Lezione 15

Programmazione multithreaded

Tutti i sistemi operativi, al giorno d'oggi, permettono di svolgere più operazioni simultaneamente. In questo momento, ad esempio, sto usando un word processor per scrivere la lezione, un lettore multimediale per ascoltare un CD Audio (Genesis, "Selling England By The Pound", un gran bel disco del '73), un programma di posta elettronica, che mi avvisa quando giungono nuove e-mail, ed un semplice editor per Java, che mi è utile per sperimentare gli esempi presenti nel manuale. Ho citato quattro differenti programmi che il mio sistema operativo esegue contemporaneamente, ma è probabile che ognuno di essi sia poi in grado di fare più cose in simultanea. Il lettore multimediale, ad esempio, ricerca in Internet le informazioni sul CD che sto ascoltando, senza però interrompere la sua riproduzione.
Un programma è *multithreaded* quando contiene due o più parti che vengono eseguite simultaneamente. La programmazione multithreaded, negli ultimi anni, ha assunto vitale importanza. Nonostante questo, non tutti i linguaggi godono di un supporto incorporato per la programmazione multithreaded. In molti casi, infatti, per eseguire più processi paralleli è necessario dialogare direttamente con il sistema operativo, chiedendogli di avviare le differenti parti del singolo programma. Java, ovviamente, ripudia questa tecnica, giacché legandosi troppo profondamente ad uno specifico sistema operativo perderebbe le sue doti di portabilità. Per questo motivo, Java vanta un supporto incorporato alla programmazione multithreaded, la cui interfaccia è totalmente indipendente dal sistema operativo in uso.

15.1 - La classe *Thread* e l'interfaccia *Runnable*

Il *thread* è l'unità atomica della programmazione multithreaded. Ogni processo in esecuzione viene racchiuso all'interno di un thread. Per eseguire contemporaneamente tre operazioni, quindi, è necessario creare ed avviare tre differenti thread. Compito dell'ambiente di esecuzione è gestire ogni singolo thread lanciato, garantendo la maggior efficienza possibile e risolvendo ogni problema di priorità, conflitto e sincronizzazione, secondo la volontà dal programmatore. Alla base di tutto questo, in Java, ci sono la classe *java.lang.Thread* e l'interfaccia *java.lang.Runnable*.
Quando un qualsiasi programma Java viene avviato, la macchina virtuale genera automaticamente un thread, all'interno del quale sarà richiamato il metodo *main()*, che dà il calcio d'inizio all'esecuzione. Il thread generato dalla macchina virtuale è chiamato thread principale, per diversi motivi:

- In ordine cronologico, è il primo thread dell'applicazione.
- Se si vuole generare un secondo thread, bisogna lanciarlo dall'interno del thread principale.
- Il programma si conclude quando termina il suo thread principale. Il thread principale è sempre l'ultimo a morire.

Se, ad un certo punto, si desidera abbandonare la linearità di esecuzione, per fare in modo che una seconda operazione venga processata indipendentemente dal thread principale, è necessario dichiarare e lanciare esplicitamente un thread secondario. Si possono creare nuovi thread in due differenti maniere: estendendo la classe *Thread* o implementando

l'interfaccia *Runnable*. La seconda tecnica, nella maggior parte dei casi, è da preferirsi. L'interfaccia *Runnable* richiede l'implementazione di un solo metodo:

```
public void run()
```

Il codice che viene inserito nel corpo del metodo *run()* può potenzialmente essere eseguito in un thread secondario. Dopo aver creato una classe che implementi l'interfaccia *Runnable*, è necessario istanziare un oggetto *Thread* che vi faccia riferimento:

```
Thread threadSecondario = new Thread(riferimentoAdUnOggettoRunnable);
```

A questo punto, il thread secondario è pronto per essere avviato:

```
threadSecondario.start();
```

L'esecuzione del thread secondario, come anticipato, comincerà dal metodo *run()* dell'oggetto *Runnable* riferito dall'istanza di *Thread*. La morte del thread secondario coincide con il termine dell'esecuzione del metodo *run()* avviato in precedenza.

Un qualsiasi thread può essere controllato, servendosi dei metodi della classe *Thread*. I più importanti di questo insieme sono riportati di seguito:

- ***static Thread currentThread()***. Restituisce un riferimento al thread corrente, cioè al thread che ha richiamato il metodo. Il metodo è statico, quindi non c'è bisogno di un riferimento preesistente ad un'istanza di *Thread*. E' il metodo stesso a fornirlo.
- ***String getName()***. Restituisce il nome del thread.
- ***int getPriority()***. Restituisce l'indice di priorità del thread.
- ***void interrupt()***. Propone l'interruzione del thread.
- ***static boolean interrupted()***. Restituisce *true* se il thread corrente è interrotto.
- ***boolean isAlive()***. Restituisce *true* se il thread è vivo.
- ***boolean isInterrupted()***. Restituisce *true* se il thread è interrotto.
- ***void join()***. Attende fino alla morte del thread sul quale è richiamato.
- ***void setName(String name)***. Imposta il nome del thread.
- ***void setPriority(int priority)***. Imposta l'indice di priorità del thread.
- ***static void sleep(long millis)***. Recupera il thread corrente e lo mette in pausa per millis millisecondi.
- ***void start()***. Avvia il thread.
- ***void yield()***. Mette temporaneamente in pausa il thread, in modo che gli altri thread dell'applicazione possano essere eseguiti.

I dettagli di molti dei metodi sopra mostrati saranno discussi in questa stessa lezione.

Prima di passare ad un esempio pratico, è bene riassumere in brevi punti quanto detto in questo paragrafo:

- Ogni programma Java ha un thread principale, che gli viene fornito dalla macchina virtuale. Al suo interno viene richiamato il metodo *main()* che avvia il programma.

Quando il thread principale muore, cioè quando *main()* esaurisce i suoi compiti, il programma termina.

- Per eseguire del codice in un thread secondario, bisogna realizzare una classe che estenda l'interfaccia *Runnable*. Questa interfaccia richiede la definizione del metodo *run()*. All'interno di *run()* va inserito il codice del thread secondario.
- Per dichiarare un thread secondario, bisogna creare un oggetto di tipo *Thread* che faccia riferimento ad un oggetto *Runnable*.
- Per avviare un thread secondario già dichiarato, basta richiamare su di esso il metodo *start()*. Sarà eseguito il metodo *run()* dell'oggetto *Runnable* riferito dal thread.
- Un qualsiasi thread, compreso quello principale, può essere controllato servendosi dei metodi della classe *Thread*.

15.2 - Un esempio pratico

Si metta in pratica quanto appreso nel corso del paragrafo precedente. Per prima cosa, si crei una classe che implementi l'interfaccia *Runnable*:

```
public class Secondario implements Runnable {

  public void run() {
    System.out.println("Ingresso nel thread secondario");
    try {
      for (int i = 0; i < 5; i++) {
        System.out.println("Secondario, ciclo " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Secondario interrotto");
    }
    System.out.println("Morte del thread secondario");
  }

}
```

Il codice contenuto nel metodo *run()* cicla cinque volte, emettendo un messaggio ogni mezzo secondo. Tutto il codice è stato racchiuso in una struttura *try ... catch*. Il metodo *sleep()* di *Thread*, infatti, può propagare una *InterruptedException*, nel caso il thread venga interrotto durante la pausa comandata. Gestire l'eventualità è sempre obbligatorio.
Si realizzi, ora, una classe dotata di un metodo *main()*, capace di eseguire contemporaneamente il thread principale ed un thread secondario riferito alla classe sopra creata:

```
public class Principale {

  public static void main(String[] args) {
    System.out.println("Ingresso nel thread principale");
    // Crea e avvia il thread secondario.
    Secondario secondario = new Secondario();
    Thread threadSecondario = new Thread(secondario);
    threadSecondario.start();
    // Compie delle operazioni nel thread principale.
    try {
```

```
      for (int i = 0; i < 5; i++) {
        System.out.println("Principale, ciclo " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Principale interrotto");
    }
    System.out.println("Morte del thread principale");
  }

}
```

Appena la classe viene avviata, il thread principale crea ed avvia il thread secondario, che è riferito alla classe *Secondario* realizzata in precedenza. Da questo momento in poi, i due thread sono eseguiti simultaneamente, l'uno indipendentemente dall'altro. Ambo i thread emettono dei messaggi, effettuando delle pause ad ogni ciclo. Il thread principale, ad ogni modo, si ferma per un secondo ad ogni ciclo, mentre il secondario resta fermo per la metà del tempo. Questo garantisce che il thread principale vivrà più a lungo del secondario, come la prassi desidera.
L'esecuzione del programma durerà all'incirca cinque secondi, producendo un output analogo al seguente (non per forza identico):

```
Ingresso nel thread principale
Principale, ciclo 0
Ingresso nel thread secondario
Secondario, ciclo 0
Secondario, ciclo 1
Principale, ciclo 1
Secondario, ciclo 2
Secondario, ciclo 3
Principale, ciclo 2
Secondario, ciclo 4
Morte del thread secondario
Principale, ciclo 3
Principale, ciclo 4
Morte del thread principale
```

Come si può osservare, l'ingresso nel thread secondario avviene (nel mio caso) quando il thread principale già ha eseguito il suo primo ciclo. Magari, ci si sarebbe aspettato il contrario. Ciò avviene poiché, dal momento stesso in cui è richiamato *threadSecondario.start()*, i due processi hanno vita indipendente. Evidentemente, mentre la macchina virtuale curava il corretto avvio del thread secondario, il codice principale ha fatto in tempo ad eseguire il suo primo ciclo. Non è detto che sia sempre così. Dipende in parte dal codice ed in parte dalla piattaforma in uso. I due processi hanno vita indipendente, e quindi è impossibile stabilire una relazione d'ordine fissa tra le istruzioni dell'uno e quelle dell'altro. Tutto dipende da svariati fattori esterni.

Ora si passerà ad uno speciale esperimento. Nel caso appena visto, si ha sempre la certezza che il thread principale viva più a lungo del secondario, grazie alle lunghe pause introdotte al suo interno. Bisogna sperimentare la situazione inversa. Si faccia in modo che il thread principale termini i propri compiti prima del secondario, abbreviando fino a 100 millisecondi le

sue pause:

```
public class Principale {

  public static void main(String[] args) {
    System.out.println("Ingresso nel thread principale");
    // Crea e avvia il thread secondario.
    Secondario secondario = new Secondario();
    Thread threadSecondario = new Thread(secondario);
    threadSecondario.start();
    // Compie delle operazioni nel thread principale.
    try {
      for (int i = 0; i < 5; i++) {
        System.out.println("Principale, ciclo " + i);
        Thread.sleep(100);
      }
    } catch (InterruptedException e) {
      System.out.println("Principale interrotto");
    }
    System.out.println("Morte del thread principale");
  }

}
```

Cosa accade compilando ed eseguendo il codice appena mostrato?

```
Ingresso nel thread principale
Principale, ciclo 0
Ingresso nel thread secondario
Secondario, ciclo 0
Principale, ciclo 1
Principale, ciclo 2
Principale, ciclo 3
Principale, ciclo 4
Secondario, ciclo 1
Morte del thread principale
Secondario, ciclo 2
Secondario, ciclo 3
Secondario, ciclo 4
Morte del thread secondario
```

Il messaggio di morte del thread principale non chiude l'esecuzione. Dopo di esso, si continuano a vedere i messaggi emessi dal thread secondario. Il programma è ancora in esecuzione. Apparentemente, questo è in contrasto con quanto detto poco sopra, riguardo alla longevità del thread principale. In realtà, non lo è. Quando il metodo *main()* termina, se c'è ancora qualche thread secondario in esecuzione, la macchina virtuale prolunga automaticamente la vita del principale, aspettando la morte dei processi da lui lanciati. In pratica, c'è una chiamata implicita a *threadSecondario.join()*.

Si passerà, ora, all'analisi del metodo *join()*. Si aggiunga una riga alla classe *Principale*:

```
public class Principale {

  public static void main(String[] args) {
```

```
    System.out.println("Ingresso nel thread principale");
    // Crea e avvia il thread secondario.
    Secondario secondario = new Secondario();
    Thread threadSecondario = new Thread(secondario);
    threadSecondario.start();
    // Compie delle operazioni nel thread principale.
    try {
      threadSecondario.join();
      for (int i = 0; i < 5; i++) {
        System.out.println("Principale, ciclo " + i);
        Thread.sleep(100);
      }
    } catch (InterruptedException e) {
      System.out.println("Principale interrotto");
    }
    System.out.println("Morte del thread principale");
  }

}
```

Il compito di *join()* è attendere fino alla morte del thread su cui è richiamato. La chiamata è stata introdotta all'interno della struttura *try ... catch*. Anche *join()*, infatti, può lanciare una *InterruptedException*, nel caso il thread venga interrotto nel corso dell'attesa. Si riscontra il seguente output:

```
Ingresso nel thread principale
Ingresso nel thread secondario
Secondario, ciclo 0
Secondario, ciclo 1
Secondario, ciclo 2
Secondario, ciclo 3
Secondario, ciclo 4
Morte del thread secondario
Principale, ciclo 0
Principale, ciclo 1
Principale, ciclo 2
Principale, ciclo 3
Principale, ciclo 4
Morte del thread principale
```

Come è possibile osservare, le istruzioni successive alla chiamata a *join()* non sono state considerate fin quando il thread secondario non è morto, come volevasi dimostrare.

15.3 - Priorità dei thread

Per parlare consapevolmente del concetto di priorità di un thread, è necessario scavare nel modello di gestione dei processi adottato da Java. Vi ricordate quando vi hanno detto che Babbo Natale non esiste? Sicuramente sarà stato un piccolo trauma accettare la realtà dei fatti. Beh, preparatevi a qualcosa di simile. A malincuore, vi rivelo che il reale multithreading non esiste. E' tutto un inganno! Dimostrarlo è semplice. Un calcolatore dotato di un solo processore non può compiere più di un'operazione alla volta. Semplicemente, il multithreading è un concetto astratto, che basa il proprio funzionamento sulla condivisione a tempo del medesimo apparato di calcolo. I processi non sono mai eseguiti contemporaneamente. I processi vengono eseguiti parzialmente, uno per volta. C'è una

specie di vigile urbano che controlla l'accesso alle risorse di calcolo da parte dei singoli processi. Ogni processo chiede, quando ne ha bisogno, di entrare nel processore. Se ci sono più processi in attesa di svolgere i propri compiti, si forma una fila. Uno alla volta, i thread accedono alle risorse di calcolo, eseguendo una piccola parte dei propri doveri. Immediatamente, escono e fanno posto agli altri processi in attesa. Se non hanno terminato il proprio lavoro, si rimettono pazientemente in fila. I computer sono veloci, mentre noi siamo lenti nel percepire. Per questo, ci sembra che più istruzioni vengano eseguite simultaneamente. Ecco svelata l'illusione ottica del multithreading.
Può capitare che un thread debba svolgere compiti più importanti di altri suoi colleghi. Quando i thread vengono creati, hanno tutti inizialmente un medesimo indice di priorità. Il programmatore, però, può assegnare indici più alti ai thread più importanti, ed indici più bassi a quelli di minore influenza. Gli indici di priorità influiscono sul tempo di impiego della CPU da parte di ogni singolo processo. Il vigile urbano (che, in realtà, si chiama *scheduler* dei thread) concede un uso delle risorse più duraturo ai thread con priorità maggiore, costringendo gli altri ad un'attesa più lunga.
La priorità associata ad un thread può essere impostata con un'istruzione del tipo:

```
riferimentoAlThread.setPriority(priority);
```

L'indice di priorità va specificato sotto forma di *int*. Il range del valore deve spaziare tra le costanti *Thread.MIN_PRIORITY* (priorità minima) e *Thread.MAX_PRIORITY* (priorità massima), estremi inclusi. Su un thread di massima importanza, quindi, è probabile che venga richiamato *setPriority()* al seguente modo:

```
riferimentoAlThread.setPriority(Thread.MAX_PRIORITY);
```

Inizialmente, tutti i thread ricevono una priorità media, che è rappresentata dalla costante *Thread.NORM_PRIORITY*.
Poiché il range degli indici può variare secondo l'implementazione della macchina virtuale, si sconsiglia di scrivere qualcosa del tipo:

```
riferimentoAlThread.setPriority(7);
```

Un valore letterale, anche se valido in una specifica macchina, potrebbe essere illegale in un'altra implementazione, futura o presente. Pertanto, le priorità vanno espresse servendosi delle tre costanti a disposizione.

L'indice di priorità di un thread può anche essere recuperato e valutato, servendosi del metodo *getPriority()*, che ovviamente restituisce un *int*.

Un altro metodo che influisce sull'accesso dei thread alle risorse di calcolo è *yield()*. Tornando alla metafora della fila fuori della porta della CPU, un thread su cui viene richiamato *yield()* esce dal locale delle risorse di calcolo e si mette all'ultimo posto della fila. In questo modo, gli altri thread possono essere eseguiti, prima che venga nuovamente il suo turno.

15.4 - Vita, morte ed interruzione di un thread

La vita di un thread comincia quando su di esso viene invocato il metodo *start()* e termina

quando il metodo *run()* dell'oggetto *Runnable* collegato al processo completa i propri doveri. In ogni momento, è possibile sapere se uno specifico thread è vivo, invocando su di esso il metodo *isAlive()*. Il metodo restituisce *true* se il processo è in vita, *false* altrimenti. Si consideri il seguente esempio:

```
public class Test1 implements Runnable {

  public void run() {
    System.out.println("Ingresso nel thread secondario");
      try {
        Thread.sleep(5000);
      } catch (InterruptedException e) {
        System.out.println("Thread secondario interrotto");
      }
    System.out.println("Morte del thread secondario");
  }

  public static void main(String[] args) {
    System.out.println("Ingresso nel thread principale");
    Test1 test1 = new Test1();
    Thread secondario = new Thread(test1);
    secondario.start();
    try {
      while (secondario.isAlive()) {
        System.out.println("Il thread secondario è vivo");
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Thread principale interrotto");
    }
    System.out.println("Morte del thread principale");
  }

}
```

Questa volta i due thread sono stati compressi all'interno di una sola classe. Si tratta di una pratica abbastanza comune. Indipendentemente da ciò, nel thread principale è stato introdotto un ciclo che verifica, ad intervalli di un secondo, lo stato vitale del thread secondario. Giacché quest'ultimo vive all'incirca cinque secondi, ci si aspetta di ricevere per circa cinque volte il messaggio "Il thread secondario è vivo", prima che il programma termini. In effetti, accade proprio così:

```
Ingresso nel thread principale
Il thread secondario è vivo
Ingresso nel thread secondario
Il thread secondario è vivo
Il thread secondario è vivo
Il thread secondario è vivo
Il thread secondario è vivo
Morte del thread secondario
Morte del thread principale
```

Finché un processo è vivo, rimane sempre possibile notificargli un invito all'interruzione, con il metodo *interrupt()*:

```
public class Test2 implements Runnable {

  public void run() {
    System.out.println("Ingresso nel thread secondario");
    try {
      Thread.sleep(5000);
    } catch (InterruptedException e) {
      System.out.println("Thread secondario interrotto");
    }
    System.out.println("Morte del thread secondario");
  }

  public static void main(String[] args) {
    System.out.println("Ingresso nel thread principale");
    Test2 test2 = new Test2();
    Thread secondario = new Thread(test2);
    secondario.start();
    try {
      Thread.sleep(1000);
      secondario.interrupt();
    } catch (InterruptedException e) {
      System.out.println("Thread principale interrotto");
    }
    System.out.println("Morte del thread principale");
  }

}
```

In questo caso, si chiede l'interruzione del thread secondario dopo un secondo dal suo avvio. Quando questo accade, nel thread secondario si propaga una *InterruptedException*:

```
Ingresso nel thread principale
Ingresso nel thread secondario
Thread secondario interrotto
Morte del thread secondario
Morte del thread principale
```

Il metodo *isInterrupted()* permette di verificare se un thread è marcato come interrotto. Ancora un esempio, questa volta basato sul controllo di *isInterrupted()*, anziché sulla gestione delle eccezioni:

```
public class Test3 implements Runnable {

  public void run() {
    System.out.println("Ingresso nel thread secondario");
    while (!Thread.currentThread().isInterrupted()) {}
    System.out.println("Morte del thread secondario");
  }

  public static void main(String[] args) {
    System.out.println("Ingresso nel thread principale");
    Test3 test3 = new Test3();
    Thread secondario = new Thread(test3);
    secondario.start();
```

```
    try {
      Thread.sleep(1000);
      secondario.interrupt();
    } catch (InterruptedException e) {
      System.out.println("Thread principale interrotto");
    }
    System.out.println("Morte del thread principale");
  }

}
```

Il thread secondario, in questo terzo esempio, rimane vivo e in esecuzione fin quando non lo si interrompe manualmente, a causa del loop:

```
while (!Thread.currentThread().isInterrupted()) {}
```

Il thread principale, dopo un secondo di attesa, chiede al secondario di interrompersi. Pertanto, il ciclo *while* termina la propria esecuzione, ed il thread muore naturalmente, non appena le istruzioni di *run()* sono finite.

Una scorciatoia statica alla dicitura

```
Thread.currentThread().isInterrupted()
```

è

```
Thread.interrupted()
```

Stratagemmi come quelli appena mostrati permettono di agevolare l'uscita da un thread non più utile.

Tecniche obsolete
Nelle versioni obsolete di Java si poteva approfittare di speciali metodi per la brusca interruzione di un thread, ed anche per la sua ripresa. Tuttavia, si trattava di un cattivo modello di gestione, che creava problemi per la sincronizzazione ed il rilascio delle risorse. Pertanto, oggi sono considerati deprecati. Per retrocompatibilità con i vecchi programmi, continuano ad essere implementati nelle nuove macchine virtuali, ma il loro utilizzo è sconsigliato in ogni nuovo software. Pertanto, non sono trattati in questo corso.

15.5 - Sincronizzazione

La programmazione multithreaded causa una serie di problemi che vanno sempre considerati. Il più importante di questi problemi è l'accesso alle risorse condivise. Si supponga che due thread distinti ed indipendenti debbano accedere ad un unico file, in scrittura. Appare immediatamente evidente come i due processi non possano simultaneamente scrivere all'interno del medesimo file. In questo caso, infatti, si otterrebbero dati imprevedibili. Bisogna fare in modo che i singoli thread si mettano in fila, ed accedano alla risorsa uno per volta, senza sovrapporsi. Il problema può essere riscontrato facilmente, anche senza servirsi di un file esterno. Basta sperimentare il seguente esempio:

```
public class Test1 implements Runnable {
```

```
  public static void scrivi(char[] c) {
    try {
      for (int i = 0; i < c.length; i++) {
        System.out.print(c[i]);
        Thread.sleep(200);
      }
      System.out.println();
    } catch (InterruptedException e) {}
  }

  public void run() {
    char[] caratteri = {
      's', 'e', 'c', 'o', 'n', 'd', 'a', 'r', 'i', 'o'
    };
    scrivi(caratteri);
  }

  public static void main(String[] args) {
    // Avvio del thread secondario.
    Test1 test1 = new Test1();
    Thread secondario = new Thread(test1);
    secondario.start();
    // Scrittura dal thread principale.
    char[] caratteri = {
      'p', 'r', 'i', 'n', 'c', 'i', 'p', 'a', 'l', 'e'
    };
    scrivi(caratteri);
  }

}
```

Questa classe gestisce due thread. Il processo principale, servendosi del metodo *scrivi()*, cerca di mandare in output la scritta "principale". Il thread secondario, simultaneamente, comanda la scrittura della stringa "secondario". Il metodo *scrivi()* impiega un po' di tempo per completare i propri doveri, giacché fa apparire una lettera alla volta, con un effetto "macchina da scrivere". Benché questa sia solo una simulazione, non è raro che l'accesso ad una risorsa esterna possa abbracciare tempi abbastanza estesi da esporsi al rischio della sovrapposizione. Poiché l'accesso a *scrivi()* avviene contemporaneamente da parte di entrambi i thread, il risultato finale mostrato in output è piuttosto impasticciato. Qualcosa come:

```
psreicnocnidpaarlieo
```

Altri linguaggi, al contrario di Java, non dispongono di meccanismi incorporati per la sincronizzazione dei thread. In casi come questo, bisogna appellarsi alle primitive del sistema operativo. Java, invece, risolve il problema in maniera semplice e pulita, assolutamente multipiattaforma. I problemi di concorrenza possono essere risolti in due maniere distinte, entrambe basate sull'impiego della parola chiave *synchronized*.

La prima tecnica consiste nel rendere *sincronizzati* (*synchronized*, appunto) i metodi che gestiscono l'accesso alle risorse condivise. Un metodo diventa sincronizzato quando si

introduce la parola chiave *synchronized* nella sua riga di definizione:

```
synchronized tipo-restituito nomeMetodo(lista-argomenti)
```

Si riprenda l'esempio precedente, rendendo sincronizzato il metodo *scrivi()*:

```
public class Test2 implements Runnable {

  public static synchronized void scrivi(char[] c) {
    try {
      for (int i = 0; i < c.length; i++) {
        System.out.print(c[i]);
        Thread.sleep(200);
      }
      System.out.println();
    } catch (InterruptedException e) {}
  }

  public void run() {
    char[] caratteri = {
      's', 'e', 'c', 'o', 'n', 'd', 'a', 'r', 'i', 'o'
    };
    scrivi(caratteri);
  }

  public static void main(String[] args) {
    // Avvio del thread secondario.
    Test1 test1 = new Test1();
    Thread secondario = new Thread(test1);
    secondario.start();
    // Scrittura dal thread principale.
    char[] caratteri = {
      'p', 'r', 'i', 'n', 'c', 'i', 'p', 'a', 'l', 'e'
    };
    scrivi(caratteri);
  }

}
```

Ecco l'output prodotto:

```
principale
secondario
```

Finalmente, tutto appare in ordine.

I metodi sincronizzati di un oggetto vengono tenuti d'occhio da un *monitor*. Quando un thread accede ad uno dei metodi sincronizzati dell'oggetto, si dice che il processo è all'interno del monitor. Tutti i metodi sincronizzati dell'oggetto sono a disposizione del solo thread che è all'interno del monitor. Se un secondo thread richiama uno dei metodi sottoposti al blocco, la macchina virtuale lo metterà automaticamente in pausa, fin quando il monitor non sarà libero. Quindi, il secondo processo dovrà pazientemente attendere che il suo collega esca dal monitor, cedendogli il posto. L'uscita dal monitor, e la conseguente rimozione del blocco,

coincide con il termine dell'esecuzione del metodo sincronizzato richiamato.

La seconda tecnica consiste nell'etichettare come sincronizzati interi blocchi di codice:

```
synchronized (oggetto) {
  // Codice sincronizzato
}
```

Una struttura come questa rende automaticamente sincronizzati tutti i metodi di oggetto. E' possibile riscrivere ancora una volta l'ultimo esempio, al seguente modo:

```
public class Test3 implements Runnable {

  public static void scrivi(char[] c) {
    try {
      for (int i = 0; i < c.length; i++) {
        System.out.print(c[i]);
        Thread.sleep(200);
      }
      System.out.println();
    } catch (InterruptedException e) {}
  }

  public void run() {
    char[] caratteri = {
      's', 'e', 'c', 'o', 'n', 'd', 'a', 'r', 'i', 'o'
    };
    synchronized (this) {
      scrivi(caratteri);
    }
  }

  public static void main(String[] args) {
    // Avvio del thread secondario.
    Test1 test1 = new Test1();
    Thread secondario = new Thread(test1);
    secondario.start();
    // Scrittura dal thread principale.
    char[] caratteri = {
      'p', 'r', 'i', 'n', 'c', 'i', 'p', 'a', 'l', 'e'
    };
    synchronized (test3) {
      scrivi(caratteri);
    }
  }

}
```

Generalmente, la prima tecnica mostrata è da preferirsi alla seconda. L'utilizzo di blocchi *synchronized* è stato previsto solo per fronteggiare delle situazioni peculiari. Si supponga di doversi servire di una classe che non è stata progettata in vista di un suo utilizzo multithreaded. Alcuni tra i suoi metodi, però, accedono a delle risorse condivise, ed esiste il rischio di fare danno quando due o più thread vi entrano simultaneamente. Inoltre, non si è tra gli autori della classe, e non si ha neanche a disposizione il suo codice sorgente. Pertanto,

non è possibile effettuare delle modifiche, e non è possibile rendere sincronizzati i metodi critici. Non resta che richiamare i metodi delle sue istanze ponendosi all'interno dei blocchi *synchronized*.

15.6 - Raffinare la sincronizzazione

La tecnica di sincronizzazione incorporata in Java può essere ulteriormente raffinata dal programmatore, anche se questo avviene soltanto in casi molto peculiari. *Object*, che è superclasse di qualsiasi classe esistente in Java, definisce tre metodi finali (cioè che nessuna sua sottoclasse può ridefinire), chiamati *wait()*, *notify()* e *notifyAll()*. Di conseguenza, tutti gli oggetti di Java hanno questi tre metodi. Il loro effetto è illustrato di seguito:

- ***wait()*** comunica al thread chiamante di uscire dal monitor che blocca l'accesso ai metodi sincronizzati dell'oggetto. Pertanto, può essere richiamato soltanto dal metodo che è nel monitor dell'oggetto (in caso contrario, si riceve un errore di runtime). Il thread sarà automaticamente messo in pausa. Nel frattempo, un secondo processo potrà entrare nel monitor. Il primo thread resterà in pausa fin quando il secondo processo non richiamerà *notify()* o *notifyAll()* sullo stesso oggetto.
- ***notify()*** risveglia il primo thread che ha chiamato *wait()* sullo stesso oggetto.
- ***notifyAll()*** risveglia tutti i thread che hanno chiamato *wait()* sullo stesso oggetto. L'ordine del loro rientro nel monitor dipende dai singoli indici di priorità.

Inoltre, esistono delle varianti di *wait()* che permettono di impostare un tempo di attesa massimo.

15.7 - Riflessioni conclusive sulla programmazione multithreaded

La programmazione multithreaded fornisce strumenti molto interessanti per la realizzazione di programmi di stampo moderno. Basta pensare alle interfacce grafiche basate sulle finestre, che spesso e volentieri funzionano proprio grazie ai thread concorrenti. Inoltre, sfruttando sapientemente il multithreading, è possibile migliorare le prestazioni di programmi altrimenti organizzati lungo una sola linea di esecuzione. Tuttavia, questa caratteristica va usata con parsimonia: l'inizializzazione, l'avvio ed il controllo di ogni singolo thread consumano diversi cicli di calcolo. Se si esagera nel suddividere il proprio programma in più linee concorrenti, si corre il rischio di diminuire le prestazioni, anziché aumentarle. Dunque, si faccia uso ricorrente della programmazione multithreaded, ma senza dimenticarsi di ragionare a lungo sulle scelte fatte, spendendo parecchio tempo in una progettazione chiara e certa.