Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

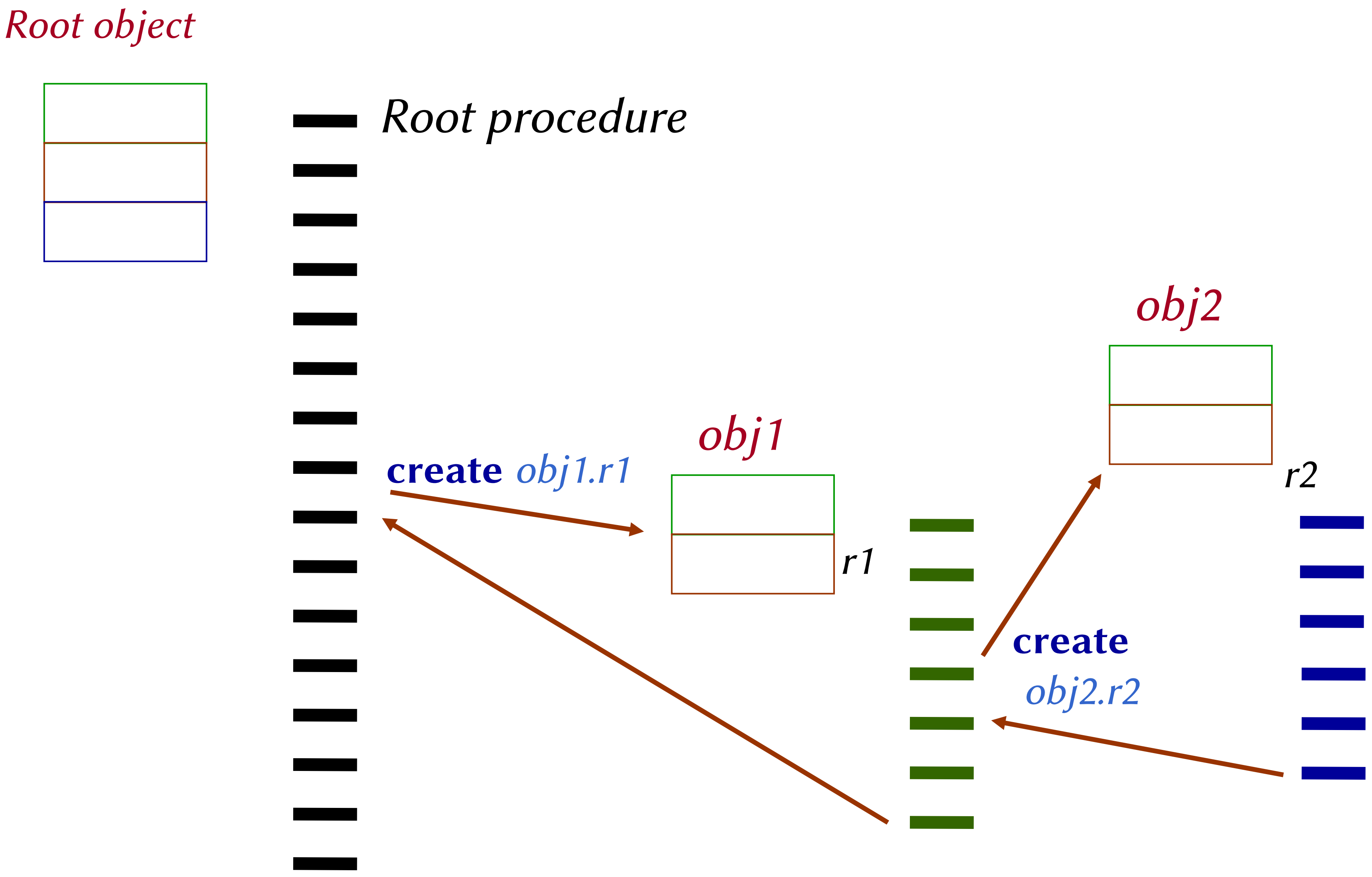
Esercitazione 2

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**.

- The runtime creates an instance called **root object** of the **root class**
- The runtime calls the **creation procedure** of the root object
- During the execution of the creation procedure the root object may create other objects, which in turn create other objects, etc.
- The application exits at the end of the creation procedure of the root object

Executing a system



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

Specifying the root

- How to specify the **root class** and **root creation procedure** of a system?
  - Automatically done by the system when a new project is created: you can choose their names
  - Names of root class and root creation procedure can be changed through “Refactor-> Rename” command in the right-click menu of the name

Check in Eiffel Studio under **Project** menu

-> Project Settings

-> Target

-> General

-> Root

EiffelStudio

A first project: a bank account

Practice debugging: the most important issue!

Remember: Eiffel Naming Conventions

- Full words, no abbreviations (with some exceptions)
- Locals and arguments share namespace with features
  - Name clashes arise when a feature is introduced, which has the same name as a local (even in parent)
- To prevent name clashes:
  - Locals are prefixed with **l_**
  - Arguments are prefixed with **a_**
- But exceptions may exist…

Remember: Editor shortcuts

- Auto-completion (CTRL+Space)
- Class name completion (CTRL+SHIFT+Space)
- Block indenting or unindenting (TAB and SHIFT+TAB)
- Block commenting or uncommenting (CTRL+K and SHIFT+CTRL+K)
- Quick search features (first CTRL+F to enter words then F3 and SHIFT+F3)
- Pretty printing (CTRL+SHIFT+P)
- Editor line numbering (Tools -> Preferences -> check "Include Values" -> Search -> Filter insert 'line' -> Editor.General.Show line numbers -> double click on 'False'
- In EiffelStudio: Tools ⟶ Preferences ⟶ Shortcuts...

An example: modeling a bank account

What do we need to represent?

DATA:

- the fact that the account is open or closed
- which is its balance

OPERATIONS:

- open the account
- close it
- deposit an amount on it
- withdraw an amount from it
- know its balance

FUNDAMENTAL RULE OF SW DEVELOPMENT:

Enable people reading the code to understand it

Features: Exercise

Hands-On

- Assume class *BANK_ACCOUNT* defines the following operations:
  (will be developed in the next practice session)
  - *deposit (i: INTEGER)*
  - *withdraw (i: INTEGER)*
  - *close*

- If b: *BANK_ACCOUNT* (b is an instance of class *BANK_ACCOUNT*) which of the following feature calls are possible:
  - *b.deposit (10)* ✓
  - *b.deposit* ✗
  - *b.close* ✓
  - *b.close (“Now”)* ✗
  - *b.open* ✗
  - *b.withdraw (100.50)* ✗
  - *b.withdraw (0)* ✓

Exercise: query or command?

Hands-On

- To know the balance of a bank account
- To withdraw some money from a bank account
- To know who is the owner of a bank account
- To know the clients of a bank whose deposits are over 100,000 euros
- To change the account type of a client
- To know how much money can a client withdraw at a time
- To set a minimum limit for the balance of accounts
- To know whether Bill Gates is a client of Credit Suisse

A first attempt for BANK_ACCOUNT

Hands-On

```
feature -- state
  open: BOOLEAN
  -- the account is open
  balance: INTEGER
  -- how much money is in the account

feature -- operation
  withdraw (a_sum: INTEGER)
  -- withdraw `a_sum' from the account
  deposit (a_sum: INTEGER)
  -- deposit `a_sum' from the account

feature -- management
  close
  -- close the account
  start
  -- open the account
```

Debugger: setup

- Setting and unsetting **breakpoints**
  - An efficient way consists of dropping the feature you want the breakpoint in into the context tool.
  - Alternatively, you can select the flat view.
  - Then click on one of the little circles in the left margin to enable/disable single breakpoints.
- Use the toolbar debug buttons to enable or disable all breakpoints globally.

- The system must be melted/frozen (finalized systems cannot be debugged).

Debugger: run

- Run the program by clicking on the Run button.
- Pause by clicking on the Pause button or wait for a triggered breakpoint.
- Analyze the program:
  - Use the **call stack pane** to browse through the call stack.
  - Use the **object tool** to inspect the current object, the locals and arguments.
- Run the program or *step over* (F10) / *into* (F11) the next statement, or *out* (↑F11) of the current one
- Stop the running program by clicking on the Stop button.