Ottava Esercitazione

Introduzione ai thread java

mutua esclusione

Agenda

I thread in java: creazione e attivazione di threads.

Esempio

- Concorrenza in Java: creazione ed attivazione di thread concorrenti.

Mutua esclusione in java: metodi synchronized

Esercizio 1 – da svolgere

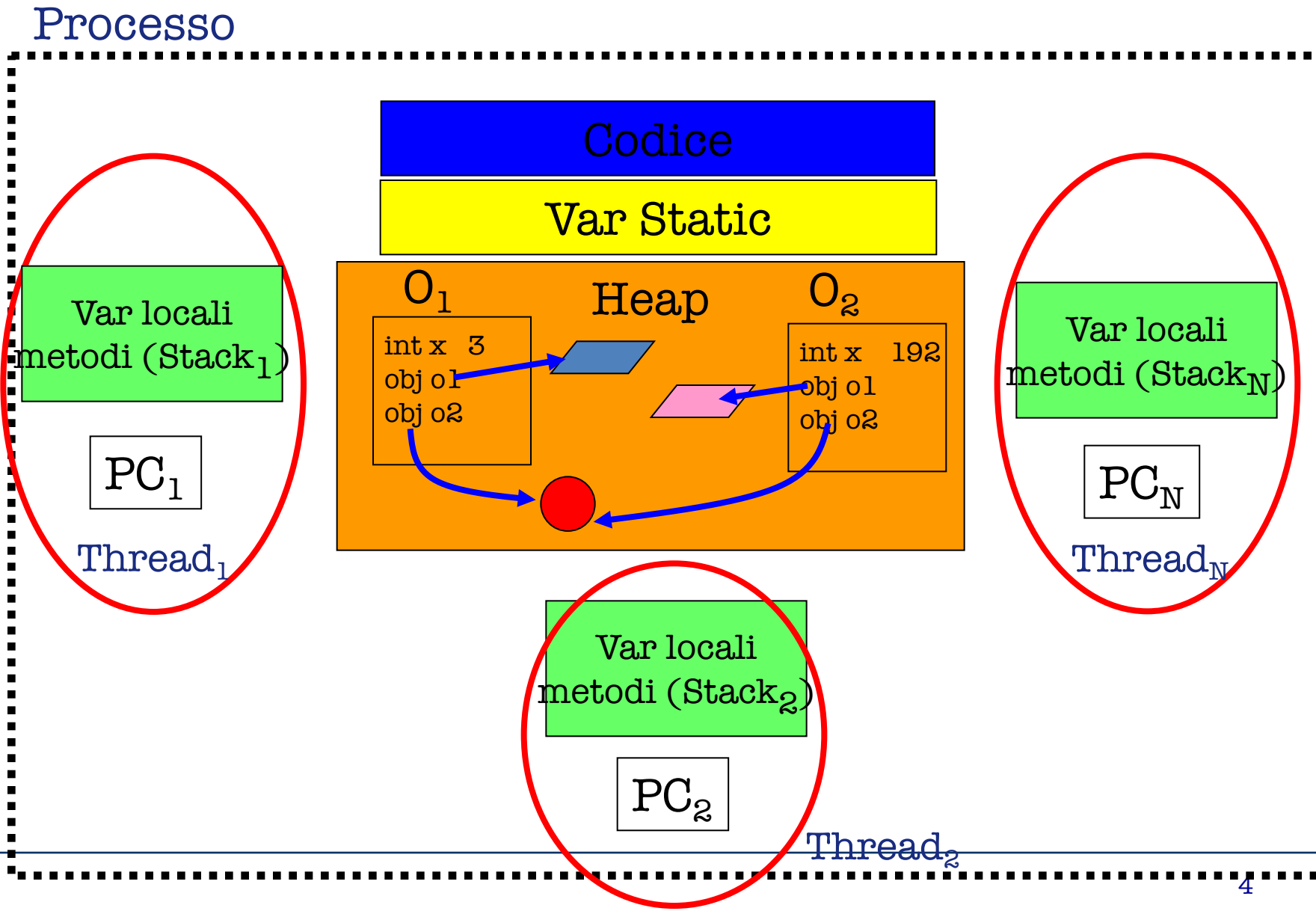
- Concorrenza in Java: sincronizzazione di thread concorrenti tramite syncronized

I threads in Java

- All'esecuzione di ogni programma Java corrisponde un task che contiene almeno un singolo thread, corrispondente all'esecuzione del metodo main() sulla JVM.

- E' possibile creare dinamicamente nuovi thread attivando concorrentemente le loro esecuzioni all'interno del programma.

Java Thread


Processo
Codice
Var Static
Heap
O_1
O_2
int x 3
obj o1
obj o2
int x 192
obj o1
obj o2
Var locali metodi ($Stack_1$)
PC_1
$Thread_1$
Var locali metodi ($Stack_N$)
PC_N
$Thread_N$
Var locali metodi ($Stack_2$)
PC_2
$Thread_2$

Java Thread: programmazione

Due metodi per definire thread in Java:

1. ***estendendo la classe*** `Thread`
2. ***implementando l' interfaccia*** `Runnable`

Definizione e uso dei java thread Esempio (1/2)

Scrivere una applicazione Java che simuli un autolavaggio.

- Nell'autolavaggio possono entrare sia automobili che moto.
- Le **automobili** possono essere di due tipi, ossia auto **grandi** oppure auto **piccole**.
- Le **moto**, invece, sono di un unico tipo.

Tutti gli autoveicoli devono essere **oggetti attivi** (ossia in grado di eseguire in maniera concorrente tramite thread)

In particolare, ciascun autoveicolo, quando eseguito, dovrà stampare su `stdout` un opportuno messaggio che descriva le sue caratteristiche.

Esempio (2/2)

Il programma deve definire le seguenti classi:

- **`Automobile`**: definisice le caratteristiche comuni di un'auto (marca, modello, targa, cilindrata, ...), un metodo astratto **`getType()`** ed un metodo concreto **`getMessage()`** che ritorna il messaggio da stampare e che richiami **`getType()`** per capire il tipo di automobile.
- **`AutoGrande`**: eredita da Automobile e specializza il comportamento di getType() in modo che ritorni la stringa "auto grande”
- **`AutoPiccola`**: eredita da Automobile e specializza getType() in modo che ritorni la stringa "auto piccola”
- **`Moto`**: definisce le caratteristiche della moto
- **`Autolavaggio`**: implementa il metodo main() che crea un numero (a scelta) di veicoli di ciascun tipo e li mette in esecuzione tramite thread

Esempio – Definizione e uso dei java thread

Scrivere una applicazione Java che simuli un autolavaggio.

- Nell'autolavaggio possono entrare sia automobili che moto.
- Le **automobili** possono essere di due tipi, ossia auto **grandi** oppure auto **piccole**.
- Le **moto**, invece, sono di un unico tipo.

Tutti gli autoveicoli devono essere oggetti attivi (ossia in grado di eseguire in maniera concorrente tramite thread)

In particolare, ciascun autoveicolo, quando eseguito, dovrà stampare su `stdout` un opportuno messaggio che descriva le sue caratteristiche.

Soluzione: classe Automobile

```
public abstract class Automobile {

        private String marca;
        private String modello;
        private String targa;
        private int cilindrata;

        public Automobile(String marca, String modello,
                          String targa, int cilindrata){
                this.marca = marca;
                this.modello = modello;
                this.cilindrata = cilindrata;
                this.targa = targa;
        }
// continua..
```

..classe Automobile

```
    //… continua
    public abstract String getType();

    public String getMessage(){
          return  this + " Tipo " + getType() +
                " Marca " + this.marca + "; Modello " +
                this.modello  + "; Cilindrata " +
                this.cilindrata;
    }

    public String toString(){
          return "[Automobile con targa: " +
                this.targa +"]";
    }

}//end of class Automobile
```

Soluzione: classe AutoGrande

```
public class AutoGrande extends Automobile
      implements Runnable {
```

Non posso estendere Thread, perchè devo già estendere Automobile. Quindi implemento Runnable.

```
      public AutoGrande(String marca, String modello,
                        String targa, int cilindrata)
      { super(marca, modello, targa, cilindrata);}

      @Override
      public String getType()
      {     return "AutoGrande";}


// continua..
```

..classe AutoGrande

```
   //.. continua

   @Override
   public void run() {
      System.out.println("Il thread per l'automobile ”
            + this + " ha iniziato" +"l'esecuzione.");
      System.out.println(this.getMessage());
      System.out.println("Il thread per l'automobile "
            + this + " sta per terminare");
   }

}//end of class AutoGrande
```

Soluzione: classe AutoPiccola

```
public class AutoPiccola extends Automobile
      implements Runnable {

      public AutoPiccola(String marca, String
modello,                  String targa, int cilindrata)
      { super(marca, modello, targa, cilindrata);}

      @Override
      public String getType() {
            return "AutoPiccola";
      }


      // continua..
```

..classe AutoPiccola

```
  // ..continua
  @Override
  public void run() {
    System.out.println("Il thread per
      l'automobile "+this+" ha iniziato"
      +"l'esecuzione.");
    System.out.println(this.getMessage());
    System.out.println("Il thread per
      l'automobile "+this+" sta per terminare");
  }

}//end of class AutoPiccola
```

Soluzione: classe Moto

```
public class Moto extends Thread {
    private String marca;
    private String targa;
    private String modello;
    private int cilindrata;

    public Moto(String marca, String modello,
                String targa, int cilindrata) {
        this.marca = marca; this.targa = targa;
        this.modello = modello;
        this.cilindrata = cilindrata;
    }
    public String getMessage() {
        return this+" Marca "+this.marca+"; Modello "
            +this.modello+"; Cilindrata
"+this.cilindrata;
    }
      // continua..
```

Posso estendere Thread, perchè questa classe non deve estendere nient'altro

..classe Moto

```
  //.. continua
      @Override
      public String toString() {
         return "[Moto con targa: " + this.targa + "]";
      }

      @Override
      public void run() {
         System.out.println("Il thread per la moto " +
            this + " ha iniziato" +"l'esecuzione.");
         System.out.println(this.getMessage());
         System.out.println("Il thread per la moto " +
            this + " sta per terminare");
      }

} //end of class Moto
```

Classe Autolavaggio

```
public class AutoLavaggio{
      public static void main(String[] args) {
         AutoPiccola a1 = new AutoPiccola("FIAT",
                  "Modello1", "AB123BC", 2000);
         AutoGrande a2 = new AutoGrande("Mercedes",
                  "Modello2", "ILNY", 3000);
         Moto m1 = new Moto("Kawasaki","Ninja",
                        "ASTFG",2);
         Thread t1 = new Thread(a1);
         Thread t2 = new Thread(a2);

         t1.start();
         t2.start();
         m1.start();
      }
}
```

AutoPiccola e AutoGrande non sono Thread, implementano solo Runnable. Mi devo solo ricordare di metterle in un Thread per poterle far partire.

Note

Provare l'applicazione (download dal sito web del corso).

Due versioni:

- **SimpleLavaggio**
- **RandomLavaggio** (definizione casuale di tipologia e numero dei thread da generare, v. **`java.util.Random`**)

Link utili:

- Oracle Java Doc per Java 8 SE: http://docs.oracle.com/javase/8/docs/api/
- Buon tutorial Oracle sulla concorrenza in Java: http://docs.oracle.com/javase/tutorial/essential/concurrency/

Mutua esclusione in Java:

metodi **synchronized**

Mutua esclusione

In java è possibile denotare alcune sezioni di codice che operano su un oggetto in modo mutuamente esclusivo (cioè, sono **sezioni critiche**) tramite la parola chiave **`synchronized`**:

- **metodi sinchronized**
- **[blocchi synchronized]**

-> Ogni parte di codice etichettata come **`synchronized`** viene eseguita sull'oggetto al quale viene riferita in modo **mutuamente esclusivo**, cioè da un thread alla volta.

Metodi synchronized

- **Mutua esclusione** tra i **metodi di una classe**

```
public class Contatore {
private int i=0;
public synchronized void incrementa()
{ i ++;
   System.out.println("Il contatore è stato incrementato.
   Nuovo valore: "+i+"!");

}
public synchronized void decrementa()
{ i--;
 System.out.println("Il contatore è stato decrementato.
 Nuovo valore: "+i+"!");}
}
```

Metodi synchronized

Quando un metodo **synchronized** viene invocato da un thread T per operare su un oggetto O della classe, l'esecuzione del metodo avviene in **mutua esclusione:**

- se un altro thread sta eseguendo un metodo **synchronized** sullo stesso oggetto (l'oggetto O è **occupato**), il thread T viene **sospeso** ed inserito nella **coda** (**entry set**) associata ad O. Quando l'oggetto O tornerà libero, verrà riattivato il primo processo in coda → T uscirà dalla coda quando l'oggetto O sarà libero e non ci saranno più thread che lo precedono nell'entry set di O.

- se nessun metodo **synchronized** sull'oggetto O è in esecuzione (l'oggetto è **libero**), il metodo viene eseguito (O viene occupato per tutta la durata della chiamata).

Esercizio 1 (1/4)

Si consideri un'azienda agricola che offre un servizio di vendita al dettaglio dei propri prodotti.

Nel periodo primaverile l'azienda si occupa della **raccolta** e della **vendita** dei seguenti prodotti:

- **fragole** (vendute a cestini)
- **asparagi** (venduti a mazzi)

Pertanto, mentre avviene la raccolta si svolge la **vendita** dei prodotti man mano raccolti, presso un **banco** dedicato.

Siano:

- **Pf** il prezzo di vendita di un cestino di fragole
- **Pa** il prezzo di vendita di un mazzo di asparagi

Esercizio 1 (2/4)

Al **banco** di vendita possono accedere:

- **operai agricoli**
- **acquirenti**

Comportamento dell'operaio agricolo.
Si supponga che ogni operaio abbia un comportamento **ciclico:** per un numero arbitrario di volte, l'operaio **accede al banco per depositare** una quantità arbitraria di cestini di fragole e/o di mazzi di asparagi appena raccolti.

Esercizio 1 (3/4)

Comportamento dell'acquirente.
Si supponga che ogni acquirente **acceda al banco per acquistare** una quantità arbitraria di cestini di fragole o di mazzi di asparagi (per semplicità si assuma che comperi un solo tipo di prodotto, fragole oppure asparagi):

- se la quantità di prodotto richiesta **non è disponibile**, l'acquirente rinuncia all'acquisto e se ne va.
- se la quantità di prodotto richiesto **è disponibile**, l'acquirente la preleva dal banco, paga e va via.

Esercizio 1 (4/4)

Progettare un'applicazione java che regoli gli accessi al **banco**, nella quale:

- **operai e acquirenti** siano rappresentati da **thread** concorrenti,
- il **banco** sia rappresentato da un **oggetto condiviso** da tutti i thread.

Il **thread iniziale** (**main**), una volta terminati tutti gli altri thread, stampa:

- **l'incasso** totale derivato dalle vendite;
- **l'invenduto**, ovvero quanti mazzi di asparagi e quanti cestini di fragole sono rimasti sul banco.

Successivamente il main termina.

Impostazione

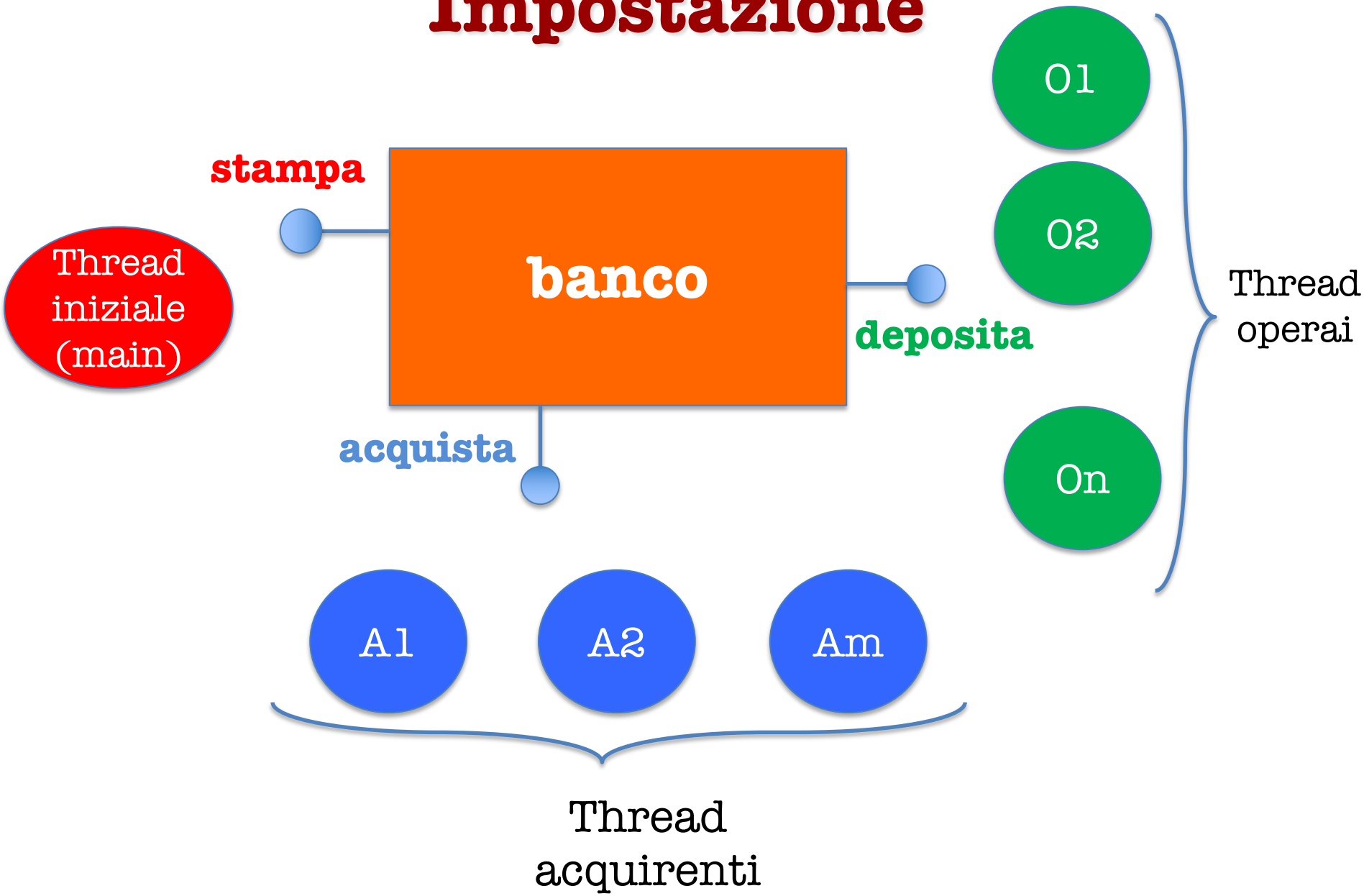
Classi da definire:

- **Banco**: è una risorsa condivisa acceduta in modo concorrente dai thread operai e acquirenti.
  - Quali variabili locali?
  - Quali metodi (necessità di sincronizzazione!)?

- **Operaio**: thread dipendente dell'azienda
- **Acquirente**: thread cliente dell'azienda

- **Main:** definisce il metodo main

Impostazione



Impostazione

Classi da definire:

- **Operaio**: il generico thread concorrente che accede al banco per depositare merce. Il suo comportamento è definito dal metodo run:

```
public class Operaio <quale metodo usare per la definizione?>{
  Banco B;
  <costruttore, etc.>

  public void run() {
       while(..)
       {      <raccogli fragole/asparagi>
              B.deposita(...);
       }

}
```

Impostazione

Classi da definire:

- **Acquirente**: il generico thread cliente che accede al banco per prelevare fragole o asparagi e consegnare denaro. Il suo comportamento è definito dal metodo run:

```
public class Acquirente <quale metodo usare per la def.?> {
  Banco B;
  <costruttore, etc.>

  public void run() {
    int OK;
    …
    OK = m.acquista(...);
    if (OK){
       <acquisto riuscito>
       }
    else
       <acquisto non riuscito>
}
```

Banco: è una risorsa condivisa da thread concorrenti.

-> Usiamo i metodi **synchronized**.

```
public class Banco {
  // var. locali: fragole_disp, asparagi_disp, incasso;

public synchronized int acquista(..){
    <verifica disponibilità + eventuale vendita>
    return risultato;
  }

  public synchronized void deposita(..){
    <aggiunta prodotto al banco>
  }

  public syncronized void stampa (){
    <stampa risultati raccolta >
  }
```

valore restituito:

- 1 se il thread ha comprato la merce,
- 0 se l'acquisto non è stato possibile

Classe Main: contiene il metodo main

```
import java.util.Random;
public class Main{
   <definizioni costanti ecc.>

   public static void main(String[] args) {
      <creazione Banco B>
      <creazione Operai>
      <creazione Acquirenti>
      <attivazione di tutti i thread>
      <attesa della terminazione di tutti i thread>
      B.stampa(); //stampa dei valori finali
   } }
```

Esercizio 2

- Definire una variante dell'esercizio 1, in cui ogni acquirente, nel caso in cui il prodotto richiesto non sia disponibile, «attende».

- Come realizzare l'attesa? → con gli strumenti finora visti l'unica soluzione possibile è **l'attesa attiva...**