ETH Zürich
Chair of Software Engineering

Einführung in die Programmierung
Introduction to Programming

Prof. Dr. Bertrand Meyer

Exercise Session 4

Today

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

Propositional Logic

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

Propositional Logic

Truth assignment and truth table

- Assigning a truth value to each propositional variable

P	Q	P implies Q
T	F	F
T	T	T
F	T	T
F	F	T

Tautology

- **True** for all truth assignments
  - P **or** (**not** P)
  - **not** (P **and** (**not** P))
  - (P **and** Q) **or** ((**not** P) **or** (**not** Q))

Contradiction

- **False** for all truth assignments
  - P **and** (**not** P)

Propositional Logic

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 φ = χ is a tautology

Tautology / contradiction / satisfiable?

Hands-On

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

Equivalence

Hands-On

Does the following equivalence hold? Prove.

(P implies Q) = (not P implies not Q) F

Does the following equivalence hold? Prove. T

(P implies Q) = (not Q implies not P)

P	Q	P implies Q	not P implies not Q	not Q implies not P
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

Useful stuff

De Morgan laws

not (P **or** Q) = (**not** P) **and** (**not** Q)

not (P **and** Q) = (**not** P) **or** (**not** Q)

Implications

P **implies** Q = (**not** P) **or** Q

P **implies** Q = (**not** Q) **implies** (**not** P)

Equality on Boolean expressions

(P = Q) = (P **implies** Q) **and** (Q **implies** P)

Predicate Logic

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

∃ h: Human | h.name = "Bill Gates"

All persons have a name

∀ p: Person | p.name /= **Void**

Some people are students

∃ p: Person | p.is_student

The age of any person is at least 0

∀ p: Person | p.age >= 0

Nobody likes Rivella

∀ p: Person | **not** p.likes (Rivella)

not (∃ p: Person | p.likes (Rivella))

Tautology / contradiction / satisfiable?

Hands-On

Let the domain of discourse be INTEGER

x < 0 **or** x >= 0

tautology

x > 0 **implies** x > 1

satisfiable

∀x | x > 0 **implies** x > 1

contradiction

∀x | x*y = y

satisfiable

∃y | ∀x | x*y = y

tautology

Semi-strict operations

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**.

  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**.

  list = **Void or else** *list.is_empty*

Strict or semi-strict?

Hands-On

- *a* = 0 **or** ______ *b* = 0
- *a* /= 0 **and** ______ *b* // *a* /= 0
- *a* /= **Void** **and** ______ *b* /= **Void**
- *a* < 0 **or** ______ *sqrt* (*a*) > 2
- (*a* = *b* **and** ______ *b* /= **Void**) **and** ______ **not** *a.name* .is_equal ("")

Assertions

Assertion **tag** (not required, but recommended)

Condition (required)

balance_non_negative: *balance* >= 0

Assertion clause

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
```

Why do we need contracts at all?

Together with tests, they are a great tool for finding bugs

They help us to reason about an O-O program at a class- and routine-level of granularity

They are executable specifications that evolve together with the code

Proving (part of) programs correct without executing them is what cool people are trying to do nowadays. This is easier to achieve if the program properties are clearly specified through contracts

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 invariants to classes *ACROBAT_WITH_BUDDY* and *CURMUDGEON*.

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
```

Class *CURMUDGEON*

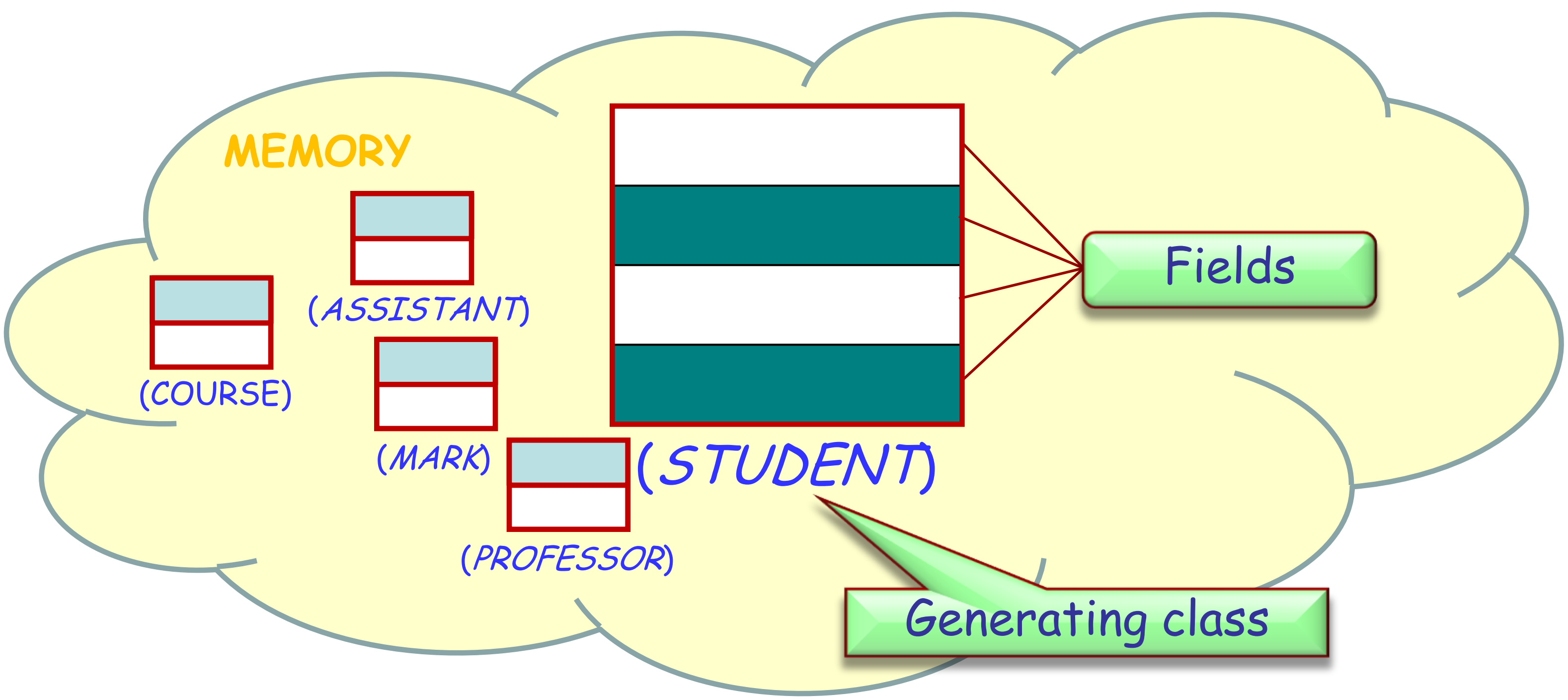
```
class
    CURMUDGEON

inherit
    ACROBAT
        redefine clap, twirl end

feature
    clap (n: INTEGER)
        do
            -- Say "I refuse".
        end

    twirl (n: INTEGER)
        do
            -- Say "I refuse".
        end
end
```

Entity vs. object

In the class text: **an entity**

joe: *STUDENT*

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

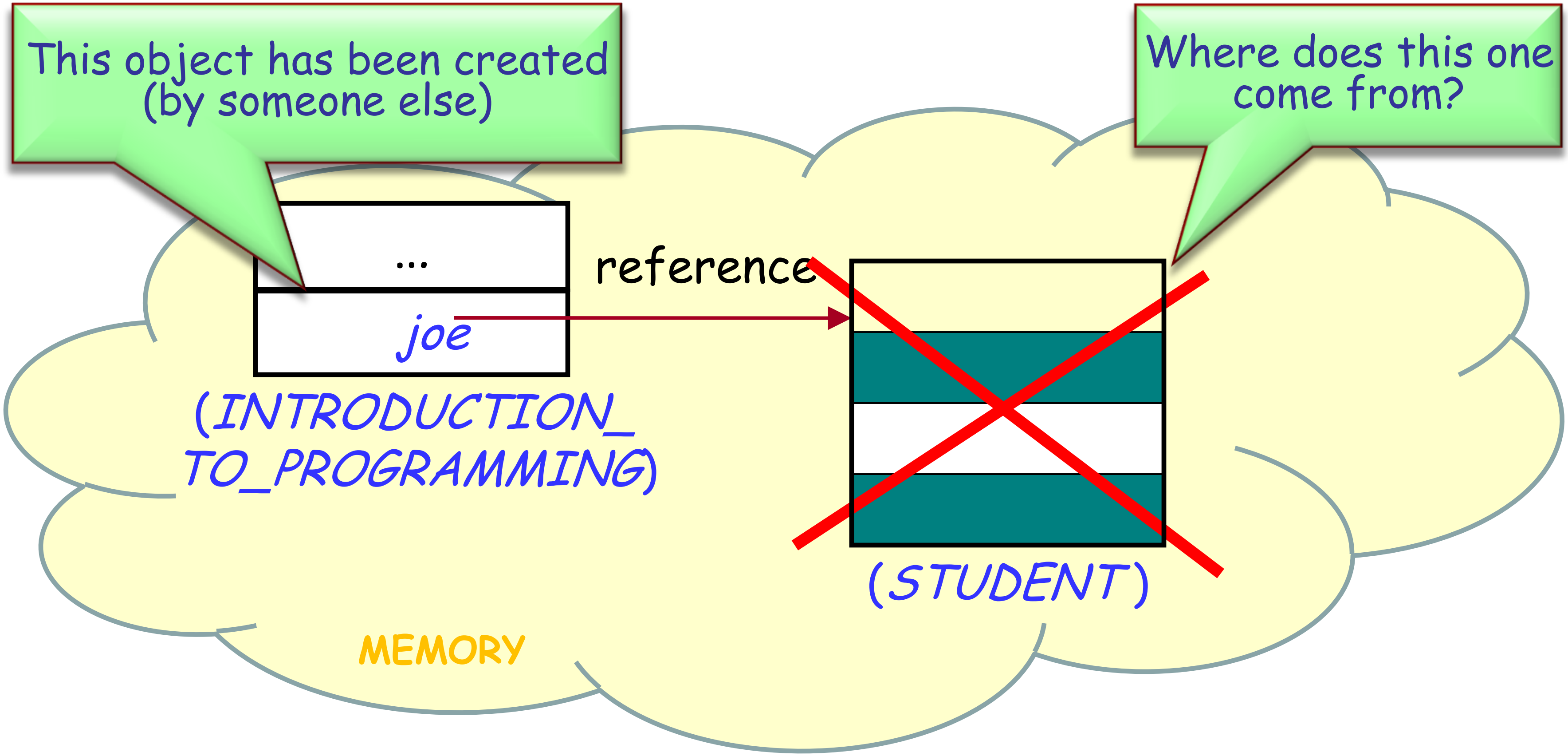


INTRODUCTION_TO_PROGRAMMING

```
class
        INTRODUCTION_TO_PROGRAMMING
inherit
        COURSE
feature
        execute
                        -- Teach `joe' programming.
                do
                        -- ???
                        joe.solve_all_assignments
                end

        joe: STUDENT
                -- A first year computer science student
end
```

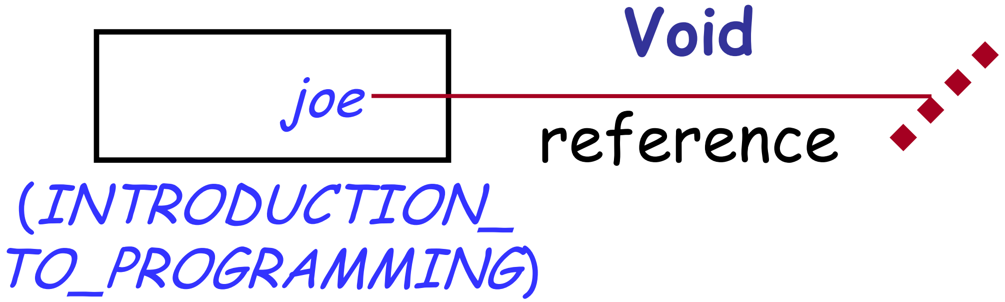
Initial state of a reference?

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



Default of references

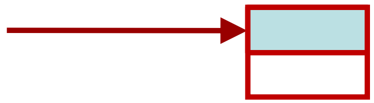
Initially, *joe* is not attached to any object:
its value is a **Void** reference.



States of an entity

During execution, an entity can:

- Be **attached** to a certain object



- Have the value **Void**

States of an entity

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

create *x*

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

x = **Void** (true iff x is void)

x /= **Void** (true iff x is attached)

Those mean void references!

The basic mechanism of computation is feature call

Apply feature *f*

$x.f(a, \ldots)$

To object to which *x* is attached

Possibly with arguments

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?

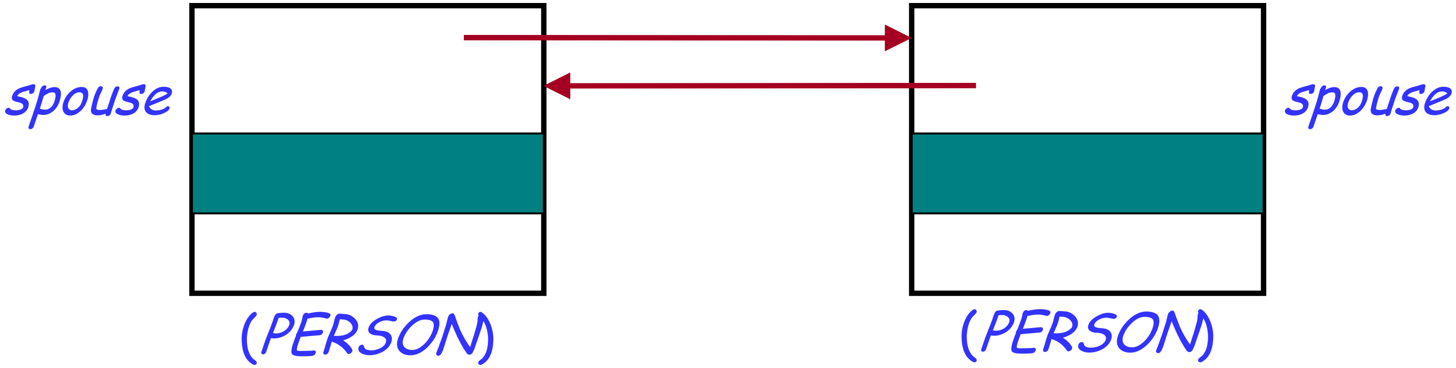
Shouldn't we assume that a declaration

joe: *STUDENT*

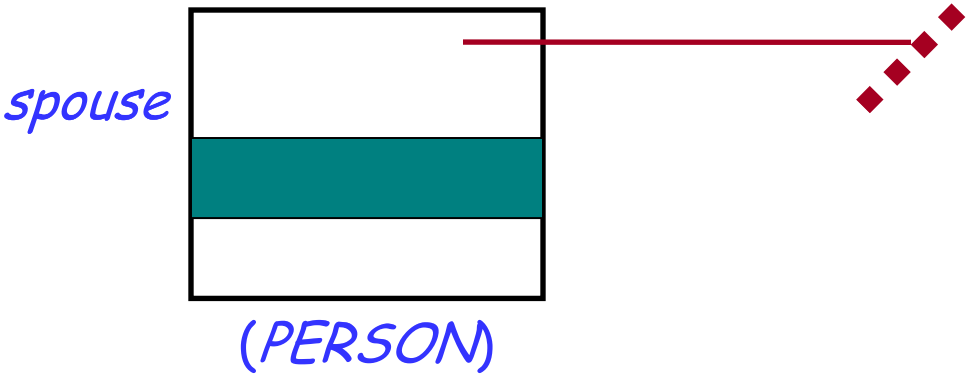
creates an instance of *STUDENT* and attaches it to *joe*?

Those wonderful void references!

Married persons:

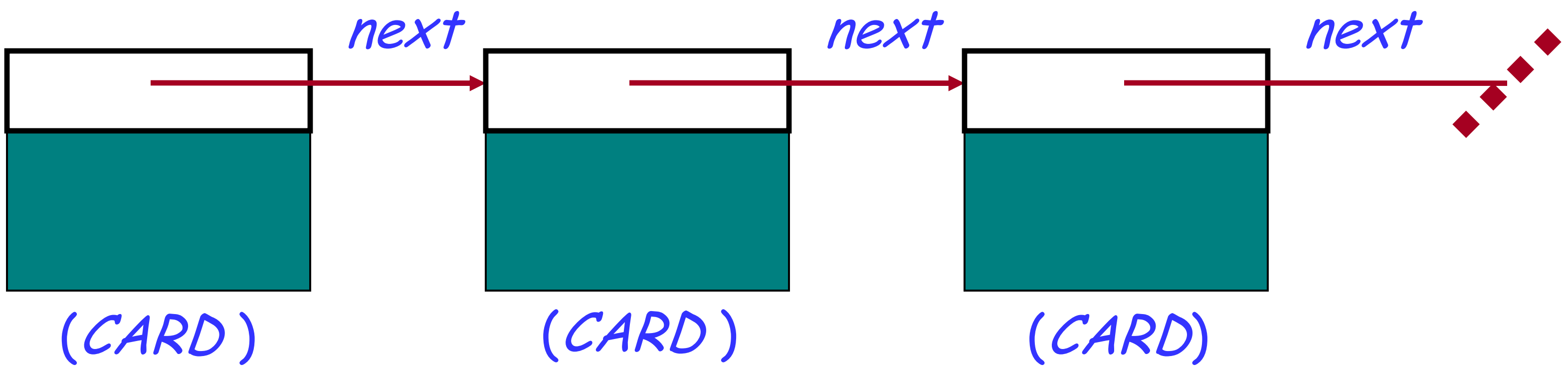


Unmarried person:



Those wonderful void references!

Imagine a DECK as a list of CARD objects

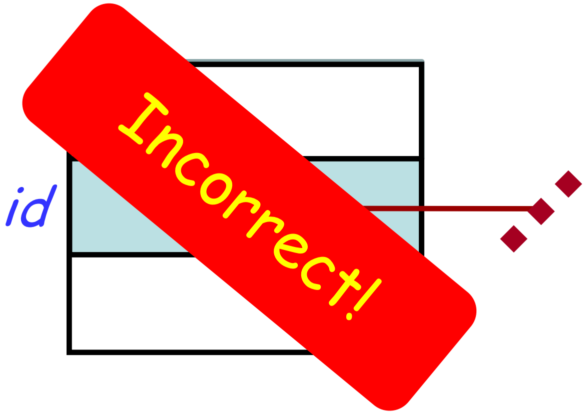


Last *next* reference is void to terminate the list.

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”)

STOP

```
Class CUSTOMER
create
    set_id
feature
    id: STRING
            -- Unique identifier for Current.

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

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: 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 the creation procedure necessary?

Meet Teddy