Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Esercitazione 3

How it all starts

Executing a system consists of

- creating a **root object**,

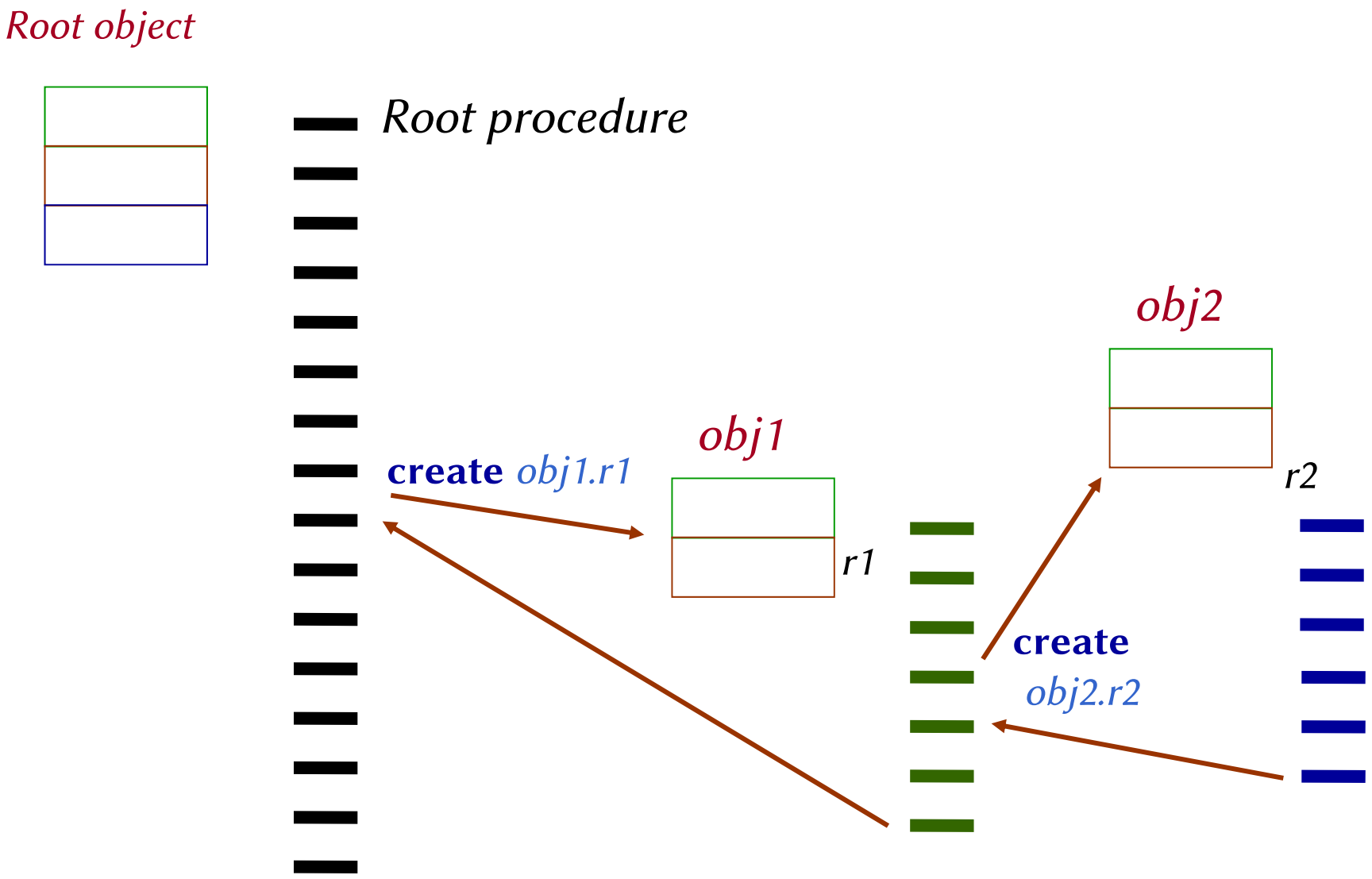
- which is an instance of a designated class from the system, called its **root class**,

- using a designated creation procedure of that class, called its **root procedure**.

Root procedure may:

- Create new objects
- Call features on them, which may create other objects
- Etc.

Executing a system



Specifying the root

How to specify the **root class** and **root creation procedure** of a system?

Use EiffelStudio

Project

-> Project Settings

-> Target

-> General

-> Root

Who are Adam and Eve?

- Who creates the first object?
  - The runtime creates a so called **root object**.
  - The root object creates other objects, which in turn create other objects, etc.
  - You define the **type (class) of the root object** in the project settings
    - `Project -> Project Settings -> Target: project -> Root`
- How is the root object created?
  - The runtime calls a creation procedure of the root object
  - You select this **creation procedure** of the root object
    - `Project -> Project Settings -> Target: project -> Root`
  - The application exits at the end of this creation procedure

The current object

At every moment during execution, there is a **current object**, on which the current feature is being executed

Initially it is the root object. Then:

- An unqualified call such as *set* (*u*, *v*) applies to the current object (i.e., to **Current**, usually omitted)
- A qualified call such as *x*.*set* (*u*, *v*) causes the object attached to *x* to become the current object. After the call the previous current object becomes current again

Acrobat game

Hands-On

- We will play a little game now.
- Objects will have different roles.

There is an acrobat object

- When asked to **Clap**, it will be given a number and it has clap its hands that many times.
- When asked to **Twirl**, it will be given a number and it has to turn completely around that many times.
- When asked for **Count**, it has to announce how many actions it has performed. This is the sum of the numbers that have been given to date.

Pseudocode

«Pseudocode» means sentences in natural language, which are not yet written code

Example:

```
-- "Clap n times"
```

We write pseudocode as comments and between quotation marks

Style: when the actual code is written, it is a good idea to keep the pseudo-code in the program as a regular comment

There is an *ACROBAT* – first version

```
class
    ACROBAT

feature
    clap (n: INTEGER)
        do
            -- "Clap `n' times and adjust `count'."
        end

    twirl (n: INTEGER)
        do
            -- "Twirl `n' times and adjust `count'."
        end

    count: INTEGER
            -- "Total # of times clapped or twirled."
end
```

There is an *ACROBAT* – adding invariants

```
class
    ACROBAT
feature
    clap (n: INTEGER)
            -- Clap `n' times and adjust `count'.
        require   n>0
        do
        -- to be completed
        ensure    count = old count + n
        end
    twirl (n: INTEGER)
            -- Twirl `n' times and adjust `count'.
        require   n>0
        do
        -- to be completed
        ensure    count = old count + n
        end
    count: INTEGER
            -- Total # of times clapped or twirled.
end
```

We need a root object

- It got created by the runtime.
- It is executing the first feature.

Here is the *DIRECTOR*

```
class
    DIRECTOR

create
        prepare_and_play

feature
        prepare_and_play
            do
                -- See following slides.
            end
```

Here is the root object (version 1)

```
prepare_and_play
        local
            mario, luigi, piero: ACROBAT



        do
            create mario
            create luigi
            create piero
            mario.clap (3)
            luigi.clap (4)
            piero.clap (5)
            mario.twirl (3)
            luigi.twirl (4)
            piero.twirl (5)
            mario.count
            luigi.count
            piero.count
        end
```

Allow objects to give feedback to what happens to them by printing it.
For example: print("%N mario.count = "); print(mario.count)

There is an *ACROBAT* – implementation

Open EiffelStudio,

copy-paste the code,

and complete it !

There are acrobat and copycat objects

- Each acrobat object will have another object as its Copycat.
  - N.B. asymmetric relation!
- When asked to **Clap**, the acrobat will be given a number. It has to clap its hands that many times and pass the same instruction to its Copycat.
- When asked to **Twirl**, the acrobat will be given a number. It has to turn completely around that many times and pass the same instruction to its Copycat.
- When asked for **Count**, the acrobat will ask its Copycat and answer with the number it answers

There is a *COPYCAT* – first version

```
class
    COPYCAT

feature
    clap (n: INTEGER)
        do
            -- "Clap `n' times and adjust `count'."
        end

    twirl (n: INTEGER)
        do
            -- "Twirl `n' times and adjust `count'."
        end

    count: INTEGER
            -- "Total # of times clapped or twirled."
end
```

There is a *COPYCAT* – adding invariants

```
class
    COPYCAT
feature
    clap (n: INTEGER)
            -- Clap `n' times and adjust `count'.
        require   n>0
        do
        -- to be completed
        ensure    count = old count + n
        end
    twirl (n: INTEGER)
            -- Twirl `n' times and adjust `count'.
        require   n>0
        do
        -- to be completed
        ensure    count = old count + n
        end
    count: INTEGER
            -- Total # of times clapped or twirled.
end
```

There is an *ACROBAT* – second version (1)

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

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

    count: INTEGER
        do
            -- "Ask copycat and return his answer."
        end
end
```

This query is no more an attribute but a routine

There is an *ACROBAT* – second version (2)

class

ACROBAT

feature

buddy: **detachable** *COPYCAT*

To be explained later

pair (p: COPYCAT)

do

-- Remember `p' being the copycat.

end

There is an *ACROBAT* – ver.2, invariants(1)

```
class
    ACROBAT
feature
    buddy: detachable COPYCAT
    pair (p: COPYCAT)
            -- Remember `p' being the copycat.
        require   p /= Void
        do
        -- to be completed
        ensure    buddy = p
        end
```

There is an *ACROBAT* – ver.2, invariants(2)

```
    clap (n: INTEGER)
            -- Clap `n' times and forward to copycat.
        require   n>0
        do
        -- to be completed
        ensure    count = old count + n
        end

    twirl (n: INTEGER)
            -- Twirl `n' times and forward to copycat.
        require   n>0
        do
        -- to be completed
        ensure    count = old count + n
        end

    count: INTEGER
            -- Ask copycat and return his answer.
        do
        -- to be completed
        end

end
```

Here is the root object (version 2)

```
prepare_and_play
        local
            mario, luigi, piero: ACROBAT
            mariuccio, luigino: COPYCAT


        do
            create mariuccio
            create mario
            mario.pair(mariuccio)
            create luigi
            create piero
            create luigino
            luigi.pair(luigino)
            mario.clap(3)
            luigino.twirl (2)
            luigi.clap (7)
            piero.twirl (luigi.count)
        end
```

Allow objects to give feedback to what happens to them by printing it.
For example: print("%N mario.count = "); print(mario.count)

There are *ACROBAT* and *COPYCAT* – implementation

Open EiffelStudio,

copy-paste the code,

and complete it !

Concepts seen DA AGGIUSTARE

Eiffel	Game
Classes with Features	Telling person to behave according to a specification
Objects	People
Interface	What queries What commands
Polymorphism	Telling different people to do the same has different outcomes
Command Call	Telling a person to do something
Query Call	Asking a question to a person
Arguments	E.g. how many times to clap

Concepts seen

Eiffel	Game
Inheritance	All people were some kind of ACROBAT
Creation	Persons need to be born and need to be named
Return value	E.g. count in ACROBAT_WITH_BUDDY
Entities	Names for the people
Chains of feature calls	E.g. partner1.buddy.clap (2)