Multithreading in Java
I parte

Lorenzo Gallucci

Java: un linguaggio ed un sistema predisposti per il multitasking

- È possibile creare e gestire Thread nativamente (senza dover ricorrere a librerie esterne)
- Sono disponibili la classe *Thread* e l'interfaccia *Runnable*
- Ogni programma Java è multithreaded
  - Un thread per l'esecuzione del main()
  - Uno o più thread per il garbage collector (finalization)
  - .
  - .
  - .
  - Altri thread creati dalle classi utente

Java: un linguaggio ed un sistema predisposti per il multitasking

- Per implementare il necessario coordinamento (“sincronizzazione”) tra thread, sono disponibili numerosi meccanismi
  - Il supporto alla sincronizzazione era presente nella classe Object fin dalla JDK 1.0
  - Nella JDK 1.5 e successive il supporto è stato esteso grazie all’aggiunta di diverse classi che implementano i meccanismi più comuni in letteratura

Dal monotasking al multitasking

- Si prenda esame una semplice classe **NumberPrinter** :

```
public class NumberPrinter
{
   int min;

   int max;

   public NumberPrinter(int min, int max)
   {
      this.min = min;
      this.max = max;
   }

   public void run()
   {
      for (int i = min; i <= max; i++)
         System.out.println(i);
   }
}
```

Dal monotasking al multitasking

- È possibile istanziare un oggetto di tipo NumberPrinter con parametri min e max
- L'invocazione del metodo run() su di esso produrrà in output l'elenco dei numeri compresi tra i valori min e max, uno per linea
- Ad esempio, esegueno la classe:

```
public class MonothreadNumberPrinterExample
{
    public static void main(String[] args)
    {
        NumberPrinter p = new NumberPrinter(4, 6);
        p.run();
        System.out.println("Done!");
    }
}
```

Dal monotasking al multitasking

- Si otterrà il seguente output

- 4
- 5
- 6
- Done!

Dal monotasking al multitasking

- L'istruzione che scrive su output la stringa "Done!" viene eseguita solo **dopo** che le istruzioni contenute nel metodo esegui sono state completamente eseguite.

- In particolare, il ciclo di for contenuto nel metodo esegui, non viene interrotto poiché non vi è alcun altro thread in esecuzione.

Dal monotasking al multitasking

- Scriviamo ora una classe simile a Printer, ma che effettua la stampa dei numeri come thread distinto da quello del main.

```
public class ThreadNumberPrinter extends Thread
{
   int min;

   int max;

   public ThreadNumberPrinter(int min, int max)
   {
      this.min = min;
      this.max = max;
   }

   public void run()
   {
      for (int i = min; i <= max; i++)
         System.out.println(i);
   }
}
```

Dal monotasking al multitasking

- L'istanziazione di un oggetto di tipo ThreadNumberPrinter corrisponde alla creazione di un nuovo thread, che, però, non è ancora in esecuzione
- La classe Thread esporta il metodo start(), la cui invocazione segnala alla Java Virtual Machine che il thread può essere posto in esecuzione.
- Quando lo scheduler della JVM metterà in esecuzione il thread, saranno eseguite sequenzialmete le istruzioni definite nel metodo run().

Dal monotasking al multitasking

- L'esecuzione della classe seguente:

```
public class MultithreadNumberPrinterExample
{
    public static void main(String[] args)
    {
        ThreadNumberPrinter tp = new
         ThreadNumberPrinter(4, 6);
        tp.start();
        System.out.println("Done!");
    }
}
```

Dal monotasking al multitasking

- … produce risultati ben diversi!

- Infatti:
  - il thread tp sarà pronto per essere eseguito, ma in modo distaccato dal thread del main
  - Il thread mian continuerà con l'istruzione che scrive su output la stringa "Done!".
- Pertanto, non possiamo prevedere in modo esatto quale sarà l'output su video. Vi sono le seguenti possibilità:

Scenario1	Scenario2	Scenario3	Scenario4
4 5 6 Done!	4 5 Done! 6	4 Done! 5 6	Done! 4 5 6

Dal monotasking al multitasking

- Nel primo caso il thread main lascia il controllo al thread tp, che viene quindi posto in esecuzione fino al termine delle istruzioni contenute nel metodo run.
- Nel secondo e nel terzo caso, il thread main lascia il controllo al thread tp, ma questa volta, prima che tp esaurisca le sue operazioni, il controllo viene ripassato al main, viene quindi eseguita l'istruzione di scrittura su video di "Done!", e quindi il controllo ripassa a tp, che termina le sue operazioni.
- Infine, nel quarto caso, il controllo non viene subito passato a tp, ma viene prima effettuata la scrittura su video di "Done!", e quindi viene posto in esecuzione tp, fino al completamento delle sue operazioni.
- Tutt'e quattro i casi sono, in linea di principio, egualmente probabili!

Dal monotasking al multitasking

- Ricapitolando, quando un programmatore vuole implementare un thread, può creare una classe che estende Thread, ridefinire il metodo run(), specificando quali sono le operazioni che il thread deve compiere.
- Successivamente, è possibile istanziare un oggetto del tipo definito e porlo in esecuzione mediante il metodo start.
- Si tenga presente che essendo il metodo run() pubblico, esso potrà essere invocato esplicitamente dal programmatore.
- Tuttavia, in tal caso, non verrebbe avviata l'esecuzione del nuovo thread creato, ma sarà esclusivamente eseguito il metodo run dal thread che lo invoca.

Cosa c'è dietro: i Runnable

- Come accennato in precedenza, è possibile utilizzare i thread di java anche utilizzando l'interfaccia Runnable.
- Questo meccanismo è indispensabile qualora si voglia creare una nuova classe con funzionalità di thread estendendo una classe diversa da Thread.
- Ad esempio, avendo implementato la classe NumberPrinter, potremmo implementare una classe NumberPrinterAsRunnable, che scriva i numeri in un thread a sè stante, implementando Runnable.
- L'interfaccia Runnable dichiara il metodo run, che pertanto dovrà essere definito da ogni classe che implementa quest'interfaccia.

Cosa c'è dietro: i Runnable

```
public class NumberPrinterAsRunnable implements Runnable
{
   int min;

   int max;

   public NumberPrinterAsRunnable(int min, int max)
   {
      this.min = min;
      this.max = max;
   }

   public void run()
   {
      for (int i = min; i <= max; i++)
         System.out.println(i);
   }
}
```

Runnable e Thread

- Creando un oggetto di tipo NumberPrinterAsRunnable, non abbiamo ancora la possibilità di avviarlo come thread.
- Infatti, il metodo start è esportato dalla classe Thread, e NumberPrinterAsRunnable non estende Thread, ma Runnable.
- È però possibile creare un oggetto Thread a partire da un oggetto Runnable.
- Infatti, Thread ammette un costruttore che riceve come parametro un oggetto Runnable.

Runnable e Thread

- Possiamo quindi creare un thread basato su NumberPrinterAsRunnable nel seguente modo

```
public class MultithreadNumberPrinterExampleWithRunnable
{
    public static void main(String[] args)
    {
        NumberPrinterAsRunnable p = new
    NumberPrinterAsRunnable(4, 6);
        Thread tp = new Thread(p);
        tp.start();
        System.out.println("Done!");
    }
}
```

Runnable e Thread

- Viene così creato e posto in attesa d'esecuzione un thread deputato ad eseguire il metodo run dell'oggetto p, ovvero dell'oggetto di tipo Runnable passato come parametro al costruttore.

Meccanismi di sincronizzazione

- Torniamo al primo esempio multitasking
- Vi erano diversi possibili output, a seconda dello scheduling del thread main e del thread secondario, “lanciato” nel main
- È possibile eliminare l’incertezza sull’output?
- È possibile forzare un particolare ordine di esecuzione?
- →Thread.join() !

Meccanismi di sincronizzazione

- Consideriamo la nuova classe main:

```
public class DeterministicMultithreadNumberPrinterExample
{
    public static void main(String[] args) throws
    InterruptedException
    {
        ThreadNumberPrinter tp = new ThreadNumberPrinter(4,
    6);
        tp.start();
        tp.join();
        System.out.println("Done!");
    }
}
```

Meccanismi di sincronizzazione

- Questa volta l'output può essere di un solo tipo:

- 4
- 5
- 6
- Done!

- L'invocazione del metodo join() sul thread appena lanciato fa sì che il controllo non ritorni al thread main, finchè tp non ha terminato la propria esecuzione

- Attenzione! Quest'attesa può essere interrotta, in tal caso viene lanciata una InterruptedException, che va opportunamente gestita

Sincronizzazione: un caso pratico

- Si abbia la classe:

```
public class Incrementer implements Runnable
{
   private final int[] data;

   public Incrementer(int[] data)
   {
      this.data = data;
   }

   public void run()
   {
      for (int i = 0; i < data.length; i++)
      {
         data[i] = data[i] + 1;
      }
   }

}
```

- Scopo della classe: ad ogni invocazione del metodo run(), tutti gli interi dell'array vengono incrementati

Sincronizzazione: un caso pratico

- Proviamo ad usare la classe Incrementer con un solo Thread, verificando che i singoli elementi siano stati effettivamente incrementati:

```
public class SingleIncrementerTest
{
   public static void main(String[] args) throws InterruptedException
   {
      int[] data = new int[10000];
      Thread thread = new Thread(new Incrementer(data));
      thread.start();
      thread.join();
      for (int i = 0; i < data.length; i++)
      {
         int j = data[i];
         if (j != 1) System.out.println("[" + i + "] = " + j);
      }
   }
}
```

- In questo caso, non vi sarà alcun output: ogni elemento dell'array sarà stato incrementato correttamente

Sincronizzazione: un caso pratico

- Complichiamo lo scenario, aggiungendo un un altro Thread "concorrente":

```
public class DoubleIncrementerTest
{
   public static void main(String[] args) throws InterruptedException
   {
      int[] data = new int[10000];
      Thread thread1 = new Thread(new Incrementer(data));
      Thread thread2 = new Thread(new Incrementer(data));
      thread1.start();
      thread2.start();
      thread2.join();
      thread1.join();
      for (int i = 0; i < data.length; i++)
      {
         int j = data[i];
         if (j != 2) System.out.println("[" + i + "] = " + j);
      }
   }
}
```

- In questo caso, è estremamente improbabile che venga rilevato qualche errore nell'incremento

Sincronizzazione: un caso pratico

- Parametrizziamo rispetto al numero di Thread:

```
public class IncrementerTest
{
   public static void main(String[] args) throws InterruptedException
   {
      int[] data = new int[10000];
      Thread[] threads = new Thread[1000];
      for (int i = 0; i < threads.length; i++)
      {
         threads[i] = new Thread(new Incrementer(data));
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].start();
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].join();
      }
      for (int i = 0; i < data.length; i++)
      {
         int j = data[i];
         if (j != threads.length) System.out.println("[" + i + "] = " + j);
      }
   }
}
```

Sincronizzazione: un caso pratico

- All'aumentare del numero di thread ed all'aumentare della durata del lavoro svolto dal thread, aumenta la probabilità di collisioni!
- Ad esempio, potremmo avere l'output:
- [317] = 999
- [435] = 998
- [962] = 999
- [1551] = 999
- [1636] = 999
- [1940] = 999
- …

Sincronizzazione: un caso pratico

- Cos'è avvenuto?
  - Alcuni thread si sono trovati a lavorare in parallelo sul medesimo elemento dell'array
  - Nella maggior parte dei casi, due o più incrementi concorrenti sono stati correttamente accodati
  - In alcuni casi (0,1-0,2%), gli incrementi concorrenti hanno dato origine ad una sovrapposizione non costruttiva: alcuni di essi sono andati persi
- **Accedere ad una medesima struttura dati da più Thread senza meccanismi di controllo della coerenza è una sicura fonte di errori!**

Sincronizzazione: un caso pratico

- Come garantire la correttezza?
- Modifichiamo l'Incrementer tramite l'uso della parola chiave synchronized

```
public class SynchronizedIncrementer implements Runnable
{
   private final int[] data;

   public SynchronizedIncrementer(int[] data)
   {
      this.data = data;
   }

   public void run()
   {
      synchronized (data)
      {
         for (int i = 0; i < data.length; i++)
         {
            data[i] = data[i] + 1;
         }
      }
   }
}
```

Sincronizzazione: un caso pratico

- La nuova classe main:

```
public class ControlledIncrementerTest
{
   public static void main(String[] args) throws InterruptedException
   {
      int[] data = new int[10000];
      Thread[] threads = new Thread[1000];
      for (int i = 0; i < threads.length; i++)
      {
         threads[i] = new Thread(new SynchronizedIncrementer(data));
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].start();
      }
      for (int i = 0; i < threads.length; i++)
      {
         threads[i].join();
      }
      for (int i = 0; i < data.length; i++)
      {
         int j = data[i];
         if (j != threads.length) System.out.println("[" + i + "] = " + j);
      }
   }
}
```

Sincronizzazione: un caso pratico

- Utilizzando il SynchronizedIncrementer, non abbiamo più interferenze tra i thread