

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

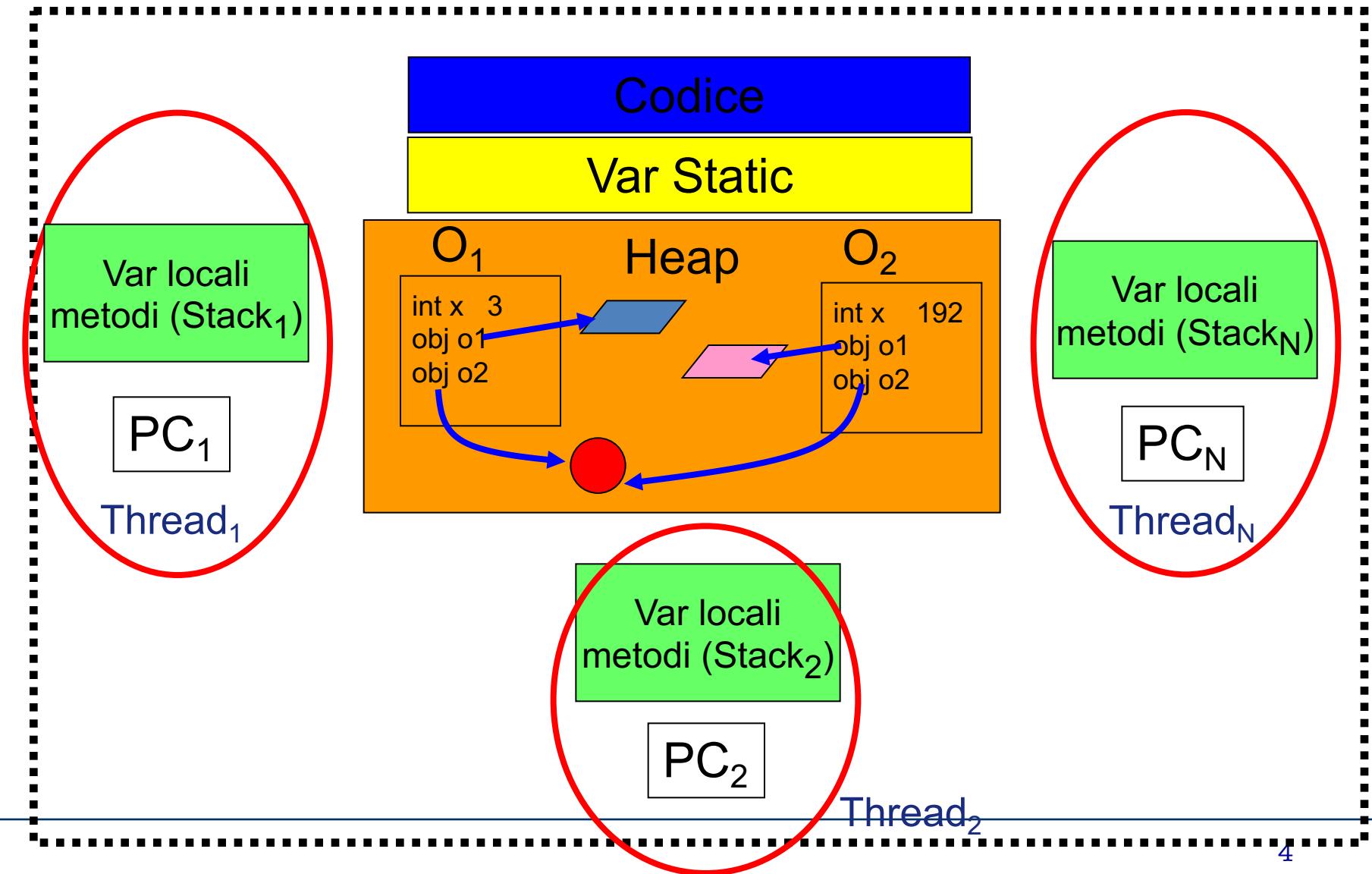
- Concorrenza in Java: sincronizzazione di thread concorrenti tramite synchronized

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



Java Thread: programmazione

Due metodi per definire thread in Java:

1. estendendo la classe Thread

2. implementando l'interfaccia Runnable

Primo metodo: esempio

```
public class SimpleThread extends Thread{  
  
    public SimpleThread(String str)  
    {super(str);}  
  
    public void run() {  
        for(int i=0; i<10; i++)  
        { System.out.println(i+ " " +getName());  
        try{  
            sleep((int)Math.random()*1000);  
        } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE! "+getName());  
    }  
}
```

```
public class EsempioConDueThreads
{
    public static void main(String[] args)
    { SimpleThread st1= new SimpleThread("Pippo");
      st1.start();
    }
}
```

2. Thread come classi che implementano Runnable

Se un thread deve ereditare da una classe C diversa da Thread, la definizione si ottiene

1. Definendo una nuova classe **C1** che:
 - **estende** la classe C
 - **implementa** l'interfaccia **Runnable**
 - **ridefinisce** il metodo **run()**.
2. si crea un'istanza **o1** della classe **C1** tramite **new**;
3. si crea **un'istanza t della classe Thread** con **new**, passandole come **parametro** l'oggetto **o1** **creato al punto 2** ;
4. si esegue il thread invocando il metodo **start()** **sull'oggetto t** **creato al punto 3**.

Secondo metodo: esempio

```
class C1 extends C implements Runnable
{
    public void run() {
        for (int i=1; i<=10; i++)
            System.out.println(i + " " + i*i);
    }
}

public class Esempio {
    public static void main(String args[]) {
        C1 o1 = new C1();
        Thread t = new Thread (o1);
        t.start();
    }
}
```

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.

Esempio - Traccia (2/2)

Il programma deve definire le seguenti classi:

- **Automobile**: definisce 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

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");

}

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/>
-

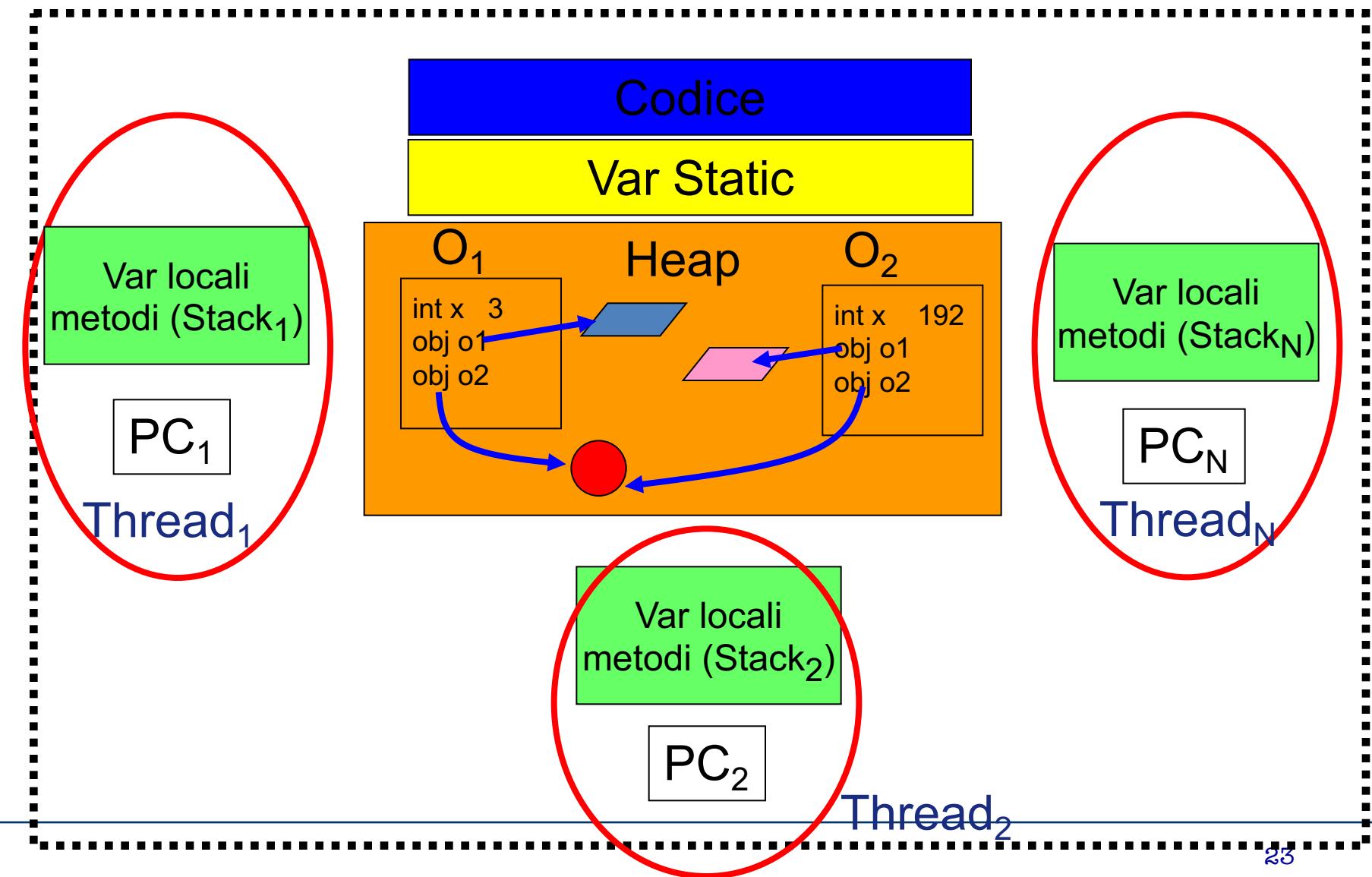
Mutua esclusione in Java:

metodi `synchronized`



Thread

Processo



-> O₁ e O₂ sono oggetti condivisi dai thread concorrenti

Mutua esclusione

In java è possibile denotare alcune sezioni di codice che operano su un oggetto come **sezioni critiche** tramite la parola chiave **synchronized**:

- metodi synchronized
- [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 per operare su un oggetto della classe, l'esecuzione del metodo avviene in **mutua esclusione**:

- se un altro thread sta eseguendo un metodo **synchronized** sullo stesso oggetto (l'oggetto è **occupato**), il thread chiamante viene sospeso.

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

Esempio: accesso concorrente a un contatore

```
public class CompetingProc extends Thread
{ Contatore r; /* risorsa condivisa */
  int T; // incrementa se tipo=1; decrementa se tipo=-1

  public CompetingProc(Contatore R,  int tipo)
  {   this.r = R;
      this.T = tipo;
  }

  public void run()
  {   try{
      while(true)
      {   if (T>0)          r.incrementa();
          else if (T<0)    r.decrementa();
      }
      }catch(InterruptedException e) {}
  }
}
```

```
public class Contatore {  
    private int C;  
  
    public Contatore(int i)  
    { this.C=i; }  
  
    public synchronized void incrementa()  
    { C++;  
        System.out.print("\n eseguito incremento: valore  
attuale del contatore: "+ C+" ....\n");  
    }  
  
    public synchronized void decrementa()  
    { C--;  
        System.out.print("\n eseguito decremento: valore  
attuale del contatore: "+ C+" ....\n");  
    }  
}
```

```
import java.util.*;  
  
public class Prova_mutex{ // test  
  
    public static void main(String args[]) {  
        final int NP=30;  
        Contatore C =new contatore(0);  
        CompetingProc []F=new CompetingProc[NP];  
        int i;  
        for(i=0;i<(NP/2);i++)  
            F[i]=new CompetingProc(C, 1); // incrementa  
        for(i=(NP/2);i<NP;i++)  
            F[i]=new CompetingProc(C, -1); // decrementa  
        for(i=0;i<NP;i++)  
            F[i].start();  
        for(i=0; i<NP; i++)  
            F[i].join();  
        System.out.print("\n main: tutti i figli sono  
terminati... Ciao!!\n");  
    } }  
  
Il metodo join() consente al thread di  
sicronizzarsi con la terminazione dei  
figli
```

Esercizio 1 - da svolgere

A causa della pandemia COVID-19 il gestore di un supermercato ha fissato al valore MAX il numero massimo di clienti che possono stare contemporaneamente all'interno del punto vendita.

Non essendo previsto un meccanismo di prenotazione, ogni cliente che vuole fare la spesa si reca singolarmente al punto vendita:

- nel caso **vi sia ancora posto**, entra occupando un posto;
 - nel caso in cui il supermercato sia **pieno**, per evitare assembramenti, il cliente rinuncia ad entrare, si fa un giretto, e ritorna dopo un tempo arbitrario per ritentare l'accesso.
-

Esercizio 1 - da svolgere

Progettare un'applicazione java che regoli gli accessi al punto vendita, nella quale:

- **ogni cliente sia rappresentato da un thread distinto,**
- il **supermercato sia rappresentato da un oggetto condiviso.**

In particolare, ogni thread che tenta l'accesso al supermercato e trova posto aggiorna lo stato del supermercato e fa la spesa; altrimenti attende un po' di tempo (si fa un giretto) e riprova a entrare. Se dopo N tentativi non trova posto, il cliente rinuncia e termina. Il supermercato ha un'unica cassa C (con € 0 di valore iniziale), il cui valore viene incrementato man mano che i clienti pagano la propria spesa.

Pertanto al termine della spesa ogni cliente paga la propria spesa **incrementando il valore della cassa C**, esce dal supermercato, e termina.

Il **thread iniziale** (main), una volta terminati tutti i thread clienti, stampa il numero totale dei clienti serviti e il valore finale della cassa; successivamente termina.

Impostazione

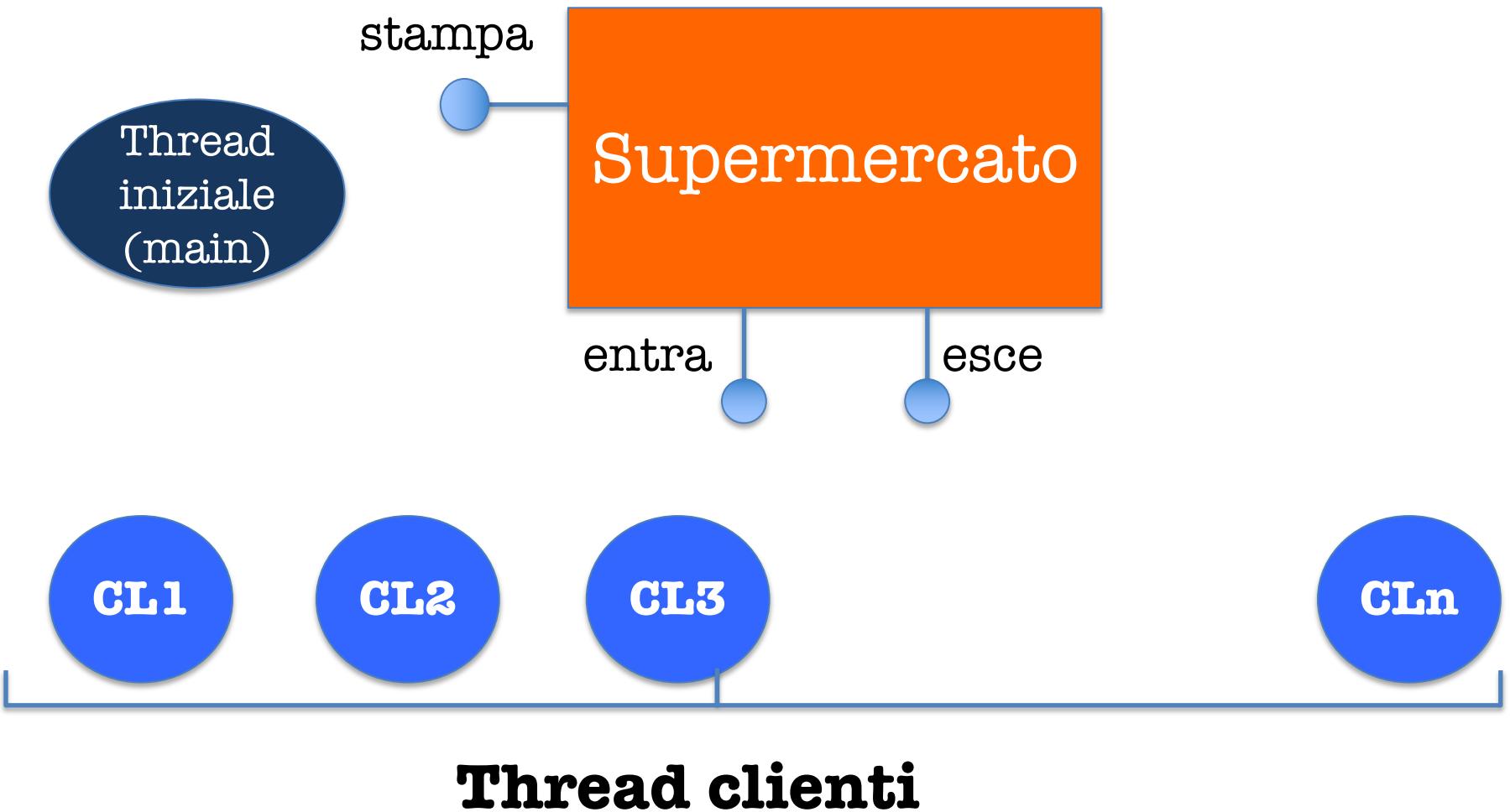
Supponiamo per semplicità che:

- Ogni cliente stia nel supermercato per il tempo che vuole.
- L'importo della spesa di ogni cliente venga definito in modo casuale.

Classi da definire:

- **Supermercato**: è una risorsa condivisa acceduta in modo concorrente dai thread clienti.
 - Quali variabili locali?
 - Quali metodi (necessità di sincronizzazione!)?
 - **Clienti**: thread
 - **Spesa** (main)
-

Impostazione



Impostazione

Classi da definire:

- **Cliente**: il generico thread concorrente che accede al supermercato. Il suo comportamento è definito dal metodo **run**:

```
public class Cliente extends Thread {  
    Supermercato s;  
    <costruttore, etc.>  
  
    public void run() {  
        int entrato;  
        ...  
        entrato = s.entra();  
        if (entrato){  
            <faccio la spesa>  
            s.esci();}  
        else  
            ...  
        ...  
    }  
}
```

Verifica se c'è posto e lo occupa

Paga e libera il posto

Teatro: Il Supermercato è condiviso da thread concorrenti.

-> Usiamo i metodi **synchronized**.

```
public class Supermercato {  
    // var. locali: posti occupati, cassa, totale_clienti  
  
    public synchronized int entra(){  
        <verifica posto+eventuale occupazione>  
        return risultato;  
    }  
  
    public synchronized void esce(){  
        <incremento cassa e totale_clienti + liberazione posto>  
    }  
  
    public synchronized void stampa (){  
        <stampa cassa e totale_clienti >  
    }  
}
```

- valore restituito:
- 1 se il thread ha occupato un posto,
 - 0 se il supermercato è pieno (il thread non entra)

Classe Supermercato: contiene il metodo main

```
import java.util.Random;
public class Spesa{
    private final static int MAX_NUM_CLIENTI = 20;

    public static void main(String[] args) {
        Random r = new Random(System.currentTimeMillis());
        int NC = r.nextInt(MAX_NUM_CLIENTI);
        Cliente[] CL = new Cliente[NC];
        Supermercato S= new Supermercato(...);

        <creazione NC clienti>
        <attivazione NC clienti>

        // il thread main deve aspettare la terminazione
        // dei clienti prima di stampare i valori finali
        // -> uso del metodo join:
        for(i=0; i<NC; i++)
            CL[i].join(); //attesa term. cliente i-simo
        S.stampa(); //stampa dei valori finali
    }
}
```