

- Expanded classes can be created **only** in the default way
  - i.e. using *default_create*, possibly redefining it, if needed for a proper initialization

```
expanded class POINT
inherit ANY
        redefine default_create
feature
        default_create
                do
                        x := 5.0; y := 5.0
                end
end
```

Do you remember this expanded class?

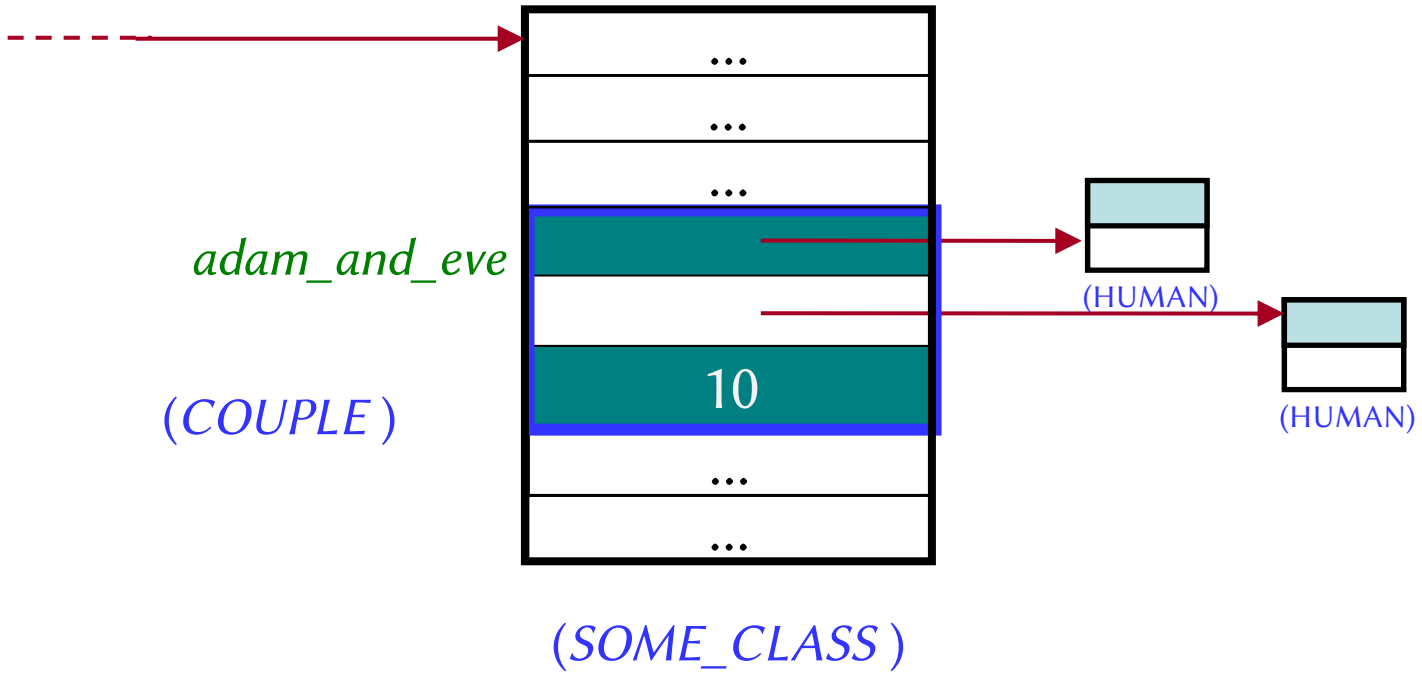

```
expanded class COUPLE
feature -- Access
        man, woman : HUMAN
        years_together : INTEGER
end
```

Assume there is *SOME_CLASS* definition with this declaration:

```
        adam_and_eve: COUPLE
```

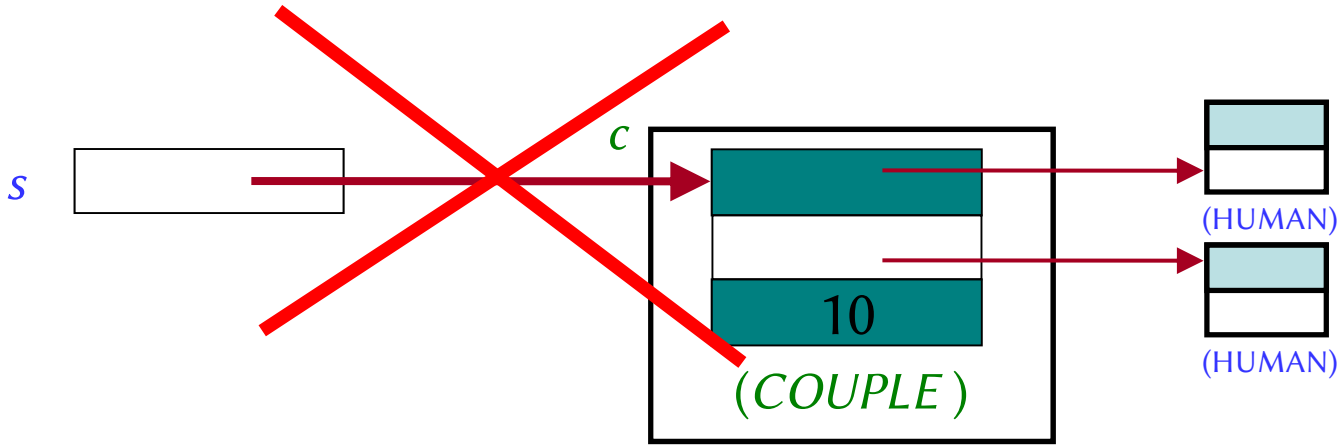
Can expanded types contain reference types?

An instance of reference type (*SOME_CLASS*) contains an instance of expanded types (*COUPLE*) which contain instances of a reference type (*HUMAN*)



Who can reference what?

Objects of expanded types can contain references to other objects...



... but they cannot be referenced by other objects!

Changing variable values: assignment

target := *source*

source is an expression and may be:

- Call to a query:
  - *position*
  - *upper_left*.*position*
- Arithmetic or boolean expression:
  - $a + (b * c)$
  - $(a < b)$ **and** $(c = d)$

target is a variable entity and may be:

- An attribute
- **Result** in a function
- A "local variable" of a routine

Semantics

- after the assignment *source* equals *target*
- the value of *source* is not changed by the assignment

Assignment to attributes

- GENERAL RULE: Direct assignment to an attribute is only allowed if an attribute is called in an unqualified way (i.e., by the object itself). Are the following allowed?

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

- There are two main reasons for the general rule:
  1. An other client may not be aware of the restrictions put on the attribute value and interdependencies with other attributes => class invariant violation
  2. The *uniform access principle* (client access independent from the implementation (memory/computation))

Effect of an assignment (1)

Reference type: value of any entity is a reference.

b, *c*: *STATION*

c := *b*

Assignment: copy the reference

Expanded type: value of an entity is an object

d, *e*: *E_STATION*

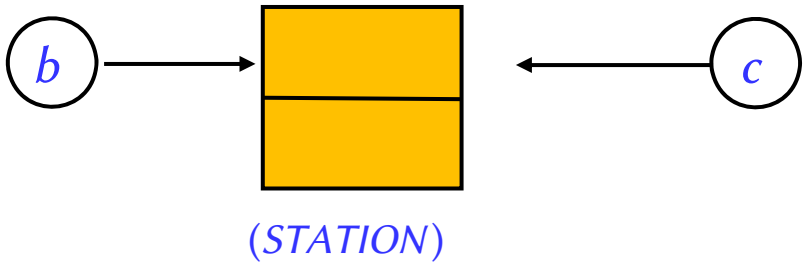
e := *d*

Assignment: copy the object

Effect of an assignment (2)

Reference type: value of any entity is a reference.

b, *c*: *STATION*

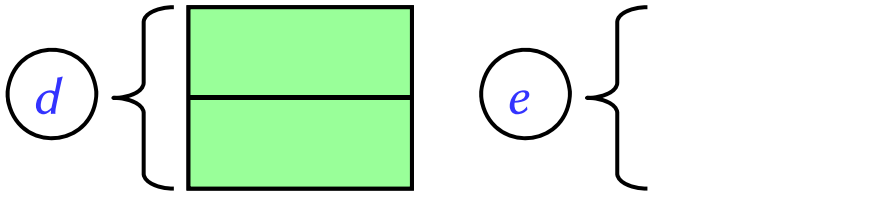


Assignment: copy the reference

c := *b*

Expanded type: value of an entity is an object

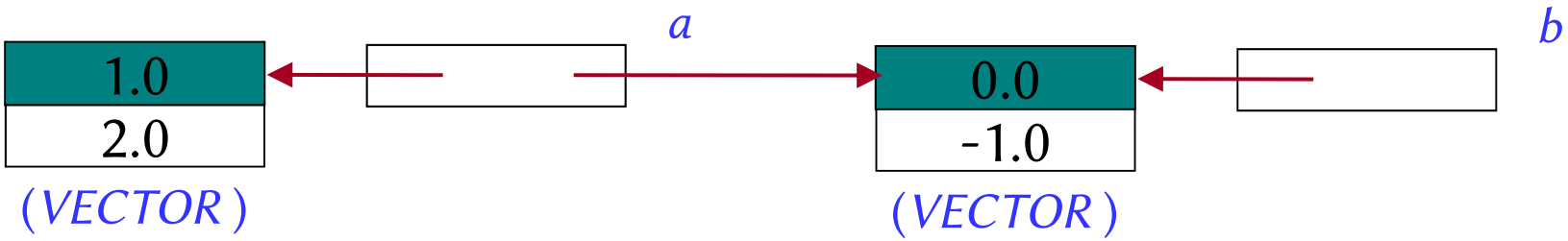
d, *e*: *E_STATION*



Assignment: copy the object

e := *d*

Reference assignment



$a := b$ Which is its effect?

now a references the same object as b :

$a = b$

Expanded assignment

$a := b$ Which is its effect?

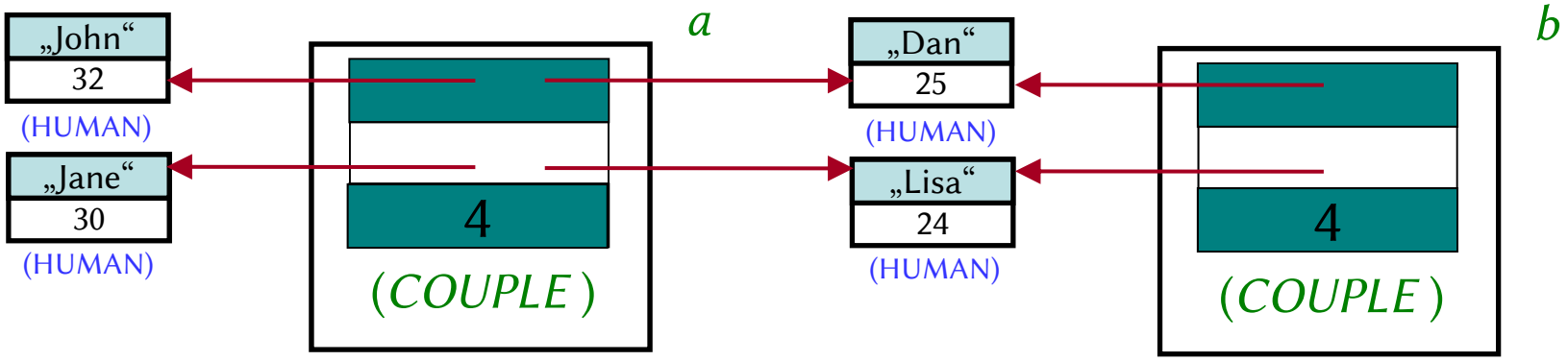
The value of b is copied into a and, again:

$a = b$

Assignment

Hands-On

Explain graphically the effect of an assignment:



$a := b$

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

Do not confuse assignment with equality

x := y

Instruction

(prescriptive and destructive)

if x = y **then**...

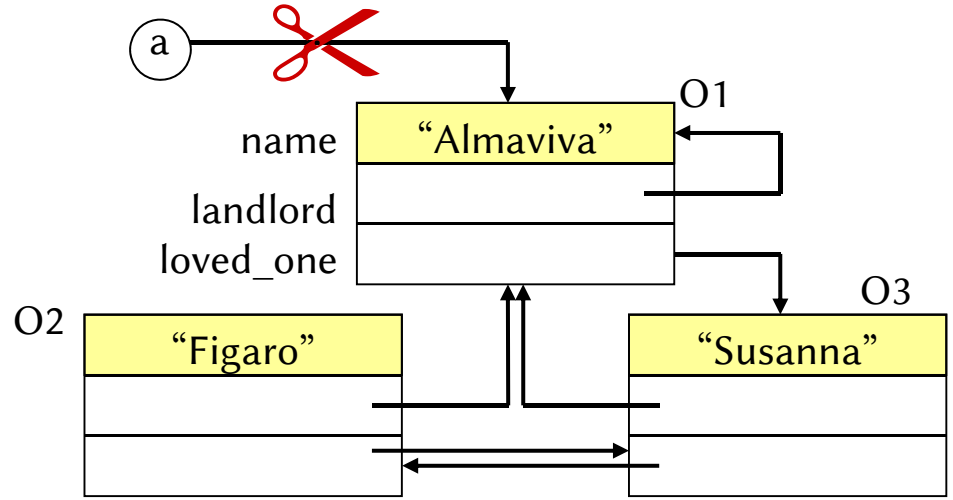
Expression (descriptive)

if x = Current **then**...

Expression (descriptive)

What to do with unreachable objects

Reference assignments may make some objects useless



Two possible approaches:

- Manual "free" (C++, Pascal)
- Automatic garbage collection (Eiffel, Oberon, Java, .NET)

Arguments for automatic collection

Manual reclamation is dangerous for reliability.

- Wrong "frees" are among the most difficult bugs to detect and correct.

Manual reclamation is tedious.

Modern garbage collectors have acceptable performance overhead.

GC is tunable: disabling, activation, parameterization....

Properties of a garbage collector (GC)

Consistency (never reclaim a reachable object).

Completeness (reclaim every unreachable object – eventually).

Consistency (also called safety) is an absolute requirement. Better no GC than an unsafe GC.

But: safe automatic garbage collection is hard in C-based languages.

The trouble with reference assignment

A comfortable mode of reasoning:

-- Here *SOME_PROPERTY* holds of *a*

"Apply *SOME_OPERATION* to *b*"

-- Here *SOME_PROPERTY* still holds of *a*

It holds for "expanded" values, e.g. integers (a=2, b=3)

-- Here *P* (a) holds, e.g. *even* (a)

OPER (b) , e.g. *increment* (b)

-- Here *P* (a) still holds of *a* , e.g. *even* (a)

Dynamic aliasing

a, b: STRING

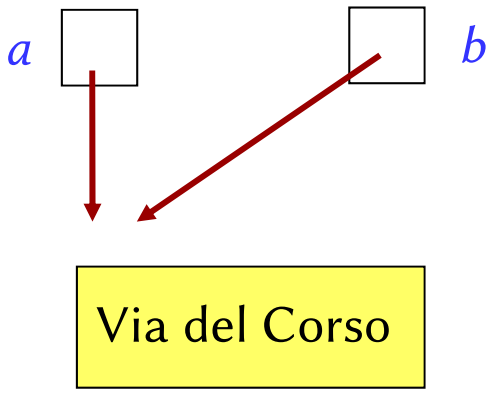
a := "*Via del Corso*"

b := *a*

-- Here *a.item* has value "*Via del Corso*"

b := "*Piazza Venezia*"

-- Here *a.item* has value ?????



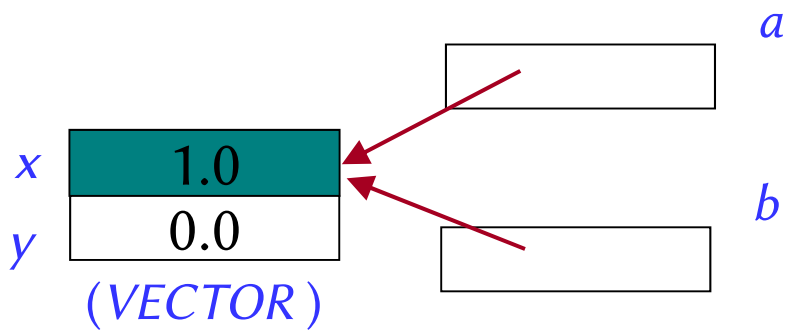
Dynamic aliasing

a, *b*: *VECTOR*

...

create *b.make* (1.0, 0.0)

a := *b*



- now *a* and *b* reference the same object (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 (1)

Hands-On

What are the values of *a.x*, *a.y*, *b.x* and *b.y* after executing each instruction 1 to 3?

a, *b*: *VECTOR*

...

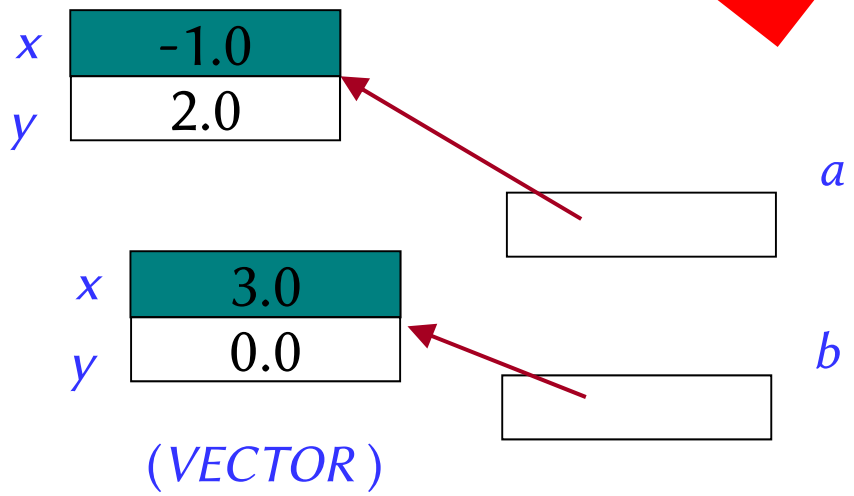
create *a.make* (-1.0, 2.0)

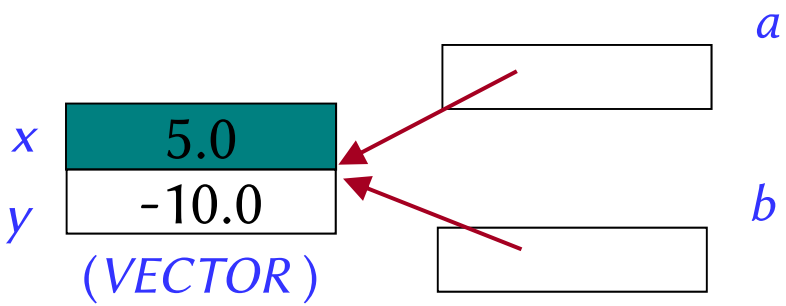
create *b.make* (3.0, 0.0)

1 *a* := *b*

2 *b.set_x* (5.0)

3 *a.set_y* (-10.0)





Dynamic aliasing (2)

Hands-On

What are the values of *a.x*, *a.y*, *b.x* and *b.y* after executing each instruction 1-3?

Now assume VECTOR is an ***expanded*** class

a, *b*: *VECTOR*

...

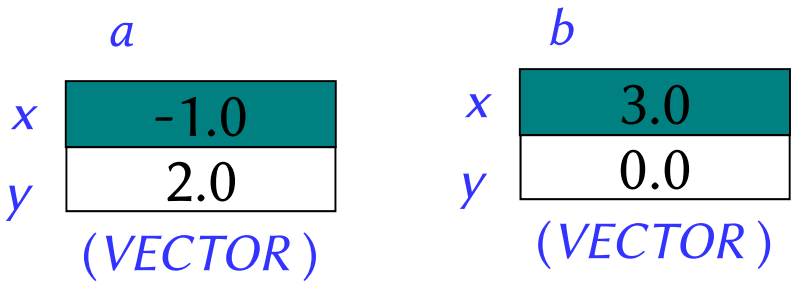
a.set (-1.0, 2.0)

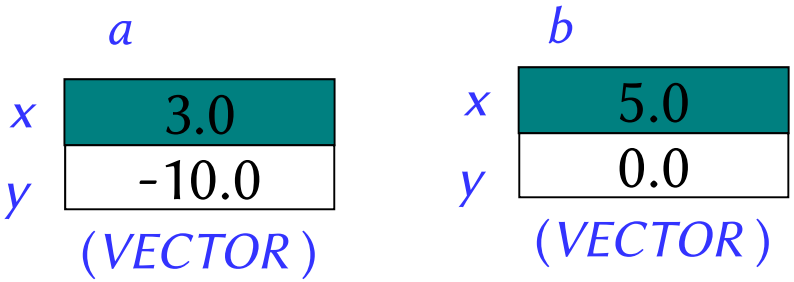
b.set (3.0, 0.0)

1 *a* := *b*

2 *b.set_x* (5.0)

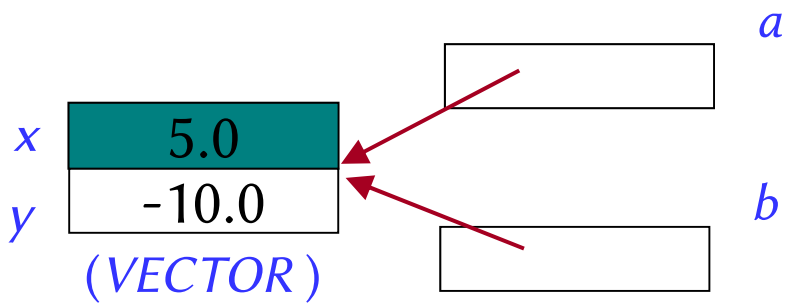
3 *a.set_y* (-10.0)



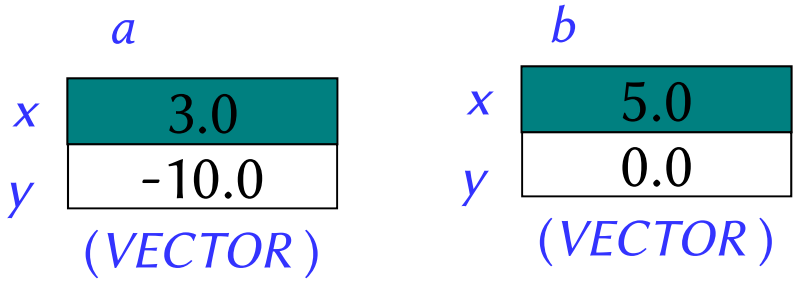


Compare the final result

VECTOR defined as **reference** type



VECTOR defined as **expanded** type



Practical advice

Reference assignment is useful

It’s also potentially tricky

As much as possible, leave it to specialized libraries of general data structures

Variants of assignment and copy

Reference assignment (*a* and *b* of the same reference types):

b := *a*

Object duplication (shallow – creates a new object whose fields receive copy of values):

c := *a.twin*

Object duplication (deep – creates a new object whose reference fields, if any, are deep duplicated):

d := *a.deep_twin*

Also: shallow field-by-field copy (an existing object receives copy of field values of another object):

e.copy (*a*)

NOTE: *c* and *d* are "*created on the fly*" by the duplication operation (or reassigned). Instead *e* must already exist

Shallow and deep cloning

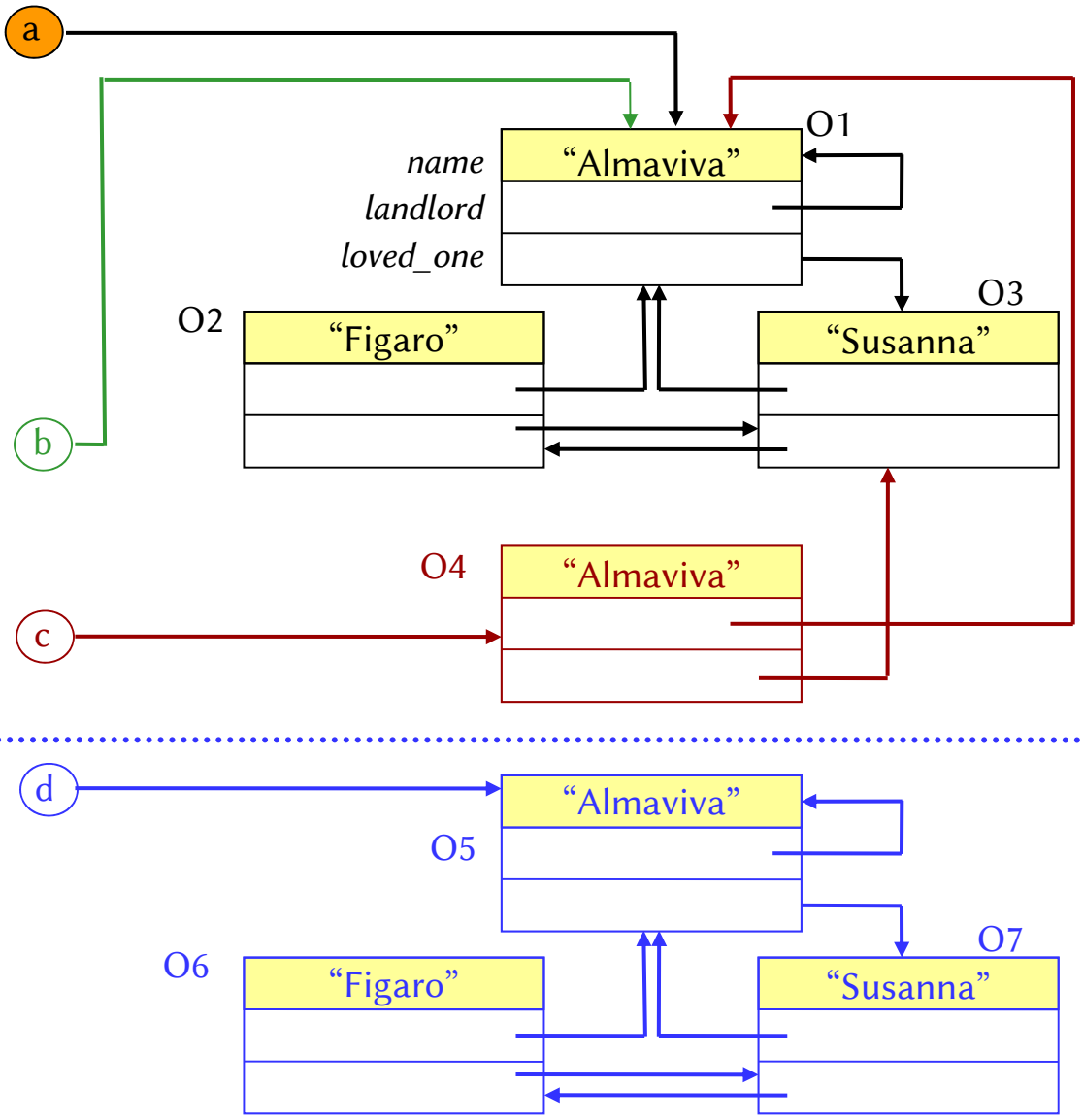
Initial situation:

Result of:

b := *a*

c := *a.twin*

d := *a.deep_twin*



Equality testing (1)

Let *a* and *b* be variable of a **reference** type

To test equality of their values (which are references) use the instruction for *reference equality*

$a = b$

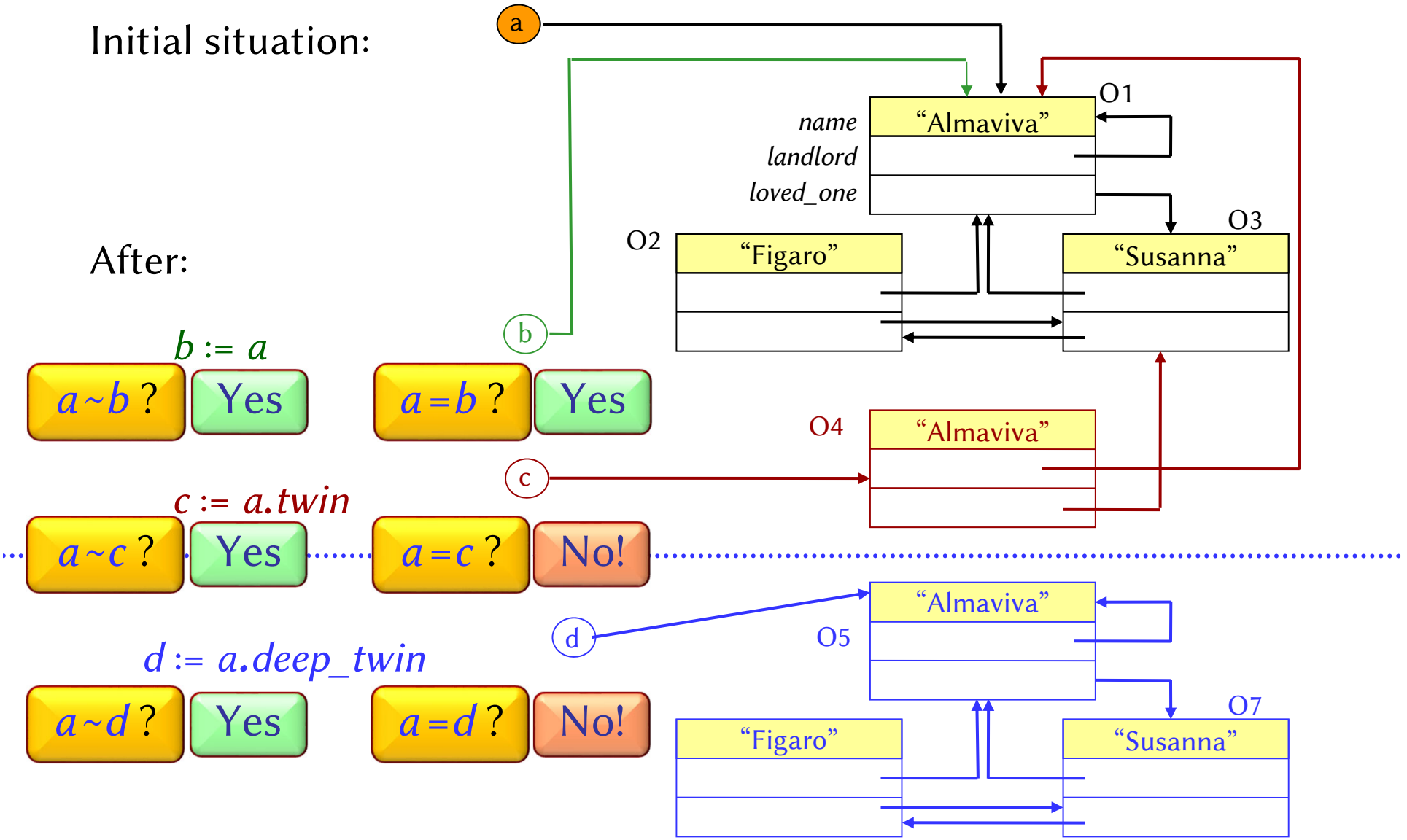
To test equality of their referenced objects, i.e. of items referenced by their values, use the instruction for ***object equality***

$a \sim b$

which is implemented by testing that all attributes are "*object equal*"

N.B.: a recursive definition

Shallow and deep cloning: equality testing

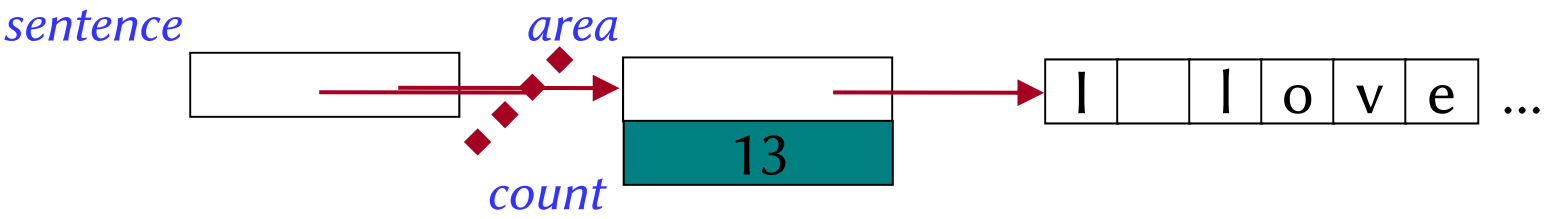


Remember strings

Strings in Eiffel are **not** expanded:

sentence : *STRING*

is a reference to an object



... and not immutable

sentence := “I love Eiffel”

sentence.append (“ very much!”) [compare with *a* := *a*+*b*]

Strings assignment and comparison

my_name, your_name: STRING

Three ways of testing equality

is_equal (a *STRING* feature) — object equality

= (equality of entities' values) — reference equality

~ (equality of referenced objects) — object equality

	is_equal ?	= ?	~ ?
my_name := "mario" *your_name* := "mario"	True	False	True
my_name := "mario" *your_name* := *my_name*	True	True	True
my_name := "mario" *your_name* := *my_name.twin*	True	False	True

Object comparison: = versus ~

s1: *STRING* = “Teddy”

s2: *STRING* = “Teddy”

…

s1 = s2 False

reference comparison on different objects

s1 ~ s2 True

object comparison

…

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

Reading and assigning strings

io is a predefined object in Eiffel referring to the system input (keyboard) and output (screen)

last_string is a query providing the (reference to the) last string that was read by **read_line** feature (but for the new line character – which is consumed but is not part of **last_string**)

Subsequent invocations of **last_string** do **not** provide a new references. Instead, the string referred to by **last_string** is rewritten every time a new string is read from the file

Then after

```
io.read_line;  s1 := io.last_string
io.read_line;  s2 := io.last_string
```

it will always be s1 = s2 = the last input line read

Use instead

```
io.read_line;  s1 := io.last_string.twin
io.read_line;  s2 := io.last_string.twin
```

to read in s1 and s2 the two input lines

Equality testing (2)

Given Eiffel is a strongly typed language:

$a = b$ implies $a \sim b$

while

when $a \sim b$ it can be $a /= b$

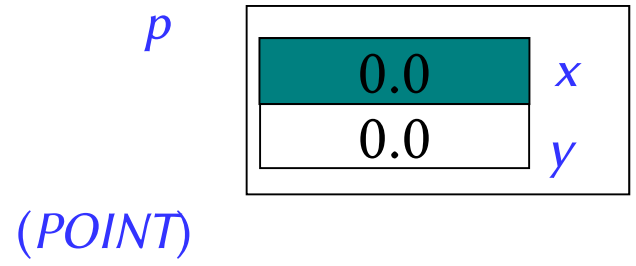
For variables of "pure" expanded type both tests always provide the same result!

An expanded type is "pure" if it is "basic" or all subtypes composing it are "pure"

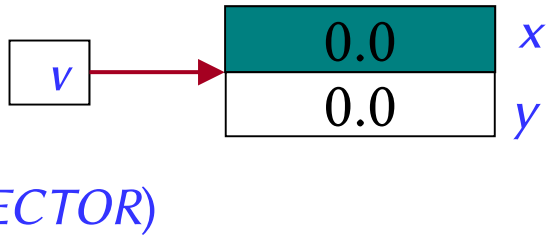
But for variables of "mixed" expanded types the two tests can have different results

Remember

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

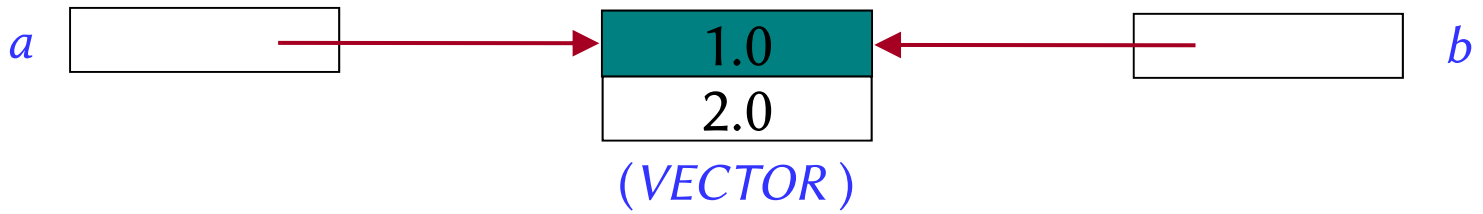


class *VECTOR*
feature x, *y* : *REAL*
end



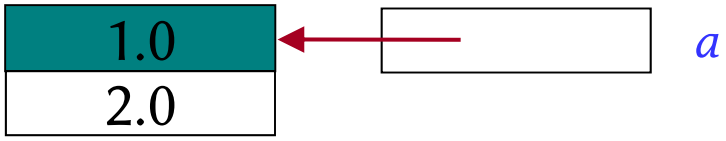
Object equality (reference types)

Hands-On

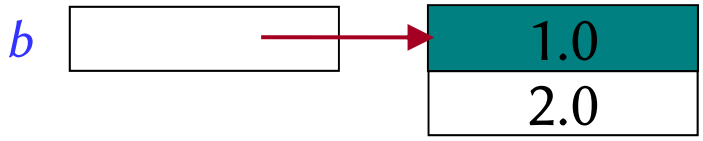


$a \sim b$? True

$a = b$? True

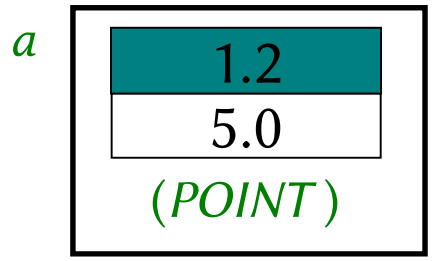


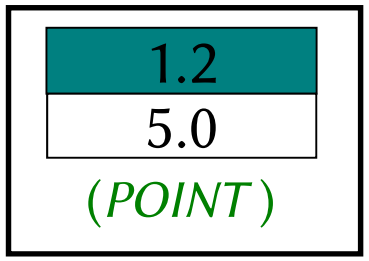
$a \sim b$? True



$a = b$? False

Object equality (expanded types)





Hands-On

$a \sim b$?

True

$a = b$?

True

Entities of expanded types are compared by value!

POINT is a "pure" expanded type: for such a type, reference equality and object equality always provide the same result!

Remember

expanded class *COUPLE*

feature -- Access

man, woman : *HUMAN*

years_together : *INTEGER*

end

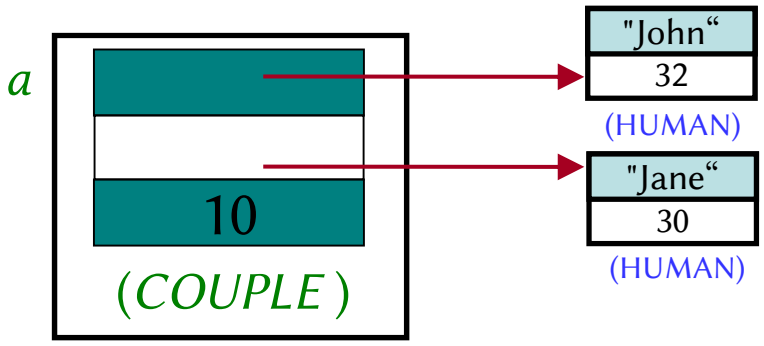
Reference type

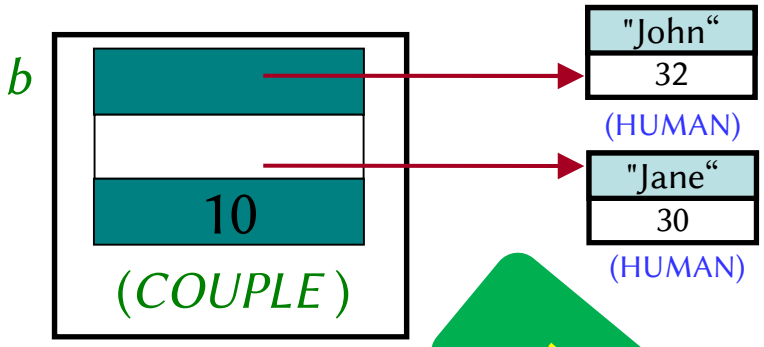
Any entities of type *COUPLE* is automatically an expanded entity:

adam_and_eve: *COUPLE*

Mixed expanded type

Expanded entities equality





Hands-On

$a \sim b$?

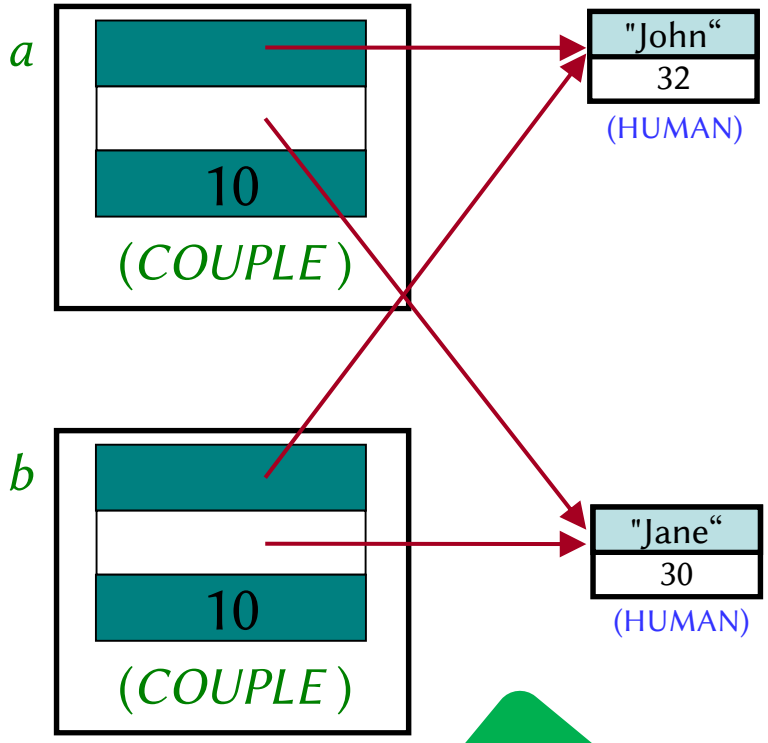
True

$a = b$?

False

COUPLE is NOT a "pure" expanded type: for such a type reference equality and object equality do not always provide the same result!

Expanded entities equality



Hands-On

$a \sim b$?

True

$a = b$?

True

COUPLE is NOT a "pure" expanded type, but in this case both equality tests give the same result!

Attachment

- Is a more general term than assignment
- Includes:
  - Assignment

$$a := b$$

  - Passing arguments to a routine

f (*a*: *SOME_TYPE*)

do ... **end**

... ...

f (*b*)

- Same semantics