Università degli Studi di Bologna
Facoltà di Ingegneria

Principles, Models, and Applications for Distributed Systems M

Lab assignment 0 (worked-out)
Java Multithreading

Luca Foschini

Some Basic Models

Often, **local patterns, modes, strategies are useful**

models and paradigms ***static/dynamic***

models and strategies ***proactive/reactive***

execution *models* ***for the system***

- **monouser/multiuser**
- **single-processor/multi-processor**

models for ***active execution***

- **processes/objects replication**

models for ***allocation entities***

- **processes/objects static/dynamic decisions**

STATIC and DYNAMIC MODELS

Static models / Dynamic models

The number of users of an application is predefined

Users can be **added / removed**

The number of processes in one application is predefined

The number of processes can **change during execution**

The maximum number of nodes is predefined

The participating processors may also **greatly increase**

The number of customers of a service is predefined

The number of services (throughput) is not predefined

The servants are known and predefined

Services (servers) have to change in the number and type

Intermediate servers catalog and activate servants

EXECUTION MODEL

Execution of one or more applications

Monouser: the use of a dedicated system is typical of prototype phases

Multiuser: several users can form a better mix of the executable entities / systems

workstation model

preferably using local resources

processor pool model

resources are used in a transparent way according to their use and availability

RESOURCE MODEL

Resources during execution

Processes: active entities capable of performing

- **local actions** on their environment and
- **distributed communication actions** with other processes by using *shared memory and message passing*

Using *external data* to the processes themselves (low confinement)

Object entities as an abstraction, the ability to

- enclose and hide **internal resources** (data abstraction) with **external visibility** of **operations**
- act on internal resources upon external operation requests

Passive objects: **data abstractions** run by external entities

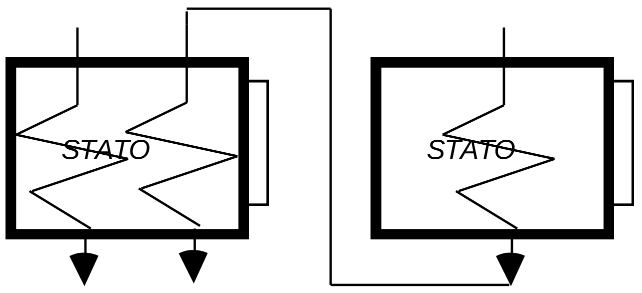
Active objects: **entities capable** of **execution** and **scheduling**

EXECUTION and OBJECTS

Passive objects

The **PASSIVE object** models predict that processes (*external to the objects themselves*) can run **across the objects** and execute methods

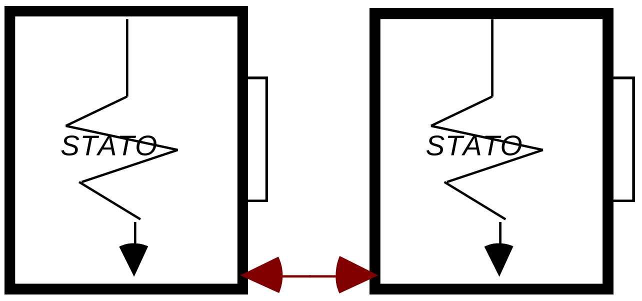
- model with a low confinement
- little protection
- interference between different processes



Active objects

The **ACTIVE object** models are more close in terms of execution: the external processes are not allowed to enter, but only to submit requests. Who run are **only internal processes** inside active object

- protected objects
- completely (self-)determined



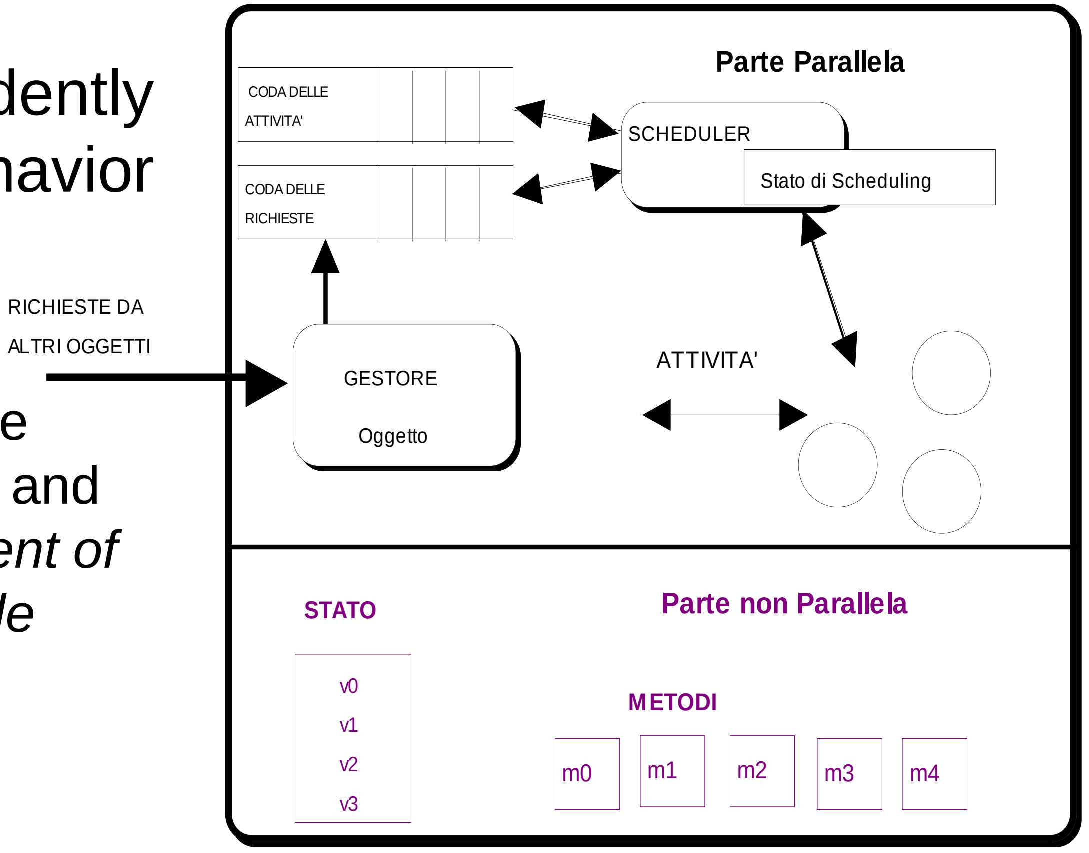
richiesta esplicita e risposta

ACTIVE OBJECTS

Active objects

Active Objects independently decide their internal behavior and hide it by confining it

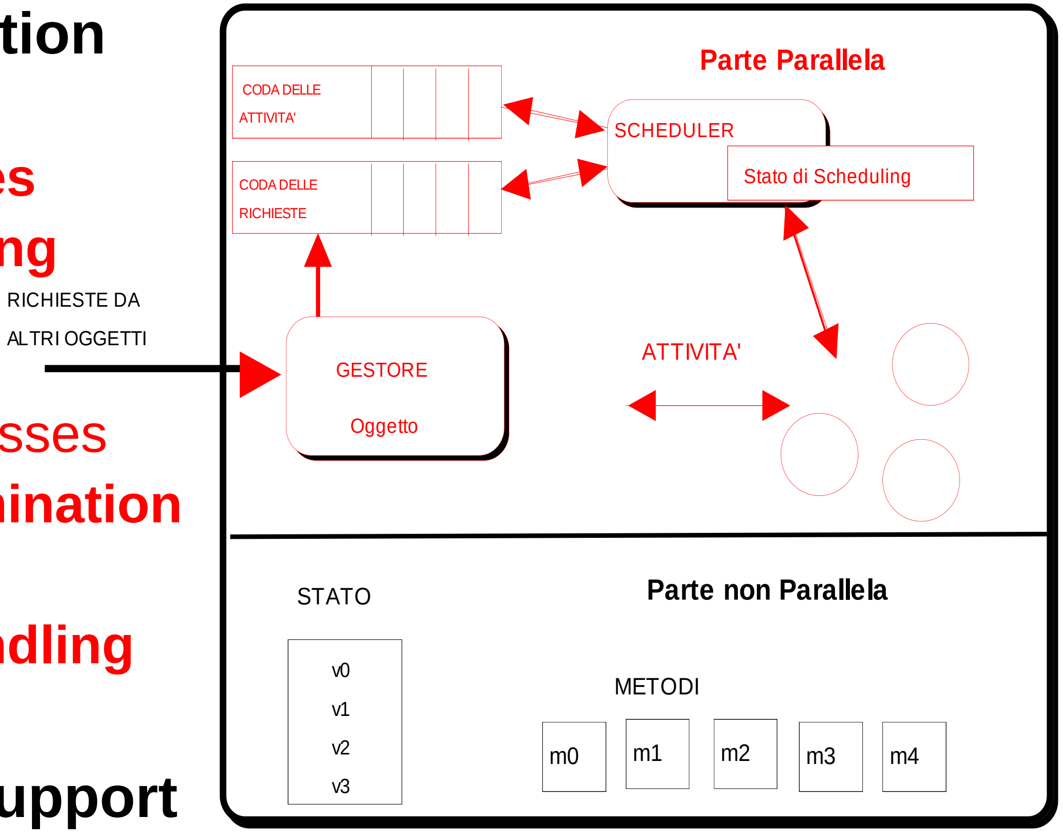
Each **active object** encloses the needed **concurrency capacity** and defines *its own local management of concurrency*, preventing *possible interferences*



ACTIVE OBJECTS

Support **automatically** adds all the **functions** for **self-determination**

- The tail of **external demands**
- The tail of the **internal activities**
- The realization of the **scheduling policy**
- The **management** of the **activation** of internal processes
- The **management** of their **termination**
- The delivery of **results**
- The **management** of **error handling**



Mechanisms given by the support
Policies left to the user

OBJECTS AND CLASSES - digression

Passive objects as **data** and **methods**

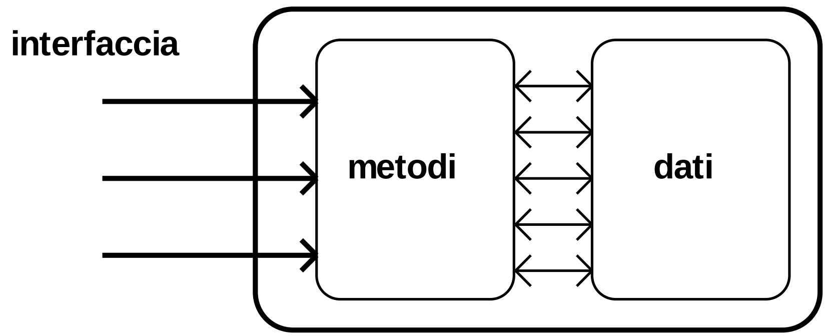
Methods are invoked by the interface and made visible from the outside

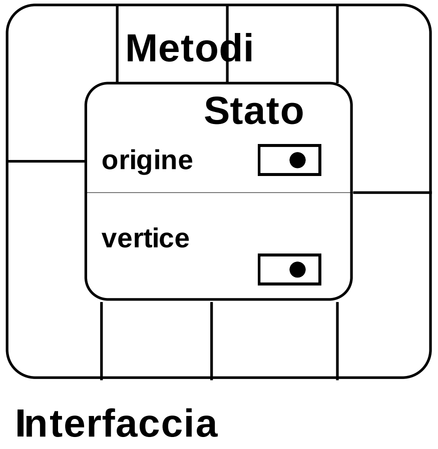
Data is typically protected and not visible from the outside

Data (attributes) are

- **Primitive data** (e.g., an integer variable called **origin**)

- **References** to other objects (e.g., a typed link to another object)





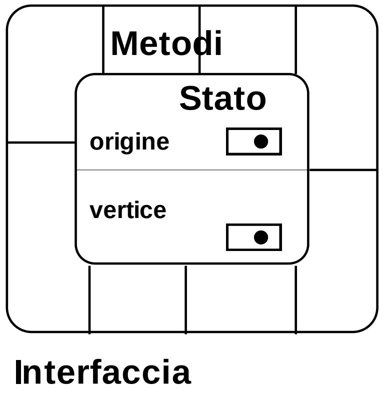
CLASSES (digression: not too much)

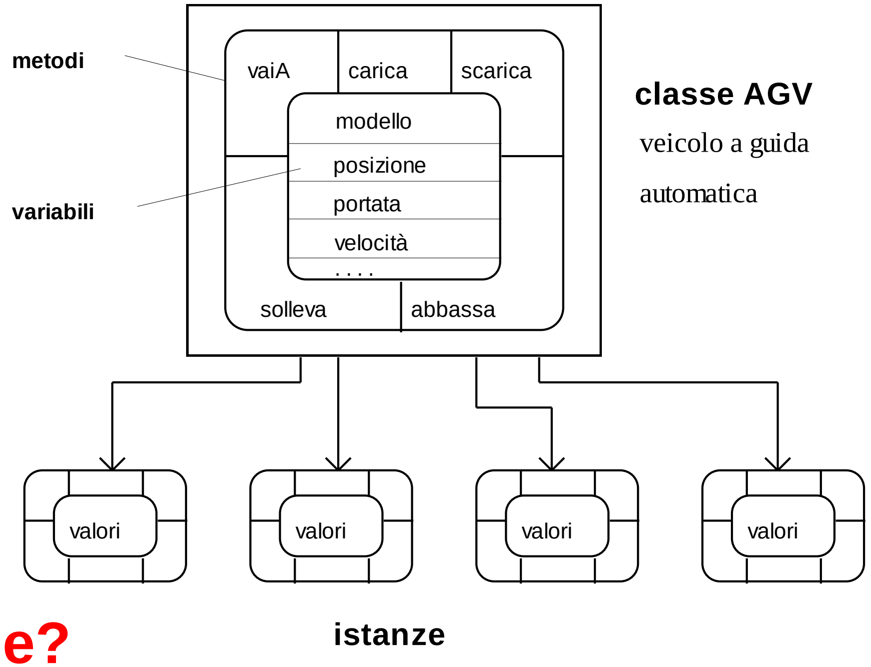
In **class-based systems**, the classes

- contain methods in a unique way for all instances

- specify for each instance internal data/types (primitive or not)

if the types are **not primitive** but **other objects**, the class specifies what class the references must be





Are there classes at run-time?

CLASSES at RUNTIME

The classes are **present at run-time** and **loaded dynamically**

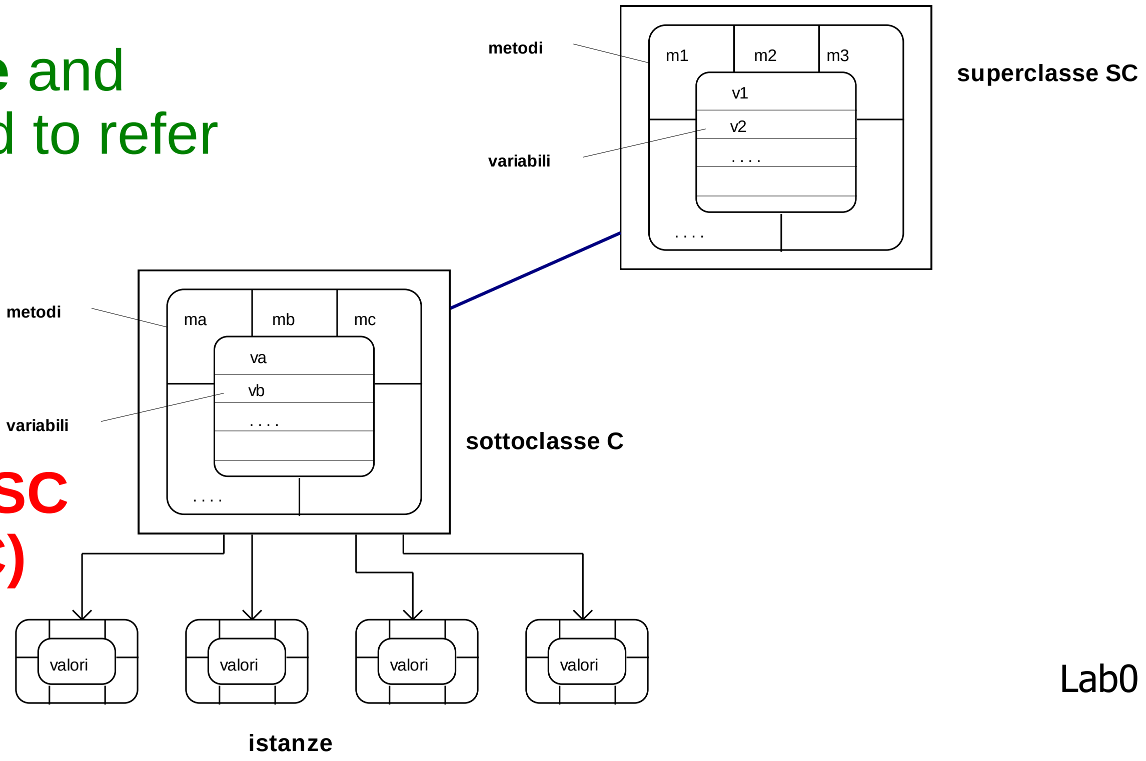
- Are loaded into a heap to find the methods and static variables at run-time

Also objects are loaded dynamically and refer to their class for their behavior

The classes are often **related by inheritance** and an instance is expected to refer to **one class** and **many superclasses**

(you can call

both m1 exposed by SC and ma exposed by C)



CLASSES vs INTERFACES

In modern architectures, the **interfaces** are the contract of interaction (in Java are programming-level entities)

The **classes** describe the specific implementations (and are unique entities describing the behavior of instances)

OO languages often provide **inheritance for both**

Multiple inheritance (multiple parents) - natural for interfaces

Simple inheritance (single parent) - typical for classes

In Java, **interfaces** are organized **in graphs** – and classes may implement multiple interfaces - but **classes are in the inheritance chain**, greatly simplifying support for instances:

- An instance requires only **the loading of a class** (defined) and **all classes in inheritance**
- Each method is dynamically bound simply **by passing through the chain of classes from the definition class** (ease in the production of static code and dynamic support)

MULTIPLE INHERITANCE

In OO languages with multiple inheritance between classes (C++, VBasic,... and branches), the dynamic exploration is on a graph of classes and, in these systems, the method to execute the search becomes more complex

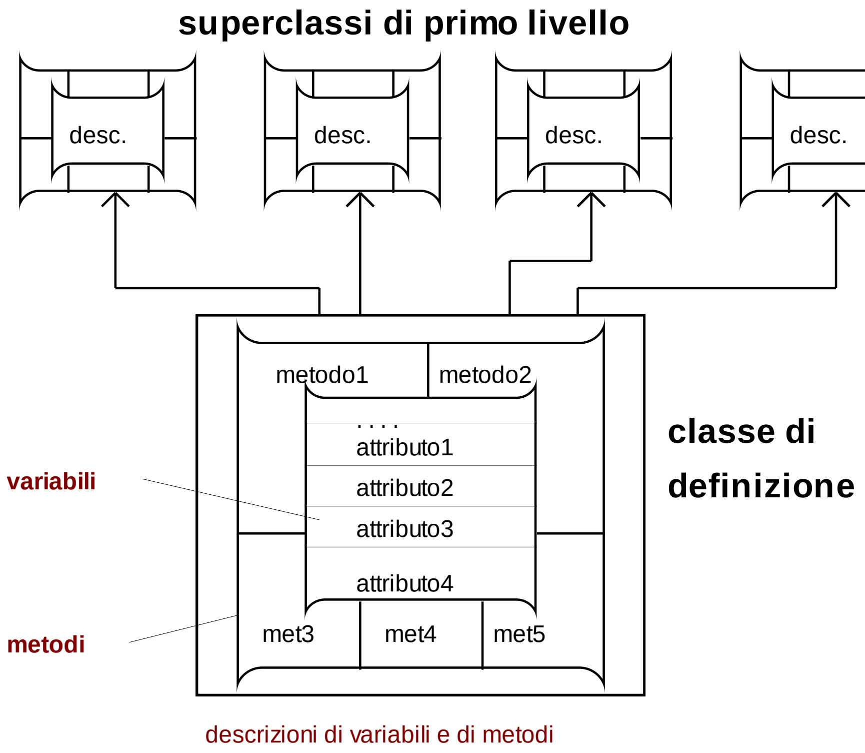
There are several classes to match and in the order established by the inheritance

It applies:

Overriding

and **in order inheritance**

in concentrated systems



by-REFERENCE SEMANTICS

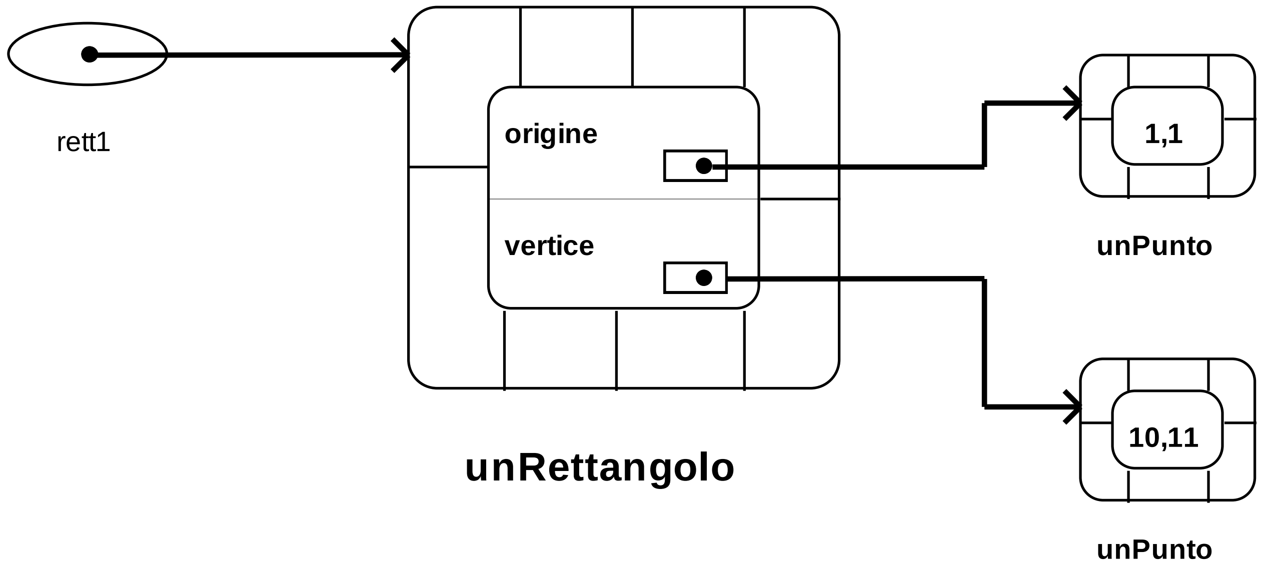
The objects do not contain other objects

In object systems, non-primitive attributes have a **by-reference** semantic: they contain only references

Through a **variable** (with a type), it is possible to refer another instance

Changing variable value it is possible to refer another instance afterwards

Passive objects are arranged in a graph of references between themselves at runtime



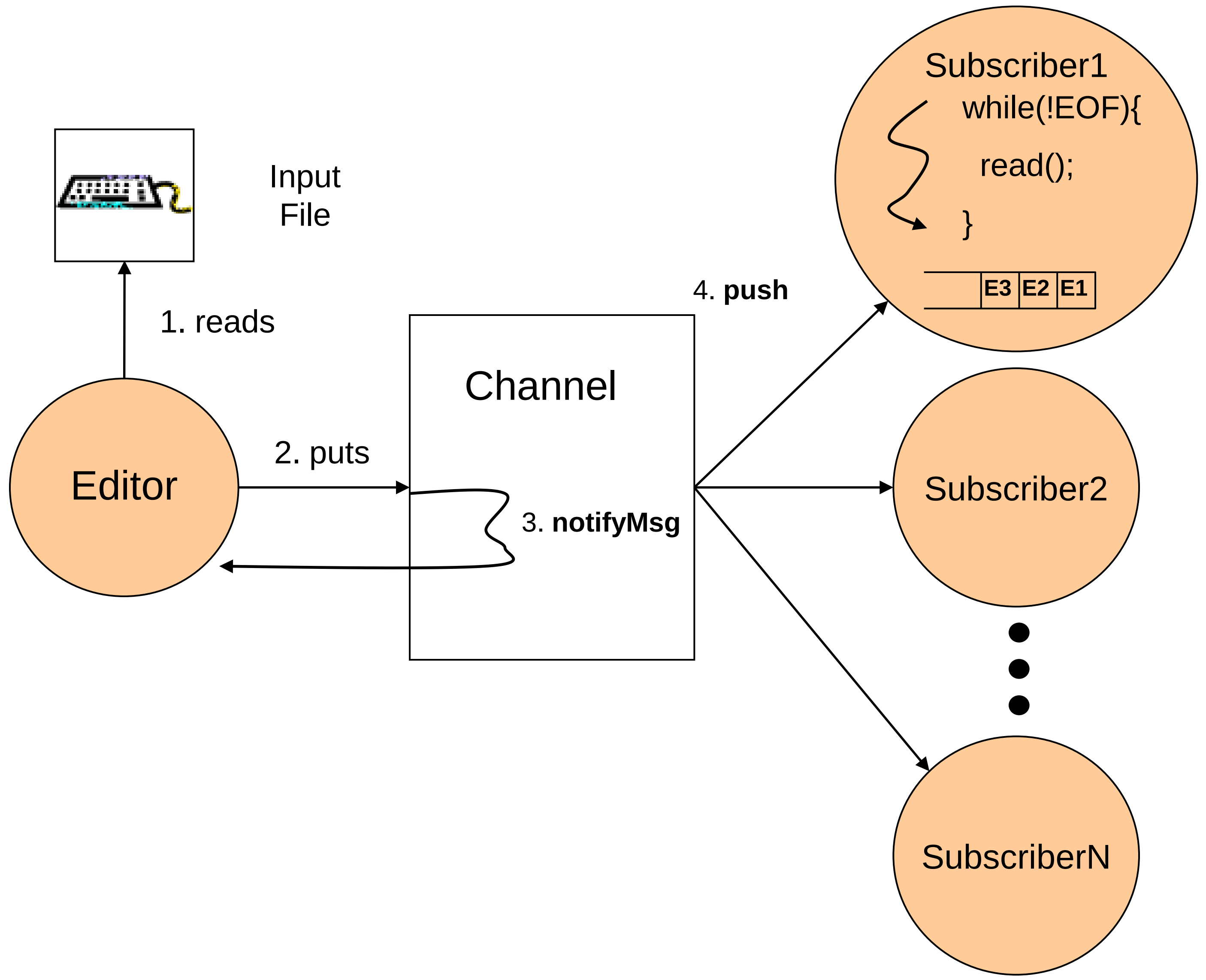
Lab assignment: Event Model exercise

The **event model architecture** involves the interaction of at least **two types of processes**:

- the **publisher** is the process that produces the events;
- the **subscriber** is the process (or processes, since several interested subscribers may register with the same publisher) that registers with the publisher to receive events.

When a new event occurs, the **publisher** notifies it to **all subscribers registered** with him and returns to listening to a new event.

Reference architecture



Some more details…

In the worked-out implementation shown here, in addition to the two entities specified above (called in the following **Editor** and **Subscriber**), we introduce a third entity called **Channel**.

The Channel **is not a process**, but a **passive object** that maintains the list of all Subscribers and implements the notification method, actively executed by the Editor as shown in the previous figure.

In our system, the **events** are **strings** cyclically read from the input console (keyboard) from the Editor process.

A **small test program** to verify all implemented entities will also be shown.

Java: Objects and Threads

Java provides two ways to implement threads:

- as a **subclass** of **`Thread`** class;

- as a **class** that implements the **`Runnable`** interface.

In any case, let us note that a single Java object that implements a thread has a “double nature” of:

- **passive object**: all methods except the **`run()`** method;
- **active object**: application method specified within the **`run()`** method.

Single entity specification: Editor

The **Editor** is a cyclic **process** that reads from the console (namely, the input file), **until the end of the input file**, the events represented by strings edited by the user, and then places them in the Channel.

The process behaves as a **filter** that **consumes all its input** until the end of file; it has also to signal other processes of the termination event.

Single entity specification: Subscriber

Subscribers are the processes that receive events and print them on screen; they execute the following pseudo-code:

1. processing (waiting for a random time between 2 and 5 sec.)
2. subscription to the Channel;
3. execution of a cycle of receive events until they are received and printed on-screen **N events** (N is an integer randomly chosen between 2 and 6), or to an End Of File (**EOF**);
4. de-signing and termination of the Channel.

Subscribers are **active entities** with a **limited lifetime** and perform a loop in their bodies until their termination.

Subscriber (more details)

Each **Subscriber** internally maintains a queue (e.g. realized as a Java Vector) to save the events sent by the channel and it implements the following methods:

- `read`: private method used to read the first event from the queue of upcoming events. The method is blocking: if the message queue is empty, the subscriber is blocked.

  → **NOTE:** the effect of read execution is to remove the event from the Subscriber event queue.
- `push`: public method used by the channel to report the event to each subscriber; it queues the event and releases the blocked Subscriber (if needed).

Tip: for a better reading of the output (to identify the recipient of the message) you can add to the event printed to the standard output (video) an identifier of the Subscriber (to identify the i-th instance of the subscriber. This field will be initialized at Subscriber construction time (within Subscriber constructor).

Single entity specification: Channel

The Channel (a passive object) must provide the following methods:

- **`add/removeSubscriber`** to, respectively, add and remove subscribers;
- **`put`**, that accepts event publications from the Editor and notifies all Subscribers of the arrival of the event.

Note: when the Editor publishes an event and there are no subscribers registered, the event is lost. In other words, events are **not persistent**.

Editor code

```
import java.io.*; /* Editor.java */

public class Editor extends Thread {
  private Channel channel;
  private final static String EOF = "-1";
  private BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));

  public Editor(Channel c) { super("Editor"); channel = c; } // Constructor method

  public void run() {
    String msg;
    try {
      System.out
          .println("\n^D(Unix)/^Z(Win)+enter to exit, enter to continue: ");
      int i = 0;
      while (stdIn.readLine() != null) { //while(!EOF)
        i++;
        System.out.print("E: message " + i + " ? ");
        msg = stdIn.readLine();
        channel.put("" + msg);
        System.out.println("^D(Unix)/^Z(Win)+enter to exit, enter to continue: ");
      } //while, here the Editor has read the EOF
    } catch (Exception e) { e.printStackTrace(); }
    System.out.println("E: ends...");
    channel.put(EOF);
  } //run()
} //Editor
```

Subscriber code 1/2

```
import java.util.Random; /* Subscriber.java */
import java.util.Vector;

public class Sottoscrittore extends Thread {
 private Channel channel; private int identifier = -1;
 private final static String EOF = "-1"; private Vector eventQueue = new Vector();
 // Constructor
 public Subscriber(Channel c, int id){channel = c; identifier = id; }

 public void run()
{try
{ Thread.sleep((int) (Math.random() * 3000)+2000);     //step a.
  System.out.println("\nSubscriber(S"+identifier+"): begins and subscribes...");
  canale.addSubscriber(this);                          //step b.
  Random r = new Random(); int N = 2 + (r.nextInt(5));
  System.out.println("\nSubscriber(S"+identifier+"): begins and subscribes...");
  for(int i=0; i<N; i++){                              //step c.
   String readMsg = read();
   if( readMsg.equals(EOF) ) break;                    //step c.
   System.out.println("S"+identifier+": "+readMsg);
  }
  Channel.removeSubscriber(this);                             //step d.
  System.out.println("\nS" + identifier + ": unsubscribes and terminates.");
 }
 catch(Exception e){ System.err.println("An error occurred, the following: "+e);
 e.printStackTrace(System.err); }
 }  //run()
```

Subscriber code 2/2

```
/**
  * This call blocks when the event queue is empty.
  * NOTE: read method consumes the message from the event queue.
  *
  */
 private synchronized String read() throws InterruptedException{
   if (eventQueue.size() == 0) wait();
   String firstReceivedMessage = (String) eventQueue.lastElement();
   eventQueue.remove(eventQueue.size() - 1);
   return firstReceivedMessage;
 }

 public synchronized void push(String msg) {
   eventQueue.insertElementAt(msg, 0);
   notify();
 }  // push
```

Channel code

```
import java.util.Vector;   /* Channel.java */
public class Channel {
  private Vector subscribers = new Vector();

 /** Public method to be used by the Editor for the publication of events. */
  public synchronized void put(String msg) {
    if (subscribers.isEmpty()) return;
    else
      for (int i = 0; i < subscribers.size(); i++) {
        Subscriber subscriberIth = (Subscriber) subscribers
           .elementAt(i);
        subscriberIth.push(msg);
      }
  }

  public synchronized void addSubscriber(Subscriber s) {
    // if the subscriber is not registered, add it to the list
    if (!subscribers.contains(s))
      subscribers.add(s);
  }

  public synchronized void removeSubscriber(Subscriber s) {
    // if the subscriber exists, remove it from the list
    if (subscribers.contains(s))
      subscribers.removeElement(s);
  }
}//Channel
```

Main program code

```
/* testMain.java */
public class testMain {
  final static int NUM_SUBSCRIBERS = 5;

  public static void main(String[] args) {
    Subscriber subscriber;
    Channel channel;
    Editor editor;
    try {
      // Create the channel and start the thread
      channel = new Channel();
      editor = new Editor(channel);
      editor.start();
      for (int i = 0; i < NUM_SUBSCRIBERS; i++) {
        // Create the subscribers and start the threads
        subscriber = new Subscriber(channel, i);
        subscriber.start();
      } //for
    } catch (Exception err) {
      System.err.println("Error, the following: " + err);
      System.exit(1);
    } //catch
  } //main
} //testMain
```

Compilation and Execution

1. How to compile a Java program

```
javac    Channel.java Editor.java
         Subscriber.java testMain.java
```

2. How to execute a Java program

```
java     testMain
```

Let’s try to compile and execute the event publication/subscription program ☺

Where are the code and the tools?

To download the code:

`http://www.lia.deis.unibo.it/Staff/`
`LucaFoschini/temp/srcLab0.zip`

See inside directory **C:\**

JAVA_HOME=`C:\Programmi\jd`

The compiler (**javac**)

`C:\Programmi\jdk1.5.0_09\b`

The Java interpreter(**java**)

`C:\Programmi\jdk1.5.0_09\java`

The **Eclipse** Integrated Development Environment (IDE)

Leaving .java e .class files in the current directory (“.”)