ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 4

Problems in Assignment-2 Solutions

- Command or query?
  - connecting_lines
    (a_station_1, a_station_2: STATION): V_SEQUENCE [LINE]
  - Noun phrases for query names; verb phrases for command names

- Instruction separation?
  - Comma (,), space( ), semicolon (;), or nothing

- STRING_8 Vs. STRING_32

```
make
    local
        l_line: STRING_32
        c: UTF_CONVERTER
    do
        Io.read_line
        l_line := c.utf_8_string_8_to_string_32 (Io.last_string)
        print (l_line.count)
    end
```

Today

- Understanding contracts
  (preconditions, postconditions, and class invariants)
- Reference types vs. expanded types
- Basic types
- Entities and objects
- Object creation
- Assignment

Why do we need contracts at all?

- They are executable specifications that evolve together with the code
    - Together with tests, they are a great tool for finding bugs
    - They help us to reason about an O-O program at the level of classes and routines
    - Proving (part of) programs correct requires some way to specify the way the program *should* operate. Contracts are a way to specify the program

Assertions

Assertion **tag** (optional, but recommended)

Condition (required)

balance_non_negative: *balance* >= 0

Assertion clause

When the condition is violated, the assertion tag (if present) is used to construct a more informative error message.

Precondition

Property that a feature imposes on every client

```
clap (n: INTEGER)
                -- Clap n times and update count.
    require
                not_too_tired: count <= 10
                n_positive: n > 0
```

A feature with no **require** clause is always applicable, as if the precondition reads

```
    require
        always_OK: True
```

Postcondition

Property that a feature guarantees on termination

```
clap (n: INTEGER)
        -- Clap n times and update count.
    require
        not_too_tired: count <= 10
        n_positive: n > 0
    ensure
        count_updated: count = old count + n
```

A feature with no **ensure** clause always satisfies its postcondition, as if the postcondition reads

```
ensure
    always_OK: True
```

Class Invariant

Property that is true of the current object at any *observable* point

```
class ACROBAT
    ...
invariant
    count_non_negative: count >= 0
end
```

A class with no **invariant** clause has a trivial invariant

```
always_OK: True
```

Pre- and postcondition example

Hands-On

Add pre- and postconditions to:

```
smallest_power (n, bound: NATURAL): NATURAL
        -- Smallest x such that `n'^x is greater or equal `bound'.
    require
        ???
    do
        ...
    ensure
        ???
    end
```

One possible solution

```
smallest_power (n, bound: NATURAL): NATURAL
        -- Smallest x such that `n'^x is greater or equal `bound'.
    require
        n_large_enough: n > 1
        bound_large_enough: bound > 1
    do
        ...
    ensure
        greater_equal_bound: n ^ Result >= bound
        smallest: n ^ (Result - 1) < bound
    end
```

Hands-on exercise

Hands-On

Add invariant(s) to the class *ACROBAT_WITH_BUDDY*.

Add preconditions and postconditions to feature *make* in *ACROBAT_WITH_BUDDY*.

Class *ACROBAT_WITH_BUDDY*

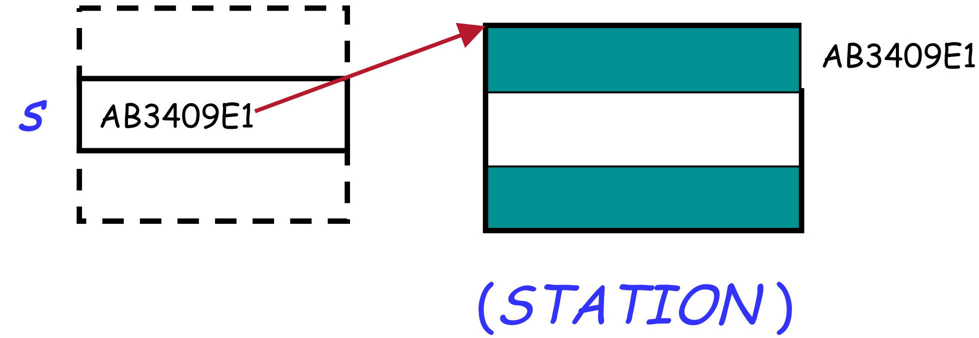
```
class
    ACROBAT_WITH_BUDDY

inherit
    ACROBAT
        redefine
            twirl, clap, count
        end

create
    make

feature
    make (p: ACROBAT)
        do
            -- Remember `p' being
            -- the buddy.
        end

    clap (n: INTEGER)
        do
            -- Clap `n' times and
            -- forward to buddy.
        end

    twirl (n: INTEGER)
        do
            -- Twirl `n' times and
            -- forward to buddy.
        end

    count: INTEGER
        do
            -- Ask buddy and return his
            -- answer.
        end

    buddy: ACROBAT
end
```

What are reference and expanded types?

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

Example:

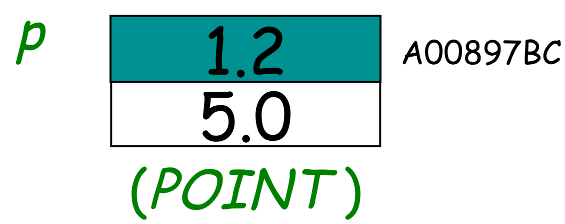
s : *STATION*



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

Example:

p : *POINT*



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.*

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
```

Reference

?

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

```
        pitt_and_jolie: COUPLE
```

Expanded

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

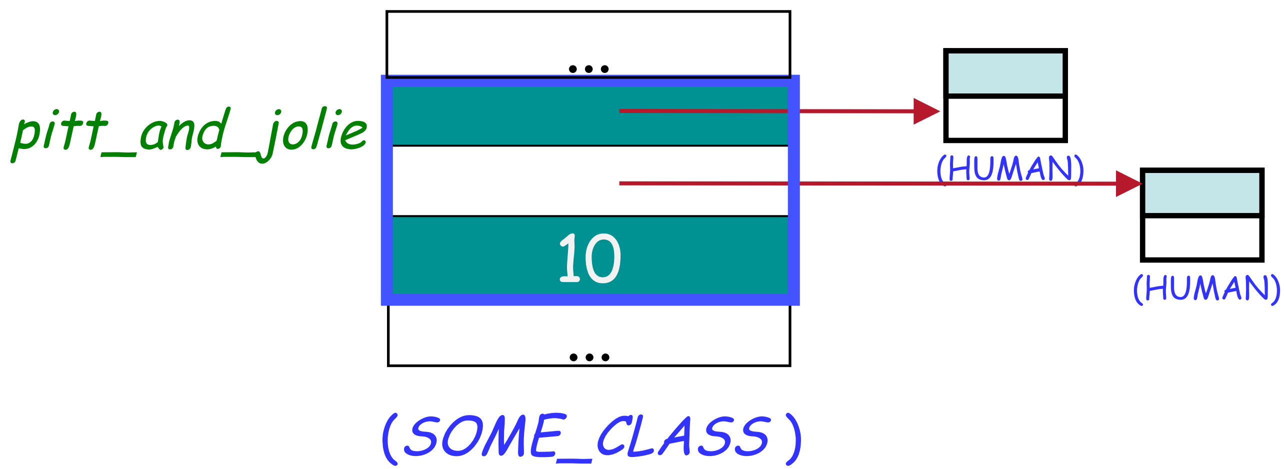
Objects of expanded types: they exist by just declaring them (they are never *Void*)

p : *POINT*

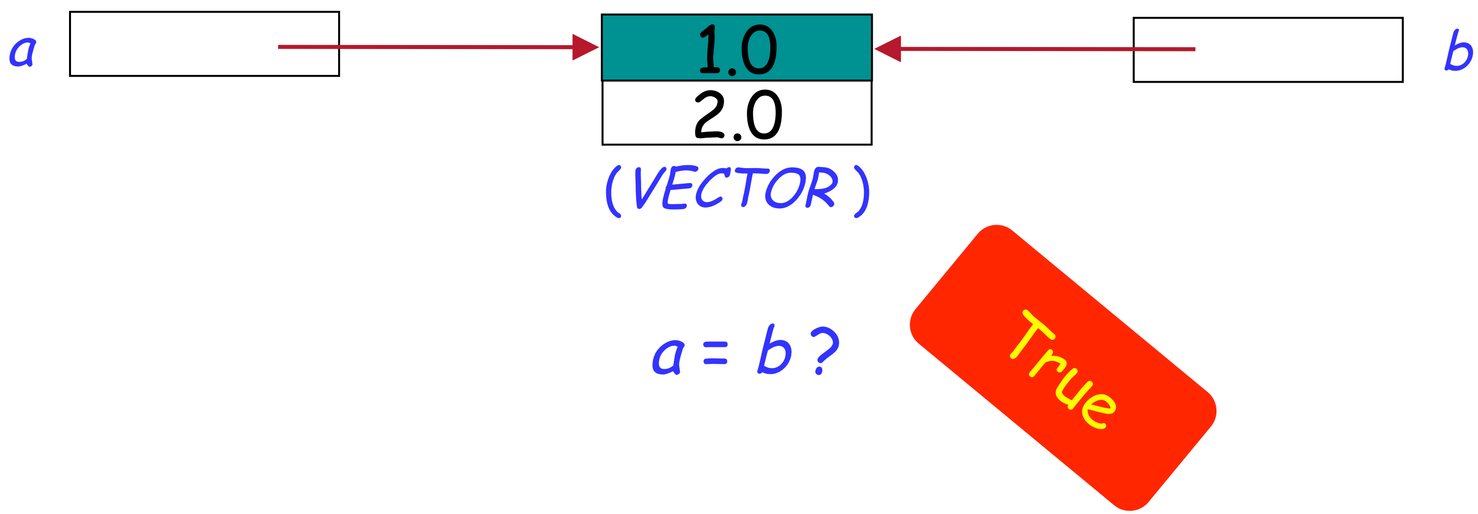
Feature *default_create* from *ANY* is implicitly invoked on them

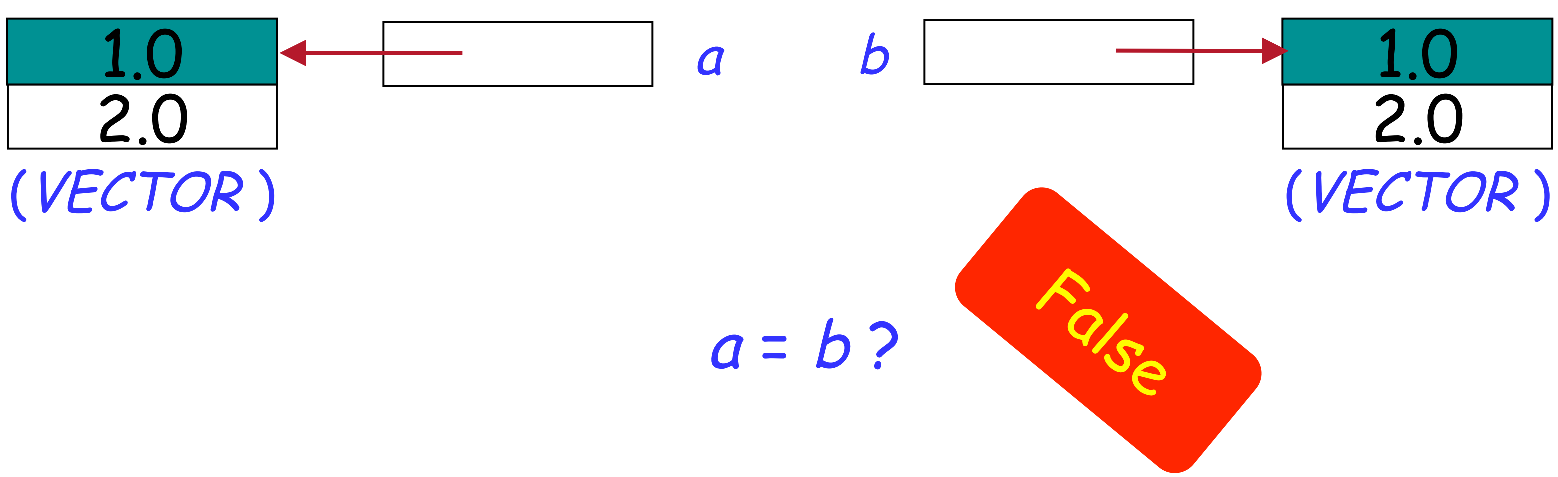
Can expanded types contain reference types?

Expanded types can contain reference types, and vice versa.

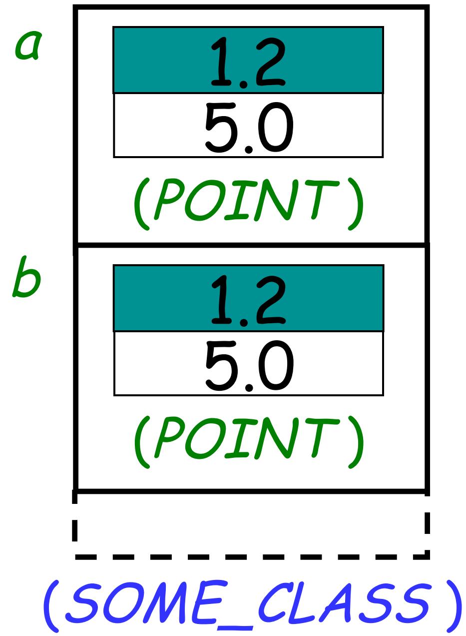


Reference equality

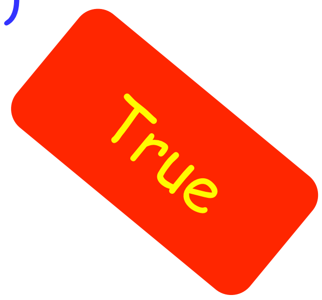




Expanded entities equality



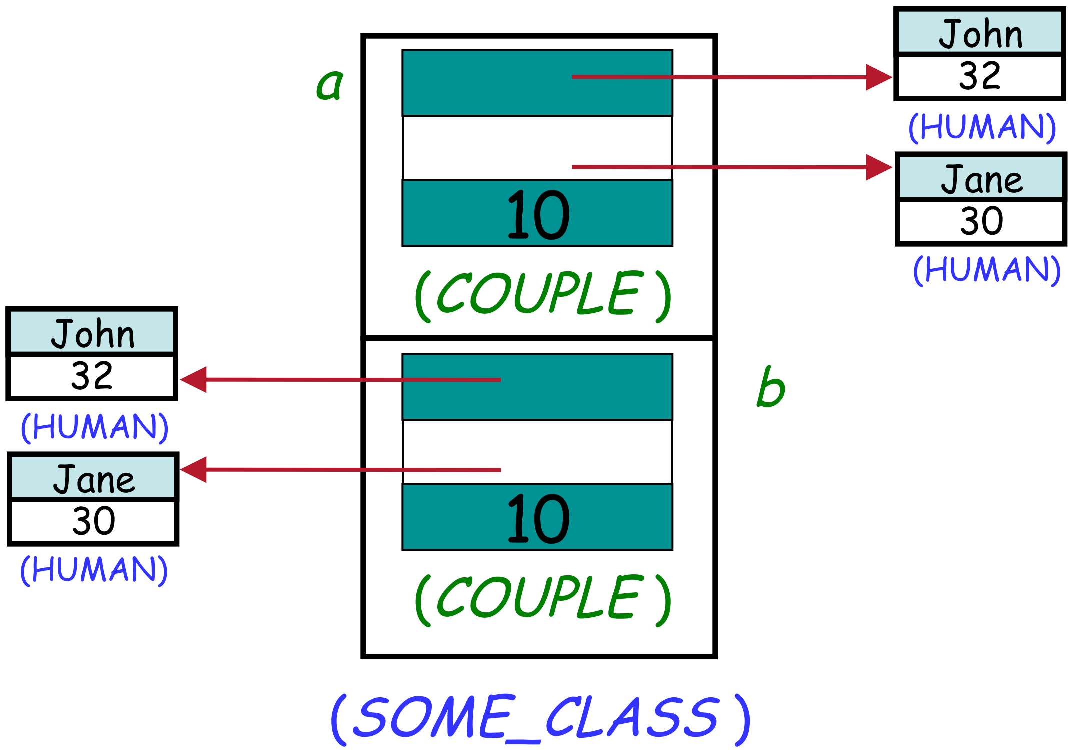
$a = b$?



Entities of expanded types are compared by value!

Expanded entities equality

Hands-On

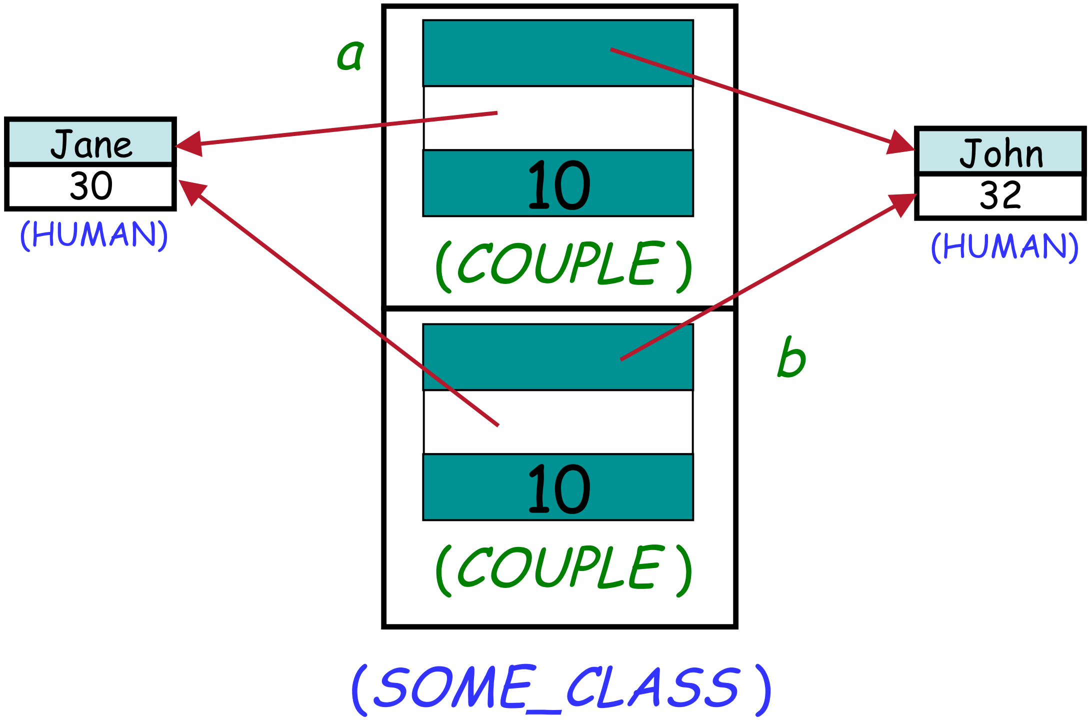


$a = b$?

False

Expanded entities equality

Hands-On



$a = b$?

True

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	3	*b*	5
a	5	*b*	5

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

a := *a*.*plus* (*b*) instead of *a*.*add* (*b*)

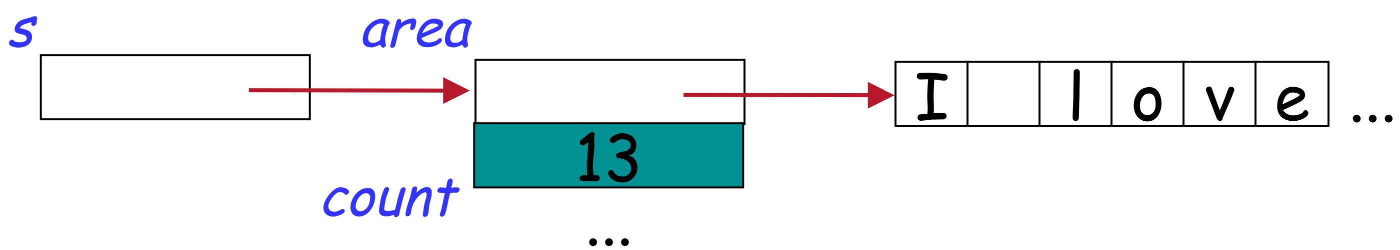
a + *b*

Alias for *plus*

Strings are a bit different

Strings in Eiffel are not expanded...

s: *STRING*



... and not immutable

s := "I love Eiffel"

s.append (" very much!")

Object comparison: = versus ~

```
s1: STRING = "Teddy"
s2: STRING = "Teddy"
…
s1 = s2 -- False: reference comparison on different objects

s1 ~ s2 --True
…
```

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

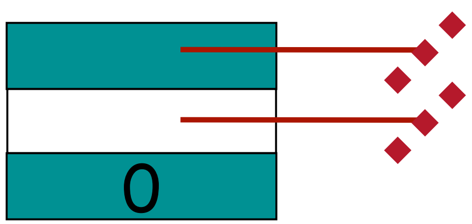
Initialization

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



(*COUPLE*)

Initialization

Hands-On

What is the default value for the following classes?

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

x	0.0
y	0.0

(*POINT*)

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

Void

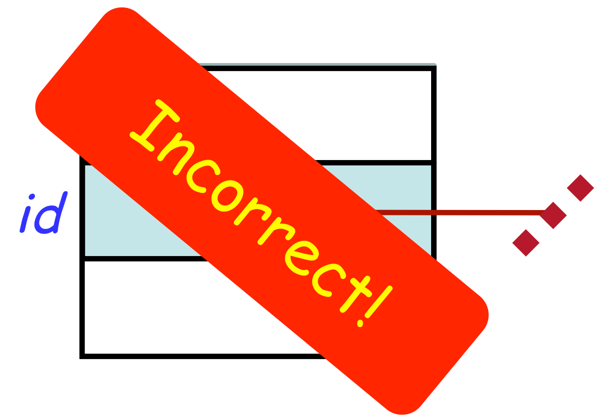
STRING

Void

Creation procedures

- Instruction **create** *x* will initialize all the fields of the new object attached to *x* with default values
- What if we want some specific initialization? E.g., to make object consistent with its class invariant?

```
Class CUSTOMER
...
        id : STRING
invariant
        id /= Void
```



- Use creation procedure:

create *a_customer.set_id* (“13400002”)

Class *CUSTOMER*

```
class CUSTOMER

create set_id

feature
        id : STRING
            -- Unique identifier for Current.

        set_id (a_id : STRING)
                -- Associate this customer with `a_id'.
            require
                id_exists : a_id /= Void
            do
                id := a_id
            ensure
                id_set : id = a_id
            end

invariant
        id_exists : id /= Void
end
```

List one or more creation procedures

May be used as a regular command and as a creation procedure

Is established by *set_id*

Object creation

To create an object:

- If class has no **create** clause, use basic form:

  create *x*

- If the class has a **create** clause listing one or more procedures, use

  create *x.make* (...)

  where *make* is one of the creation procedures, and (...) stands for arguments if any.

Some acrobatics

Hands-On

```
class DIRECTOR
create prepare_and_play
feature
    acrobat1, acrobat2, acrobat3: ACROBAT
    friend1, friend2: ACROBAT_WITH_BUDDY
    author1: AUTHOR
    curmudgeon1: CURMUDGEON

    prepare_and_play
            do
                author1.clap (4)
                friend1.twirl (2)
                curmudgeon1.clap (7)
                acrobat2.clap (curmudgeon1.count)
                acrobat3.twirl (friend2.count)
                friend1.buddy.clap (friend1.count)
                friend2.clap (2)
            end
end
```

What entities are used in this class?

What's wrong with the feature *prepare_and_play*?

Some acrobatics

Hands-On

```
class DIRECTOR
create prepare_and_play
feature
    acrobat1, acrobat2, acrobat3: ACROBAT
    friend1, friend2: ACROBAT_WITH_BUDDY
    author1: AUTHOR
    curmudgeon1: CURMUDGEON

    prepare_and_play
            do
1               create acrobat1
2               create acrobat2
3               create acrobat3
4               create friend1.make_with_buddy (acrobat1)
5               create friend2.make_with_buddy (friend1)
6               create author1
7               create curmudgeon1
            end
end
```

Which entities are still **Void** after execution of line 4?

Which of the classes mentioned here have creation procedures?

Why is the creation procedure necessary?

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
```

Incorrect

- 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
```

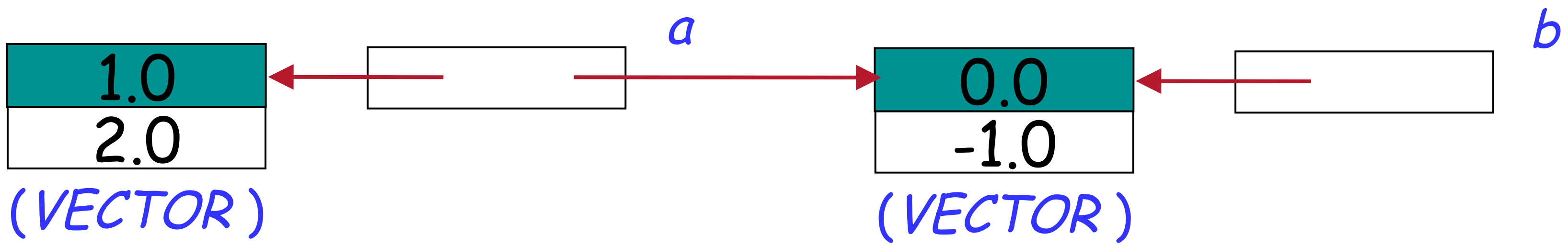
Assignment

- 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

$a := b$

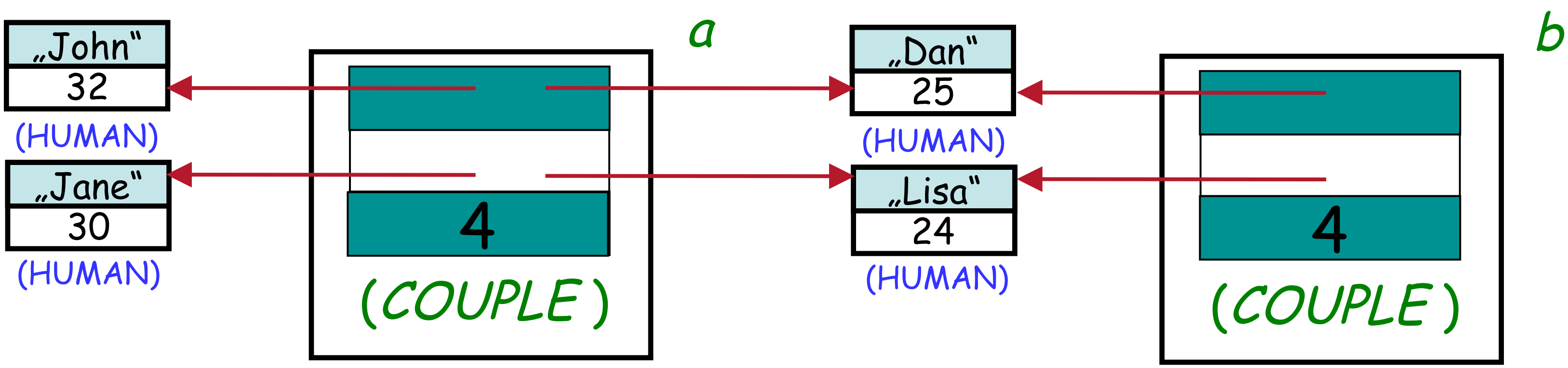
The value of b is copied to a, but again:

$a = b$

Assignment

Hands-On

Explain graphically the effect of an assignment:



a := *b*

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

Attachment

- More general term than assignment
- Includes:
  - Assignment

    a := *b*

  - Passing arguments to a routine

```
f (a: SOME_TYPE)
    do … end

f (b)
```

- Same semantics

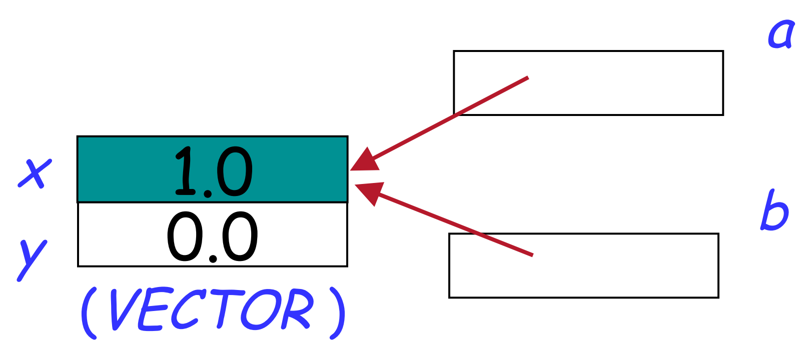
Dynamic aliasing

a, b: VECTOR

...

create *b.make* (1.0, 0.0)

a := *b*



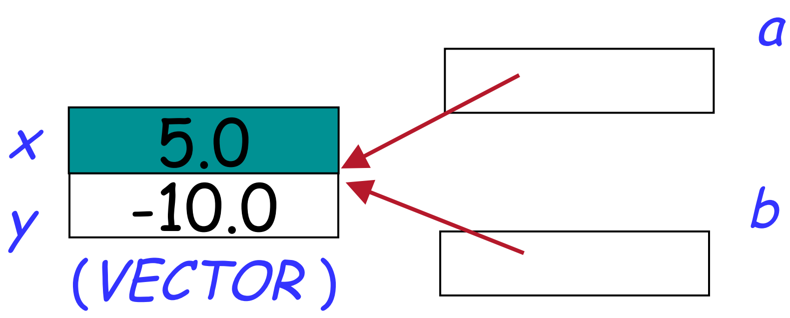
- now *a* and *b* reference the same object (they are two names or aliases of the same object)
- any change to the object attached to *a* will be reflected when accessing it using *b*
- any change to the object attached to *b* will be reflected when accessing it using *a*

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)
```



Meet Teddy