Corso di Laurea in Ingegneria Informatica e Automatica
A. A. 2016/2017

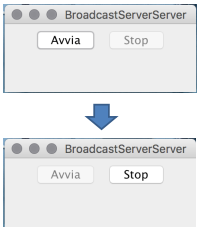
Programmazione orientata agli oggetti

Luca Iocchi, Massimo Mecella, Daniele Sora

E8 – costruzione di un semplice forum in Java

Specifiche server

- Costruire un programma server che si comporti in questo modo
  - Apra un finestra di controllo con cui l'utente può avviare e spegnere il server
  - Si metta in ascolto sulla porta 3000 e gestisca vari client in multi-threading

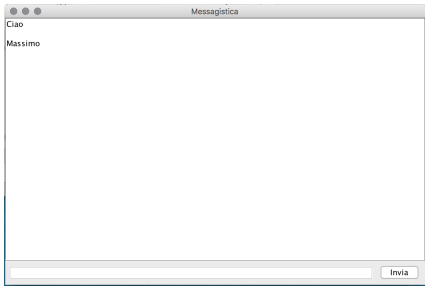


Specifiche client

- Costruire un programma client che si comporti in questo modo
  - Si connette al server precedentemente descritto
  - Offra all'utente un'interfaccia per scrivere messaggi testuali e mostrare quelli ricevuti da altri client
  - In particolare, l'utente inserisce un testo nella text area in basso e spinge *Invia* per mandarlo in broadcast a tutti i client connessi

Specifiche client

- Ad es., la figura mostra il caso di un client che ha ricevuto Ciao e sta scrivendo Massimo. Alla pressione di Invia tutti i client connessi (incluso il mittente) riceveranno il messaggio e lo mostreranno nella text area in alto (vedi figura)



Specifiche di comunicazione

- Il client recupera messaggi dal server interrogandolo ogni 2000 millisec, ed è multi-threaded (al fine di garantire interattività con l'utente e contemporaneamente gestire il *polling* verso il server)
- Il server gestisce ogni client attraverso un apposito thread
- Quando un thread del server riceve un messaggio dal proprio client, il programma server lo distribuisce a tutti i thread in modo che tutti i client lo ricevano
- In caso di qualsiasi errore, sia i programmi client che quello server devono mostrare opportuni messaggi all'utente tramite pannello e poi terminano in modo controllato

COMPLEMENTI SUI THREAD - SYNCHRONIZED

Sincronizzazione per accesso dati condivisi

- Ci sono molte situazioni in cui diversi thread concorrenti in esecuzione condividono le medesime risorse. Ogni thread deve tener conto dello stato e delle attività degli altri.
- Visto che i thread condividono una risorsa comune, devono essere sincronizzati in qualche modo in modo da poter cooperare.

Esempio accesso a dati condivisi (1/4)

Consideriamo la seguente classe Contatore

```
public class Contatore {
   private int valore;

   public void incrementa() {
        int tmp = valore;
        try { Thread.sleep(10);
        } catch (InterruptedException ex) {}
        valore = tmp + 1;
        System.out.println(
                Thread.currentThread().getName() +
                                ": il contatore segna " + valore);

   }

   public int getValore() {
        return valore;
   }

}
```

Esempio accesso a dati condivisi (2/4)

Consideriamo poi il seguente processo che conta usando un Contatore

```
public class ProcessoConta implements Runnable {
   private Contatore contatore;

   public ProcessoConta(Contatore c) {
        contatore = c;
   }

   public void run() {
        for (int i = 0; i < 10; i++) {
                contatore.incrementa();
                try { Thread.sleep(10);
                } catch (InterruptedException ex) {}

        }
   }
}
```

Esempio accesso a dati condivisi (3/4)

Consideriamo infine il seguente main che genera due processi conta che condividono il contatore.

```
public class Main {
   public static void main(String[] args) {
        Contatore cnt = new Contatore();
        Thread a = new Thread(new ProcessoConta(cnt));
        Thread b = new Thread(new ProcessoConta(cnt));
        a.start();
        b.start();
   }
}
```

Esempio accesso a dati condivisi (4/4)

Eseguendo il main otteniamo il seguente risultato! Il contatore sbaglia!!!

```
Thread-0: il contatore segna 1
Thread-1: il contatore segna 1
Thread-0: il contatore segna 2
Thread-1: il contatore segna 2
Thread-1: il contatore segna 3
Thread-0: il contatore segna 3
Thread-1: il contatore segna 4
Thread-0: il contatore segna 4
Thread-1: il contatore segna 5
Thread-0: il contatore segna 5
Thread-1: il contatore segna 6
Thread-0: il contatore segna 6
Thread-1: il contatore segna 7
Thread-0: il contatore segna 7
Thread-1: il contatore segna 8
Thread-0: il contatore segna 8
Thread-1: il contatore segna 9
Thread-0: il contatore segna 9
Thread-1: il contatore segna 10
Thread-0: il contatore segna 10
```

Esempio accesso a dati condivisi (1b/4)

Consideriamo la seguente classe Contatore

```
public class Contatore {
   private int valore;

   public synchronized void incrementa() {
        int tmp = valore;
        try { Thread.sleep(10);
        } catch (InterruptedException ex) {}
        valore = tmp + 1;
        System.out.println(
                Thread.currentThread().getName() +
                        ": il contatore segna " + valore);

   }

   public synchronized int getValore() {
        return valore;
   }

}
```

Esempio accesso a dati condivisi (4/4b)

Eseguendo il main otteniamo il seguente risultato! Ora con il contatore sychronized i risultati sono corretti!!!

```
Thread-0: il contatore segna 1
Thread-1: il contatore segna 2
Thread-0: il contatore segna 3
Thread-1: il contatore segna 4
Thread-0: il contatore segna 5
Thread-1: il contatore segna 6
Thread-0: il contatore segna 7
Thread-1: il contatore segna 8
Thread-0: il contatore segna 9
Thread-1: il contatore segna 10
Thread-0: il contatore segna 11
Thread-1: il contatore segna 12
Thread-0: il contatore segna 13
Thread-1: il contatore segna 14
Thread-0: il contatore segna 15
Thread-1: il contatore segna 16
Thread-0: il contatore segna 17
Thread-1: il contatore segna 18
Thread-0: il contatore segna 19
Thread-1: il contatore segna 20
```

Monitor (1/2)

- "Custode" di un oggetto che determina l'accesso a suoi comportamenti (metodi e sezioni di codice).
- Mutua esclusione su gruppi di procedure
  - Un thread alla volta in esecuzione (altri thread sospesi).
- In Java la parola chiave è ***synchronized***
  - Nella segnatura del metodo di un oggetto (non solo...)
  - Su esecuzione di "synchronized", verifica da parte del monitor sull'uso dell'oggetto ed accesso garantito o bloccato al thread.
  - Coda di thread per l'accesso al metodo *synchronized di* un oggetto.
- Monitor oggetto rilasciato dal thread a fine esecuzione di metodo (o sezione) synchronized.

14

Monitor (2/2)

Cosa sincronizziamo

1. Metodi di istanza (`public synchronized`…)
   - I metodo di istanza synchronized può essere attivo in un determinato momento.
2. Metodi statici di classi (`public static synchronized`…)
   - monitor per classe che regola l'accesso a tutti i metodi static di quella classe.
   - Solo un metodo static synchronized può essere attivo in un determinato momento.
3. Singoli blocchi di istruzione, controllati da una variabile oggetto `synchronized (oggetto) {…}`
   - Espressione restituisce un oggetto, NON un tipo semplice.
   - Usata per sincronizzare metodi non sincronizzabili (ad es. controllo accesso agli oggetti array).
   - La sincronizzazione viene fatta in base all'oggetto e un solo blocco synchronized su un particolare oggetto può essere attivo in un determinato momento.
     - NB: se la sincronizzazione viene fatta su istanze differenti, i due blocchi non saranno mutuamente esclusivi!

15

Forme di codice synchronized

```
synchronized void metodo1(…) {…}
synchronized void metodo2(…) {…}
```

Guarda il monitor su oggetto `this`

```
static synchronized void metodo1(…) {…}
static synchronized void metodo2(…) {…}
```

Guardano il monitor su oggetto `.class` associato alla classe

```
void metodo1(…) {
      synchronized(ogg){
                …
      }
}
```

Guarda il monitor su oggetto `ogg`

16

Metodi non synchronized

- Non interpellano il monitor dell'oggetto
- Ignorano il monitor e vengono eseguiti regolarmente da qualsiasi thread ne faccia richiesta
- Da ricordare:
  - un solo thread alla volta può eseguire metodi ***synchronized***
  - un qualsiasi numero di thread può eseguire metodi **non *synchronized***

17

Deadlock

- Un uso non attento dei lock/monitor/synchronized può portare a diversi problemi che non si manifestano in applicazioni sequenziali. Il più impartante tra questi è il deadlock
- **Deadlock**: due thread devono accedere a due oggetti ma ciascuno acquisisce il lock su uno dei due e poi aspetta il rilascio dell'altro oggetto da parte dell'altro thread, il che non avverrà mai perché a sua volta l'altro thread è in attesa del rilascio del primo oggetto da parte del primo thread.
- Nel seguito si mostra un esempio di deadlock in Java

Esempio deadlock (1/4)

```
public class Risorsa {
    private String nome;
    private int valore;

    public Risorsa(String id, int v) {
          nome = id;
          valore = v;
    }

    public synchronized int getValore() {
          return valore;
    }

    public synchronized String getNome() {
          return nome;
    }

    public synchronized Risorsa maxVal(Risorsa ra) {
          try {
                Thread.sleep(100);
          } catch (InterruptedException e) {
          }
          int va = ra.getValore();
          if (valore >= va)
                return this;
          else
                return ra;
    }
}
```

Consideriamo la seguente risorsa. Si noti che in maxVal this accede a un'altra risorsa ra (nel thread corrente)

Esempio deadlock (2/4)

```
public class Confronto implements Runnable {
    Risorsa r1;
    Risorsa r2;

    public Confronto(Risorsa r1, Risorsa r2) {
          this.r1 = r1;
          this.r2 = r2;
    }

    public void run() {
          System.out.println(Thread.currentThread().getName() + "
    begins");
          Risorsa rm = r1.maxVal(r2);
          String rn = rm.getNome();
          System.out.println(Thread.currentThread().getName()
                                      + ": la risorsa più grande è " + rn);
          System.out.println(Thread.currentThread().getName() + "
    ends");
    }

}
```

Confronto è un Runnable che in run()invoca maxVal() sulla r1, e tale metodo a sua volta invoca getValore() su r2.

Esempio deadlock (3/4)

```
public class MainSingoloConfronto {
    public static void main(String[] args) throws InterruptedException {
          Risorsa r1 = new Risorsa("alpha",5);
          Risorsa r2 = new Risorsa("beta",20);
          System.out.println(Thread.currentThread().getName() + " begins");
          Confronto c1 = new Confronto(r1, r2);
          Thread t1 = new Thread(c1);
          t1.start();
          Thread.sleep(10000); //aspetta 10 sec
          System.out.println(Thread.currentThread().getName() + " ends");
    }
}
```

Output:

```
main begins
Thread-0 begins
main ends
Thread-0 ends
```

Consideriamo prima un solo thread. Tutto va come previsto.

Esempio deadlock (3/4)

```
public class MainDeadlock {
    public static void main(String[] args) throws InterruptedException {
          Risorsa r1 = new Risorsa("alfa",5);
          Risorsa r2 = new Risorsa("beta", 20);
          System.out.println(Thread.currentThread().getName() + " begins");
          Confronto c1 = new Confronto(r1,r2);
          Confronto c2 = new Confronto(r2,r1);
          Thread t1 = new Thread(c1); //t1.setDaemon(true);
          Thread t2 = new Thread(c2); //t2.setDaemon(true);
          t1.start();
          t2.start();
          Thread.sleep(10000);
          System.out.println(Thread.currentThread().getName() + " ends");
    }
}
```

Output:

```
main begins
Thread-0 begins
Thread-1 begins
main ends
I due thread Thread-0 e Thread-1 sono in deadlock
```

Consideriamo ora un main multithread. Abbiamo deadlock!

Esempio deadlock (discussione)

Vediamo come si genera il deadlock nel main multithread.!

- **`t1`** chiama il metodo **`synchronized maxRis`** su **`r1`**, prendendo il monitor di r1. Intanto t2 chiama **`maxRis`** su **`r2`** prendendo il monitor di **`r2`**.
- Il thread **`t1`**, eseguendo **`maxRis`** su **`r1`**, invoca il metodo **`synchornized getValore`** e su **`r2`**. Ma il monitor di r2 è in possesso di t2, quindi t1 si mette in attesa.
- Intanto **`t2`**, eseguendo **`maxRis`** su **`r2`**, invoca il metodo **`synchornized getValore`** su **`r1`**. Ma il monitor di **`r1`** è in possesso di **`t1`**, quindi anche **`t2`** si mette in attesa.
- **`t1`** aspetta **`t2`** che aspetta **`t1`**. Deadlock !!!

Esempio deadlock (discussione)

- Si noti che il **`main`** dopo 10 sec termina, mentre t1 e t2 rimangono running ma bloccati sui monitor di r2 e r1 rispettivamente.

- *NB: Se vogliamo che al termine del main terminino forzatamente anche t1 e t2, nel main dopo la creazione dei thread e prima di dargli lo start dobbiamo dichiararli come "demoni" attraverso l'istruzione setDeamon.Cioè nel main scriviamo:*
  - ***`t1.setDaemon(true);`***
  - ***`t2.setDaemon(true);`***

  I thread dichiarati demoni terminano forzatamente dopo la terminazione dei thread non demoni.

Nota: Synchronized è “rientrante” in Java

Si noti che in JAVA (come in tutti i linguaggi OO moderni) synchronized è “**rientrante**”.

Cioè **il thread che ha bloccato l'oggetto nell'eseguire del codice synchronized può invocare altri metodi synchronized (sullo stesso oggetto)** senza creare problemi di deadlock.

Infatti vengono bloccati gli **altri** thread che invocano metodi synchronized sullo stesso oggetto, non il thread che ha il lock.