Ottava Esercitazione

introduzione ai thread java
mutua esclusione

Agenda

Esempio 1

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

Esercizio 2 – da svolgere

- Concorrenza in Java: sincronizzazione di thread concorrenti tramite syncronized

Esempio 1 – Definizione e uso dei java thread

Scrivere una applicazione Java che simuli un semplice **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 1 - Traccia (2/2)

Nello specifico, 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

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)

- 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/

Esercizio 2 – da svolgere

Si consideri un **teatro** usato per lo svolgimento di eventi musicali.

Per motivi di sicurezza il teatro non può accogliere più di MAX persone.

Non essendo previsto un meccanismo di prenotazione, prima di un evento ogni cliente accede singolarmente al teatro, e,

- nel caso **vi sia ancora posto**, entra occupando un posto;
- nel caso in cui il teatro sia **pieno**, il cliente rinuncia ad entrare.

Esercizio 2 – da svolgere

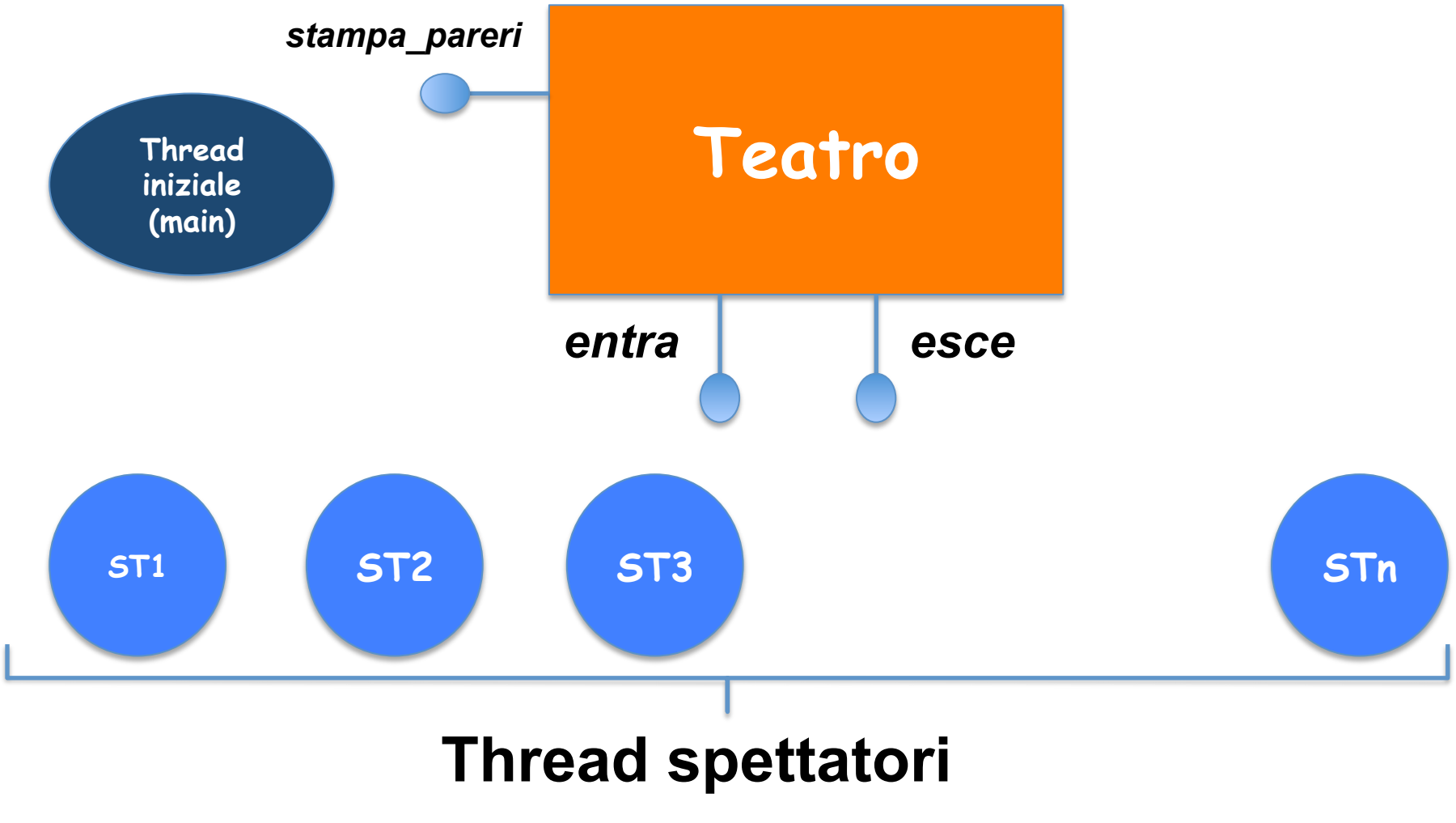
Progettare un'applicazione java che regoli gli accessi al teatro per un singolo evento, nella quale:

- **ogni spettatore sia rappresentato da un thread distinto,**
- il **teatro sia rappresentato da un oggetto condiviso.**

In particolare, ogni thread che tenta l'accesso al teatro e trova posto aggiorna lo stato del teatro e assiste allo spettacolo; altrimenti esso termina.

Al termine dello spettacolo ogni spettatore **esprime un parere** (positivo o negativo) sull'evento (che verrà registrato dal teatro), esce dal teatro, e termina.
Il thread iniziale (main), una volta terminati tutti i thread spettatori, stampa il numero dei pareri positivi e negativi sullo spettacolo e temina.

Impostazione



Impostazione

Supponiamo per semplicità che:

- Ogni spettatore sta dentro il teatro per il tempo che vuole. NON modelliamo un vero e proprio spettacolo in cui lo spettatore esce solo quando è finito.

Classi da definire:

- **Teatro**: il teatro è una risorsa condivisa acceduta in modo concorrente dai thread spettatori.
  - Quali variabili locali?
  - Quali metodi (necessità di sincronizzazione!)?
- **Spettatore**
- **Spettacolo** (main)

Impostazione

Classi da definire:

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

```
public class Spettatore extends Thread {
  Teatro t;
  <costruttore, etc.>
  public void run() {
    int entrato;
    entrato = t.entra();
    if (entrato){
       <visione spettacolo>
       t.esci();}
    else
         System.out.println("Il thread "+this.getName()+
              " non e' riuscito a entrare!");
}
```

Verifica se c'è posto e lo occupa

Scrive il parere e libera il posto

Teatro: Il teatro è condiviso da thread concorrenti.
Usiamo IL LOCK DELL'OGGETTO: metodi **synchronized**.

```
public class Teatro {
  // var. locali: posti occupati, pareri pos./neg.
  private int capienza,liberi,pos,neg;
  public synchronized int entra(){
    <verifica posto+eventuale occupazione>
    return risultato;
  }

  public synchronized void esce(){
    <registrazione parere + liberazione posto>
  }

  public syncronized void stampa_pareri(){
    <stampa numero pareri positivi, numero negativi>
  }
```

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

Classe Spettacolo: contiene il metodo main

```
import java.util.Random;
public class Spettacolo{
    private final static int MAX_NUM_SPETTATORI = 100;
    public static void main(String[] args) {
        Random r = new Random(System.currentTimeMillis());
        int SP = r.nextInt(MAX_NUM_SPETTATORI);
        Spettatore[] ST = new Spettatore[SP];
        Teatro T= new Teatro(…);
        <creazione SP spettatori>
        <attivazione SP spettatori>
        //attesa terminazione spettatori:
        for(i=0; i<SP; i++)
            ST[i].join(); //attesa terminazione spettatori
        T.stampa_pareri(); //stampa dei pareri
    }
}
```