



Einführung in die Programmierung Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 4



- A bit of logic
- Understanding contracts (preconditions, postconditions, and class invariants)
- Entities and objects
- Object creation

- Constants: **True, False**
- Atomic formulae (propositional variables): **P, Q, ...**
- Logical connectives: **not, and, or, implies, =**
- Formulae: φ, χ, \dots are of the form
 - **True**
 - **False**
 - **P**
 - **not φ**
 - **φ and χ**
 - **φ or χ**
 - **φ implies χ**
 - **$\varphi = \chi$**

Tautology

- **True** for all truth assignments
 - $P \text{ or } (\text{not } P)$
 - $\text{not } (P \text{ and } (\text{not } P))$
 - $(P \text{ and } Q) \text{ or } ((\text{not } P) \text{ or } (\text{not } Q))$

Contradiction

- **False** for all truth assignments
 - $P \text{ and } (\text{not } P)$

Satisfiable

- **True** for at least one truth assignment

Equivalent

- φ and χ are equivalent if they are satisfied under exactly the same truth assignments, or if $\varphi = \chi$ is a tautology

Tautology / contradiction / satisfiable?



P or Q

satisfiable

P and Q

satisfiable

P or (not P)

tautology

P and (not P)

contradiction

Q implies (P and (not P))

satisfiable

Hands-On

Equivalence



Hands-On

Does the following equivalence hold? Prove.

$$(P \text{ implies } Q) = (\text{not } P \text{ implies not } Q)$$

Does the following equivalence hold? Prove.

$$(P \text{ implies } Q) = (\text{not } Q \text{ implies not } P)$$

De Morgan laws

$$\text{not } (P \text{ or } Q) = (\text{not } P) \text{ and } (\text{not } Q)$$

$$\text{not } (P \text{ and } Q) = (\text{not } P) \text{ or } (\text{not } Q)$$

Implications

$$P \text{ implies } Q = (\text{not } P) \text{ or } Q$$

$$P \text{ implies } Q = (\text{not } Q) \text{ implies } (\text{not } P)$$

Equality on Boolean expressions

$$(P = Q) = (P \text{ implies } Q) \text{ and } (Q \text{ implies } P)$$

- Domain of discourse: D
- Variables: $x: D$
- Functions: $f: D^n \rightarrow D$
- Predicates: $P: D^n \rightarrow \{\text{True}, \text{False}\}$
- Logical connectives: **not, and, or, implies, =**
- Quantifiers: \forall, \exists
- Formulae: φ, χ, \dots are of the form
 - $P(x, \dots)$
 - **not φ | φ and χ | φ or χ | φ implies χ | $\varphi = \chi$**
 - $\forall x \varphi$
 - $\exists x \varphi$

Existential and universal quantification



There exists a human whose name is Bill Gates

$\exists h: \text{Human} \mid h.\text{name} = \text{"Bill Gates"}$

All persons have a name

$\forall p: \text{Person} \mid p.\text{name} \neq \text{Void}$

Some people are students

$\exists p: \text{Person} \mid p.\text{is_student}$

The age of any person is at least 0

$\forall p: \text{Person} \mid p.\text{age} \geq 0$

Nobody likes Rivella

$\forall p: \text{Person} \mid \text{not } p.\text{likes}(\text{Rivella})$

$\text{not } (\exists p: \text{Person} \mid p.\text{likes}(\text{Rivella}))$

Tautology / contradiction / satisfiable?



Let the domain of discourse be **INTEGER**

$x < 0$ or $x \geq 0$

tautology

$x > 0$ implies $x > 1$

satisfiable

$\forall x \mid x > 0$ implies $x > 1$

contradiction

$\forall x \mid x * y = y$

satisfiable

$\exists y \mid \forall x \mid x * y = y$

tautology

Hands-On

Semi-strict operators (**and then**, **or else**)

➤ *a* **and then** *b*

has same value as *a* and *b* if *a* and *b* are defined, and has value **False** whenever *a* has value **False**.

about_joe: text != Void and then text.contains("Joe")

➤ *a* **or else** *b*

has same value as *a* or *b* if *a* and *b* are defined, and has value **True** whenever *a* has value **True**.

nobody_subscribed: list = Void or else list.is_empty

Strict or semi-strict?



Hands-On

- $a = 0$ or $b = 0$
- $a \neq 0$ and $b \neq 0$
- $a \neq \text{Void}$ and $b \neq \text{Void}$
- $a < 0$ or $\text{sqrt}(a) > 2$
- $(a = b \text{ and } \text{input type="checkbox"/> $b \neq \text{Void}$) \text{ and } \text{input type="checkbox"/> not $a.name.is_equal("")$$

Assertion tag (not
required, but
recommended)

Condition
(required)

balance_non_negative: balance >= 0

Assertion clause

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
```

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
```


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



Add pre- and postconditions to:

```
square_root (i: INTEGER): INTEGER  
  -- Rounded square root of `i`  
do  
  ...  
end
```

Hands-On

One possible solution



```
square_root (i: INTEGER): INTEGER
  -- Rounded square root of `i`
  require
    i_positive: i >= 0
  ensure
    not_too_large:
       $abs(\text{Result} * \text{Result} - i) < abs((\text{Result} + 1) * (\text{Result} + 1) - i)$ 
    not_too_small:
       $abs(\text{Result} * \text{Result} - i) < abs((\text{Result} - 1) * (\text{Result} - 1) - i)$ 
  end
```

Hands-On

Add invariants to classes *ACROBAT_WITH_BUDDY* and *CURMUDGEON*.

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

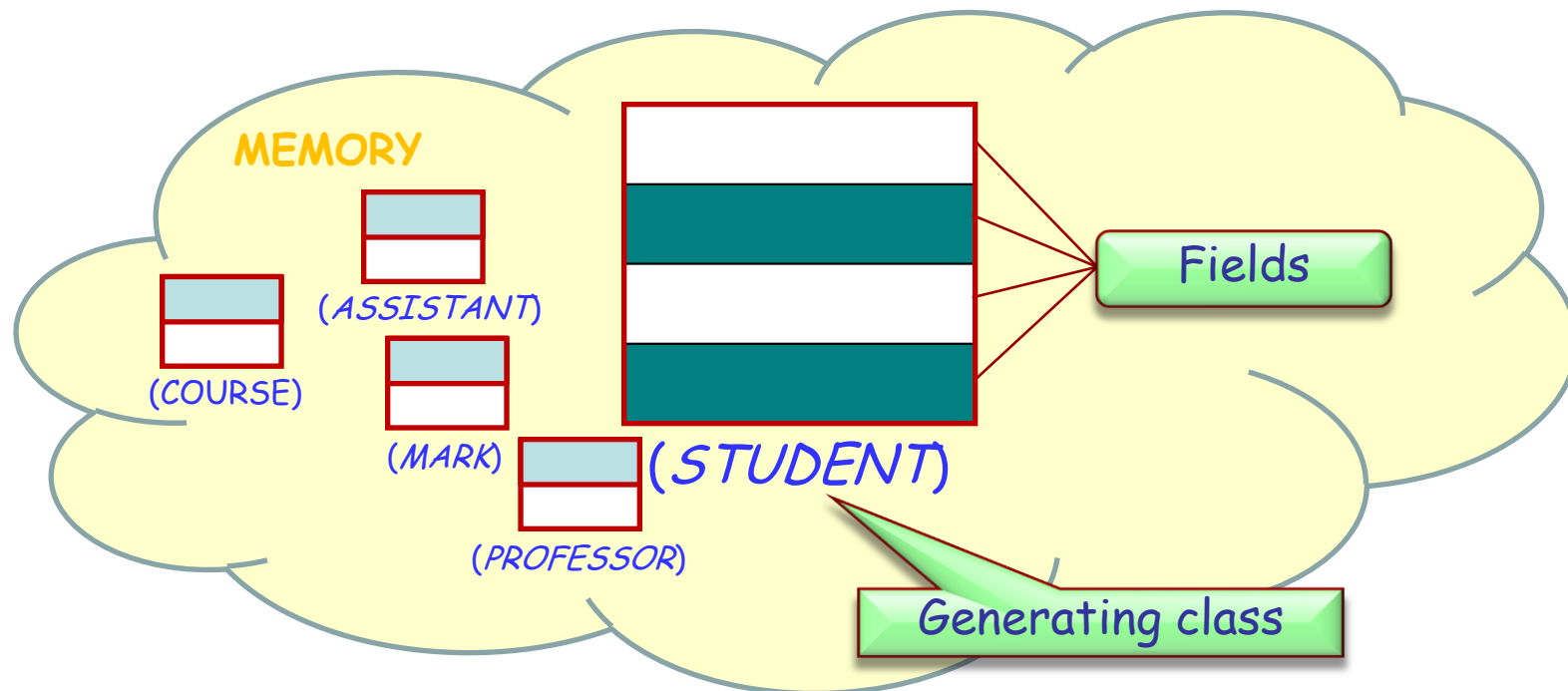
Entity vs. object



In the class text: **an entity**

joe: STUDENT

In memory, during execution: **an object**



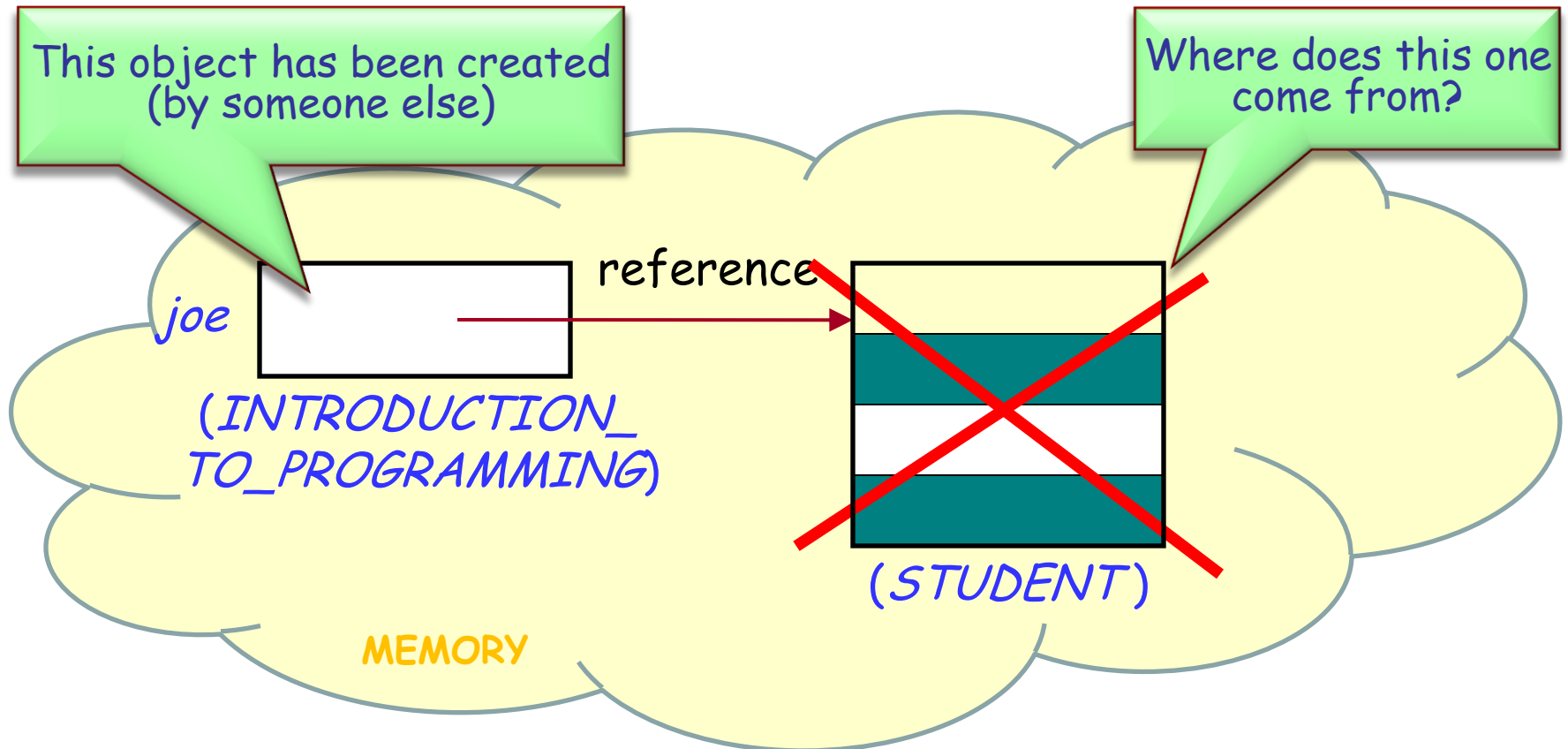


```
class
  INTRODUCTION_TO_PROGRAMMING
inherit
  COURSE
feature
  execute
    do
      -- Teach `joe' programming.
      -- ???
      joe.solve_all_assignments
    end
  joe: STUDENT
      -- A first year computer science student
end
```

Initial state of a reference



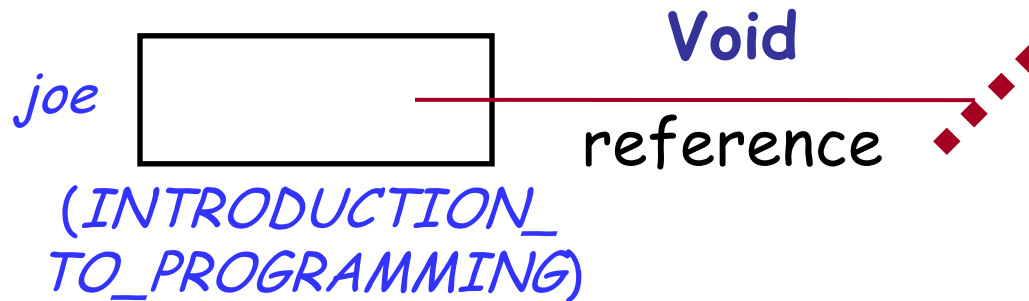
In an instance of *INTRODUCTION_TO_PROGRAMMING*, may we assume that *joe* is attached to an instance of *STUDENT*?



By default



Initially, *joe* is not attached to any object:
its value is a **Void** reference.



States of a reference



During execution, a reference can be:

- **Attached** to a certain object
- **Void**



States of a reference



- To denote a void reference: use **Void** keyword
- To create a new object in memory and attach it to x : use **create** keyword

create x

- To find out if x is void: use the expressions

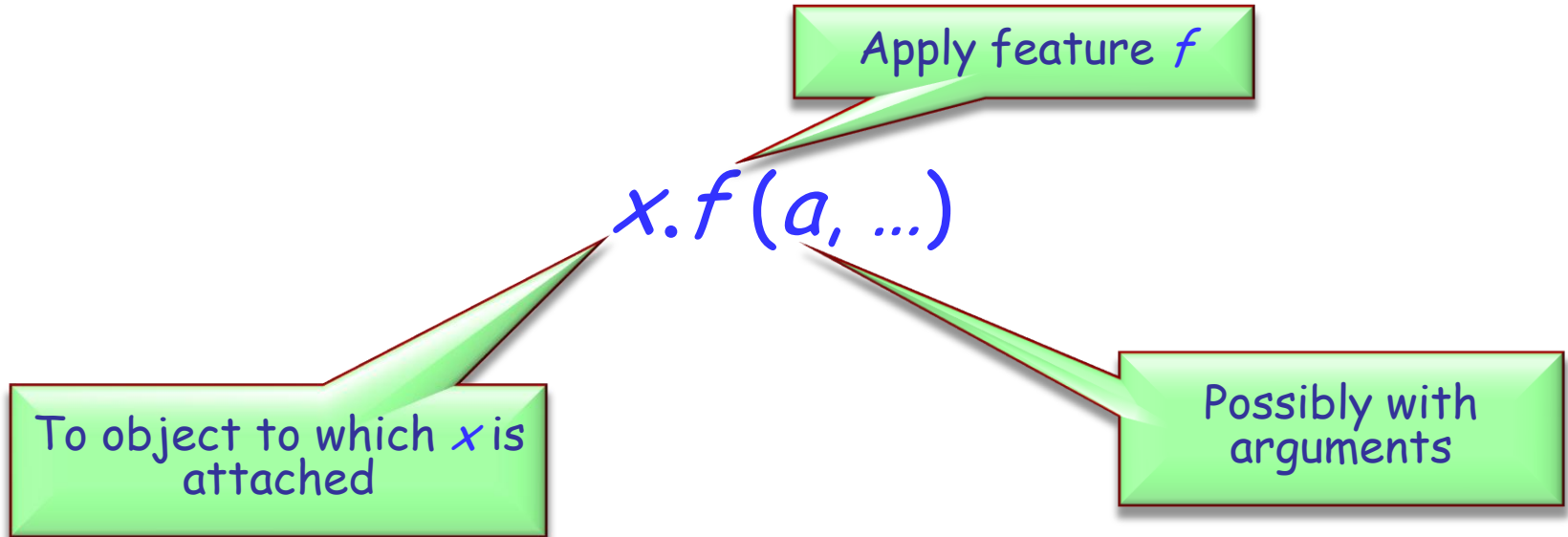
$x = \mathbf{Void}$ (true iff x is void)

$x \neq \mathbf{Void}$ (true iff x is attached)

Those mean void references!



The basic mechanism of computation is feature call



Since references may be void, x might be attached to no object

The call is erroneous in such cases!

Why do we need to create objects?



Shouldn't we assume that a declaration

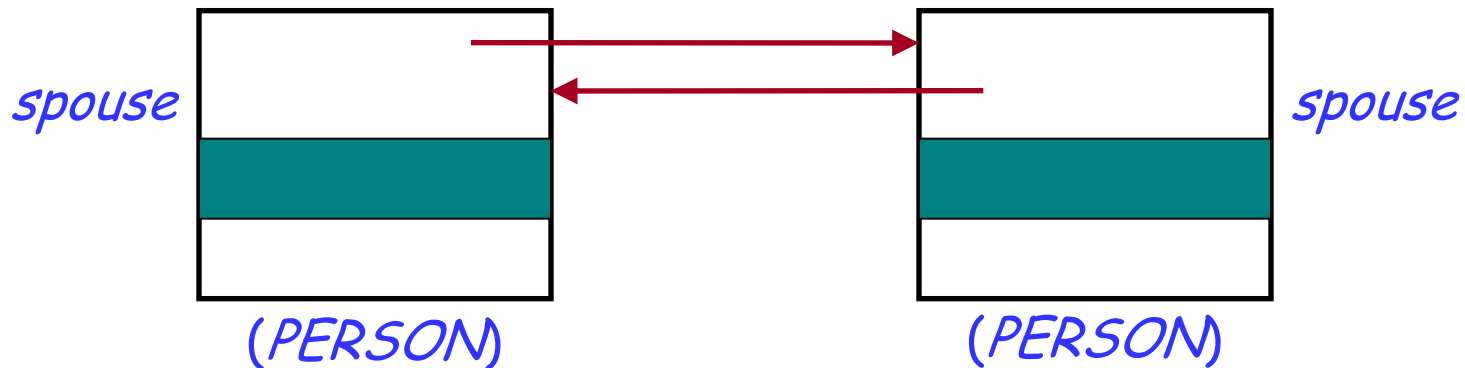
joe: STUDENT

creates an instance of *STUDENT* and attaches it to *joe*?

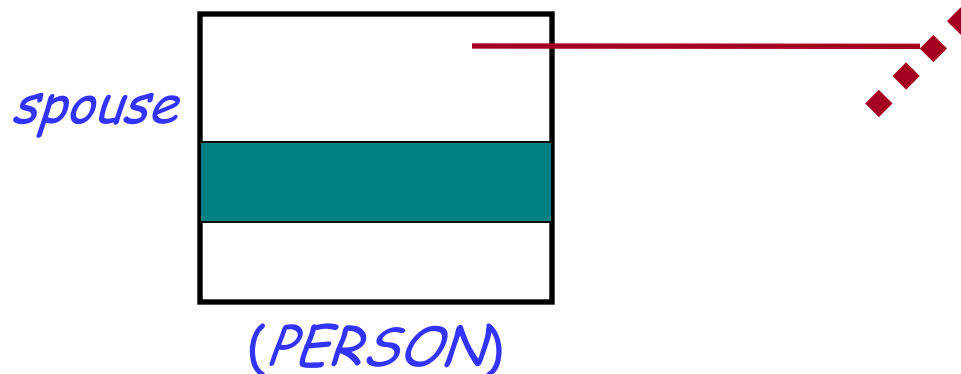
Those wonderful void references!



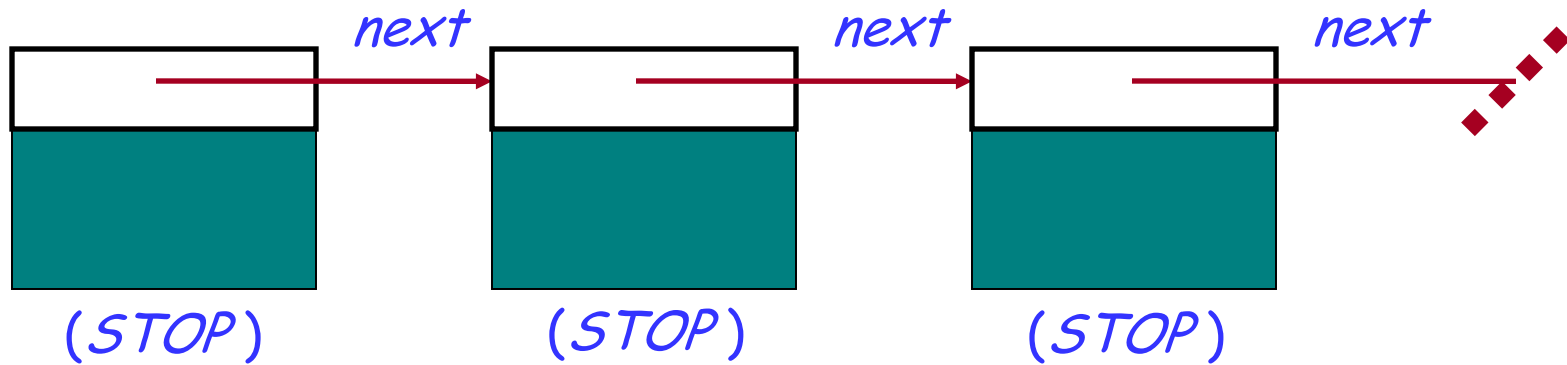
Married persons:



Unmarried person:



Those wonderful void references!



Last *next* reference is void to terminate the list.

- 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?

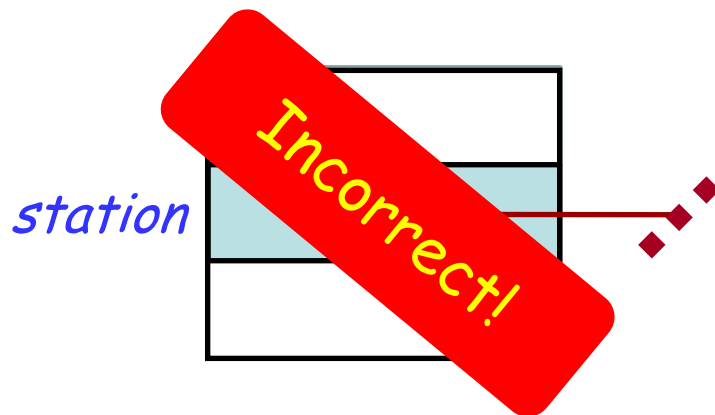
```
class STOP
```

```
...
```

```
    station: STATION
```

```
invariant
```

```
    station != Void
```



- Use creation procedure:

```
create stop1.set_station(Central)
```

STOP



class *STOP*

create

set_station

feature

station: STATION

-- Station which this stop represents

next: SIMPLE_STOP

-- Next stop on the same line

set_station(s: STATION)

-- Associate this stop with *s*.

require

station_exists: s /= Void

ensure

station_set: station = s

link(s: SIMPLE_STOP)

-- Make *s* the next stop on the line.

ensure

next_set: next = s

invariant

station_exists: station /= Void

end

List zero or more creation procedures

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

Is established by *set_station*

Object creation: summary



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 this procedure necessary?