Capitolo 11

Programmazione concorrente e gestione del multithread in Java

Paolo Aiello, Giovanni Puliti

Introduzione

Una delle potenti caratteristiche del linguaggio Java è il supporto per la *programmazione concorrente* o *parallela*. Tale *feature* permette di organizzare il codice di una stessa applicazione in modo che possano essere mandate in esecuzione contemporanea più parti di codice differenti fra loro.

Prima di descrivere questi aspetti del linguaggio saranno introdotti alcuni concetti fondamentali che aiuteranno ad avere un'idea più chiara dell'argomento e delle problematiche correlate.

Processi e multitasking

Tutti i moderni sistemi operativi offrono il supporto per il *multitasking*, ossia permettono l'esecuzione simultanea di più *processi*. In un sistema Windows, Unix o Linux si può, ad esempio, scrivere una e-mail o un documento di testo mentre si effettua il download di un file da Internet. In apparenza questi diversi programmi vengono eseguiti contemporaneamente, anche se il computer è dotato di un solo processore.

In realtà i processori dei calcolatori su cui si è abituati a lavorare analizzano il flusso delle istruzioni in maniera sequenziale in modo che in ogni istante una sola istruzione sia presa in esame ed eseguita (questo almeno in linea di massima, dato che esistono architetture particolari che permettono il parallelismo a livello di microistruzioni).

Ma anche se, per sua natura, un computer è una macchina sequenziale, grazie a una gestione

ciclica delle risorse condivise (prima fra tutte il processore centrale), si ottiene una specie di parallelismo, che permette di simulare l'esecuzione contemporanea di più programmi nello stesso momento.

Grazie alla elevata ottimizzazione degli algoritmi di gestione di questo pseudoparallelismo, e grazie alla possibilità di un processo di effettuare certi compiti quando gli altri sono in pausa o non sprecano tempo di processore, si ha in effetti una simulazione del parallelismo fra processi, anche se le risorse condivise sono in numero limitato.

Nel caso in cui si abbiano diversi processori operanti in parallelo, è possibile che il parallelismo sia reale, nel senso che un processore potrebbe eseguire un processo mentre un altro processore esegue un diverso processo, senza ripartizione del tempo: in generale non è possibile però fare una simile assunzione, dato che normalmente il numero di processi in esecuzione è maggiore (o comunque può esserlo) del numero di processori fisici disponibili, per cui è sempre necessario implementare un qualche meccanismo di condivisione delle risorse.

Un processo è un flusso di esecuzione del processore corrispondente a un programma. Il concetto di processo va però distinto da quello di programma in esecuzione, perché è possibile che un processore esegua contemporaneamente diverse istanze dello stesso programma, ossia generi diversi processi che eseguono lo stesso programma (ad esempio diverse istanze del Notepad, con documenti diversi, in ambiente Windows).
Per multitasking si intende la caratteristica di un sistema operativo di permettere l'esecuzione contemporanea (o pseudocontemporanea, per mezzo del time-slicing) di diversi processi.

Vi sono due tipi di multitasking:

il *cooperative multitasking* la cui gestione è affidata agli stessi processi, che mantengono il controllo del processore fino a che non lo rilasciano esplicitamente. Si tratta di una tecnica abbastanza rudimentale in cui il funzionamento dipende dalla bontà dal codice del programma, quindi in sostanza dal programmatore. C'è sempre la possibilità che un programma scritto in modo inadeguato monopolizzi le risorse impedendo il reale funzionamento multitasking. Esempi di sistemi che usano questo tipo di multitasking sono Microsoft Windows 3.x e alcune versioni del MacOS;

il *preemptive multitasking* è invece gestito interamente dal sistema operativo con il sistema del *time-slicing* (detto anche *time-sharing*), assegnando ad ogni processo un intervallo di tempo predefinito, ed effettuando il cambio di contesto anche senza che il processo intervenga o ne sia a conoscenza. Il processo ha sempre la possibilità di rilasciare volontariamente le risorse, ma questo non è necessario per il funzionamento del sistema. Il sistema operativo in questo caso utilizza una serie di meccanismi per il controllo e la gestione del tempo del processore, in modo da tener conto di una serie di parametri, legati al tempo trascorso e all'importanza (*priorità*) di un determinato processo.

Nonostante il fatto che i termini preemptive e time-slicing abbiano significato simile, in realtà preemptive si riferisce alla capacità di un processo di "prevalere" su un altro di minore priorità sottraendogli il processore in base a tale "diritto di priorità", mentre il time-slicing, anche se generalmente coesiste con la preemption, si riferisce unicamente alla suddivisione del tempo gestita dal sistema (e non lasciata ai processi), anche tra processi a priorità uguale. Lo scheduling usato per gestire i processi a uguale priorità è generalmente il cosiddetto round-robin scheduling, in cui un processo, dopo che ha usufruito della sua porzione di tempo, viene messo in attesa in coda fra processi con uguale priorità. Sia la preemption che il time-slicing presuppongono un intervento da parte del sistema operativo nel determinare quale processo deve essere mandato in esecuzione. Comunque possono esserci sistemi preemptive che non usano il time-slicing, ma usano ugualmente le priorità per determinare quale processo deve essere eseguito. Si tornerà su questo aspetto a proposito della gestione dei thread in Java.

Si è detto che per simulare il parallelismo fra processi differenti si effettua una spartizione del tempo trascorso in esecuzione nel processore. Il meccanismo di simulazione si basa sul cambio di contesto (*context-switch*) fra processi diversi: in ogni istante un solo processo viene messo in esecuzione, mentre gli altri restano in attesa.

Il contesto di un processo P1 è l'insieme delle informazioni necessarie per ristabilire esattamente lo stato in cui si trova il sistema al momento in cui interrompe l'esecuzione del processo P1 per passare a un altro processo P2. Tra queste informazioni di contesto le principali sono lo *stato dei registri del processore*, e la *memoria del processo*, che a sua volta contiene il *testo* del programma, ossia la sequenza di istruzioni, i *dati* gestiti dal processo e lo *stack* (spazio di memoria per le chiamate di funzioni e le variabili locali).

Infatti, un aspetto fondamentale della gestione dei processi è il fatto che *ogni processo ha un suo spazio di memoria privato*, a cui esso soltanto può accedere. Quindi, salvo casi eccezionali (memoria condivisa) un processo non ha accesso alla memoria gestita da un altro processo.

I processi sono normalmente organizzati secondo una struttura gerarchica in cui, a partire da un primo processo iniziale creato alla partenza del sistema operativo, ogni successivo processo è "figlio" di un altro processo che lo crea e che ne diviene il "padre".

Nei sistemi preemptive vi è poi un processo particolare che gestisce tutti gli altri processi, lo *scheduler*, responsabile della corretta distribuzione del tempo della CPU tra i processi in esecuzione. A tale scopo esistono diversi algoritmi di *scheduling*, che comunque generalmente si basano sul tempo di attesa (maggiore è il tempo trascorso dall'ultima esecuzione, maggiore è la priorità del processo) e su livelli di priorità intrinseci assegnati dal sistema sulla base della natura del processo, oppure dall'utente sulla base delle sue esigenze particolari. A prescindere da questo normale avvicendamento di esecuzione, i processi possono subire delle interruzioni (*interrupt*) dovute al verificarsi di eventi particolari, originati dall'hardware come l'input di una periferica (interrupt hardware), dal software (interrupt software) oppure da errori di esecuzione che causano le cosiddette *eccezioni*. In questi casi viene effettuato un context-switch come nel normale scheduling, viene eseguito del codice specifico che gestisce l'interruzione, dopodiché si torna al processo interrotto con un altro context-switch. I processi, durante il loro ciclo di vita, assumono *stati* differenti, in conseguenza del loro funzionamento interno

e dell'attività dello scheduler. Semplificando al massimo, questi sono i principali stati che un processo può assumere:

- *in esecuzione*: il processo è attualmente in esecuzione;
- *eseguibile*: il processo non è in esecuzione, ma è pronto per essere eseguito, appena la CPU si rende disponibile;
- *in attesa*: il processo è in attesa di un dato evento, come lo scadere di una frazione di tempo, la terminazione di un altro processo, l'invio di dati da un canale I/O.

I processi normalmente sono entità tra loro separate ed estranee ma, qualora risultasse opportuno, sono in grado di comunicare tra di loro utilizzando mezzi di comunicazione appositamente concepiti, genericamente identificati dalla sigla IPC (Inter Process Communication). Tra questi si possono citare la memoria condivisa (*shared-memory*), i *pipe*, i *segnali*, i *messaggi*, i *socket*. A seconda della tipologia di comunicazione tra processi, possono sorgere dei problemi derivanti dall'acceso contemporaneo, diretto o indiretto, alle medesime risorse. Per evitare che questo dia origine a errori e incongruenze, generalmente le risorse vengono acquisite da un singolo processo con un *lock*, e rilasciate una volta che l'operazione è terminata. Solo a quel punto la risorsa sarà disponibile per gli altri processi. Per gestire questo tipo di problemi di *sincronizzazione* esistono appositi meccanismi, tra cui il più conosciuto è quello dei *semafori*. Per maggiori approfondimenti legati a questi argomenti si faccia riferimento alla bibliografia.

Thread e multithreading

L'esecuzione parallela e contemporanea di più tasks (intendendo per task l'esecuzione di un compito in particolare), risulta utile non solo nel caso di processi in esecuzione su un sistema operativo multitasking, ma anche all'interno di un singolo processo.

Si pensi, ad esempio, a un editor di testo in cui il documento corrente viene automaticamente salvato su disco ogni *n* minuti. In questo caso il programma è composto da due flussi di esecuzione indipendenti tra loro: da un lato l'editor che raccoglie i dati in input e li inserisce nel documento, dall'altra il meccanismo di salvataggio automatico che resta in attesa per la maggior parte del tempo e, a intervalli prestabiliti, esegue la sua azione.

Sulla base di simili considerazioni è nata l'esigenza di poter usare la programmazione concorrente all'interno di un singolo processo, e sono stati concepiti i *thread*, i quali in gran parte replicano il modello dei processi concorrenti, applicato però nell'ambito di una singola applicazione. Un processo quindi non è più un singolo flusso di esecuzione, ma un insieme di flussi: ogni processo contiene almeno un thread (thread principale) e può dare origine ad altri thread generati a partire dal thread principale.

Come per il multitasking, anche nel multithreading lo scheduling dei thread può essere compiuto dal processo (dal thread principale), eventualmente appoggiandosi ai servizi offerti dal sistema operativo, se questo adotta il *time-slicing*; in alternativa può essere affidato ai singoli thread, ed allora il programmatore deve fare attenzione a evitare che un singolo thread mono-

polizzi le risorse, rilasciandole periodicamente secondo criteri efficienti.

La differenza fondamentale tra processi e thread sta nel fatto che i thread *condividono lo stesso spazio di memoria*, se si prescinde dallo stack, ossia dai dati temporanei e locali usati dalle funzioni.

Questo porta diverse conseguenze: il cambio di contesto fra thread è molto meno pesante di quello tra processi, e quindi l'uso di thread diversi causa un dispendio di risorse inferiore rispetto a quello di processi diversi; inoltre la comunicazione fra thread è molto più semplice da gestire, dato che si ha condivisione dello stesso spazio di memoria.

D'altra parte, proprio questa condivisione rende molto più rilevanti e frequenti i problemi di sincronizzazione, come si vedrà dettagliatamente in seguito.

Un thread è un flusso di esecuzione del processore corrispondente a una sequenza di istruzioni all'interno di un processo. Analogamente ai processi, bisogna distinguere il concetto di esecuzione di una sequenza di istruzioni da quello di thread, poiché ci possono essere diverse esecuzioni parallele di uno stesso codice, che danno origine a thread diversi.

Per multithreading si intende l'esecuzione contemporanea (ovvero pseudocontemporanea, per mezzo del time-sharing) di diversi thread nell'ambito dello stesso processo. La gestione del multithreading può essere a carico del sistema operativo, se questo supporta i thread, ma può anche essere assunta dal processo stesso.

I thread e la Java Virtual Machine

Si è visto che un thread è un flusso di esecuzione nell'ambito di un processo. Nel caso di Java, ogni esecuzione della macchina virtuale dà origine a un processo, e tutto quello che viene mandato in esecuzione da una macchina virtuale (ad esempio un'applicazione o una Applet) dà origine a un thread.

La virtual machine Java è però un processo un po' particolare, in quanto funge da ambiente portabile per l'esecuzione di applicazioni su piattaforme differenti. Quindi la JVM non può fare affidamento su un supporto dei thread da parte del sistema operativo, ma deve comunque garantire un certo livello minimo di supporto, stabilito dalle specifiche ufficiali della virtual machine. Queste stabiliscono che una VM gestisca i thread secondo uno scheduling di tipo preemptive chiamato *fixed-priority scheduling*. Questo schema è basato essenzialmente sulla priorità ed è preemptive perché garantisce che, se in qualunque momento si rende eseguibile un thread con priorità maggiore di quella del thread attualmente in esecuzione, il thread a maggiore priorità prevalga sull'altro, assumendo il controllo del processore.

La garanzia che sia sempre in esecuzione il thread a priorità più alta non è assoluta. In casi particolari lo scheduler può mandare in esecuzione un thread a priorità più bassa per evitare situazioni di stallo o un consumo eccessivo di risorse. Per questo motivo è bene non fare affidamento su questo comportamento per assicurare il corretto funzionamento di un algoritmo dal fatto che un thread ad alta priorità prevalga sempre su uno a bassa priorità.

Le specifiche della VM non richiedono il time-slicing nella gestione dei thread, anche se questo è in realtà presente nei più diffusi sistemi operativi e, di conseguenza può essere utilizzato dalle VM che girano su questi sistemi. Per questo motivo, se si vuole che un'applicazione Java funzioni correttamente indipendentemente dal sistema operativo e dalla implementazione della VM, non si deve assumere la gestione del time-sharing da parte della VM, ma bisogna far sì che ogni thread rilasci spontaneamente le risorse quando opportuno. Quest'aspetto sarà analizzato nei dettagli più avanti.

Si diceva che generalmente le VM usano i servizi di sistema relativi ai thread, se presenti. Ma ciò non è tassativo. Una macchina virtuale può anche farsi interamente carico della gestione dei thread, senza far intervenire il sistema operativo, anche se questo supporta i thread.In questo caso la VM è vista dal sistema come un processo con un singolo thread, mentre i thread Java sono ignorati totalmente dal sistema stesso. Questo modello di implementazione dei thread nella VM è conosciuto come *green-thread* (*green* in questo caso è traducibile approssimativamente con *semplice*) ed è adottato da diverse implementazioni della VM, anche in sistemi (ad esempio alcune versioni di Unix) in cui esiste un supporto nativo dei thread. Viceversa in ambiente Windows, le VM usano generalmente i servizi del sistema operativo. Analogamente ai processi, i thread assumono in ogni istante un determinato stato. Nella VM si distinguono i seguenti stati dei thread:

- *initial*: un thread si trova in questa condizione tra il momento in cui viene creato e il momento in cui comincia effettivamente a funzionare;
- *runnable*: è lo stato in cui si trova normalmente un thread dopo che ha cominciato a funzionare. Il thread in questo stato può, in qualunque momento, essere eseguito;
- *running*: il thread è attualmente in esecuzione. Questo non sempre viene considerato uno stato a sé, ma in effetti si tratta di una condizione diversa dallo stato runnable. Infatti ci possono essere diversi thread nello stato runnable in un dato istante ma, in un sistema a singola CPU, uno solo è in esecuzione, e viene chiamato *thread corrente*;
- *blocked*: il thread è in attesa di un determinato evento;
- *dead*: il thread ha terminato la sua esecuzione.

La programmazione concorrente in Java

Dopo tale panoramica su programmazione parallela e thread, si può analizzare comeutilizzare i thread in Java. Gli strumenti a disposizione per la gestione dei thread sono essenzialmente due: la classe `java.lang.Thread` e l'interfaccia `java.lang.Runnable`.

Dal punto di vista del programmatore, i thread in Java sono rappresentati da oggetti che sono o istanze della classe `Thread`, o istanze di una sua sottoclasse, oppure oggetti che implementano l'interfaccia `Runnable`. D'ora in avanti si utilizzerà il termine thread sia per indicare il concetto di thread, sia per far riferimento alla classe `Thread` che a una qualsiasi classe che implementi le funzionalità di un thread.

Creazione e terminazione di un thread

Inizialmente verrà presa in esame la modalità di creazione e gestione dei thread basata sull'utilizzo della classe Thread, mentre in seguito sarà analizzata la soluzione alternativa basata sull'interfaccia Runnable.

La classe Thread è una classe *non astratta* attraverso la quale si accede a tutte le principali funzionalità per la gestione dei thread, compresa la creazione dei thread stessi. Il codice necessario per creare un thread è il seguente:

```
Thread myThread = new Thread();
```

A meno di associarvi un oggetto Runnable, istanziando direttamente un oggetto della classe Thread però non si ottiene nessun particolare risultato, dato che esso termina il suo funzionamento quasi subito: infatti le operazioni svolte in modalità threaded sono quelle specificate nel metodo run(), metodo che deve essere ridefinito dalle classi derivate.

Se si desidera quindi che il thread faccia qualcosa di utile ed interessante, si deve creare una sottoclasse di Thread, e ridefinire il metodo run(). Qui di seguito è riportato un esempio

```
public class SimpleThread extends Thread {
 String message;

 public SimpleThread(String s){
 message = s;
 }

 public void run() {
    for (int i = 0; i < 100; i++)
    System.out.println(message);
 }

 public static void main(String[] args) {
    SimpleThread st1, st2;
    st1 = new SimpleThread("Buongiorno");
    st2 = new SimpleThread("Buonasera");
    st1.start();
    st2.start();
 }
}
```

Per far partire il thread, una volta creato, si usa il metodo start(), il quale provoca l'esecuzione del metodo run(); terminato tale metodo, il thread cessa la sua attività, e le risorse impegnate per quel thread vengono rilasciate. A questo punto, se all'oggetto Thread è collegata a una variabile in uno scope ancora attivo, l'oggetto non viene eliminato dal garbage collector a meno che la variabile non venga impostata a null.

Tuttavia tale oggetto non è più utilizzabile, ed una successiva chiamata del metodo start(), pur non generando alcuna eccezione, non avrà alcun effetto; la regola di base dice infatti che l'oggetto Thread è concepito per essere usato una volta soltanto.

È quindi importante tener presente che, se si hanno uno o più riferimenti, si dovrebbe aver cura di impostare tali variabili a null per liberare la memoria impegnata dall'oggetto. Se invece creiamo il Thread senza alcun riferimento a una variabile, ad esempio

```
new SimpleThread("My Thread").start();
```

la virtual machine si fa carico di mantenere l'oggetto in memoria per tutta la durata di esecuzione del thread, e di renderlo disponibile per la garbage collection una volta terminata l'esecuzione.

La classe Thread contiene anche un metodo stop(), che permette di terminare l'esecuzione del thread dall'esterno. Ma questo metodo è deprecato in Java 2 per motivi di sicurezza. Infatti in questo caso l'esecuzione si interrompe senza dare la possibilità al thread di eseguire un cleanup: il thread in questo caso non ha alcun controllo sulle modalità di terminazione. Per questo motivo l'uso di stop() è da evitare comunque, indipendentemente dalla versione di Java che si usa.

L'interfaccia Runnable

L'altra possibilità che permette di creare ed eseguire thread si basa sull'utilizzo della interfaccia Runnable a cui si accennava in precedenza. Ecco un esempio, equivalente al precedente, ma che usa una classe Runnable anziché una sottoclasse di Thread:

```
public class SimpleRunnable implements Runnable {
 String message;

 public SimpleRunnable(String s) {
    message = s;
 }

 public void run() {
    for (int i = 0; i < 100; i++)
       System.out.println(message);
 }

 public static void main(String[] args) {
    SimpleRunnable sr1, sr2;
    sr1 = new SimpleRunnable(„Buongiorno");
    sr2 = new SimpleRunnable("Buonasera");
```

```
        Thread t1 = new Thread(sr1);
        Thread t2 = new Thread(sr2);
        t1.start();
        t2.start();
    }
}
```

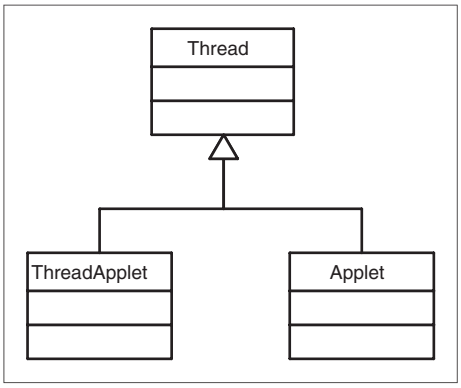
L'interfaccia Runnable contiene un solo metodo, il metodo run(), identico a quello già visto per la classe Thread. Questa non è una semplice coincidenza, dal momento che la classe Thread, in realtà, implementa l'interfaccia Runnable.

Per la precisione implementando l'interfaccia Runnable e il metodo run(), una classe *non derivata da* Thread può funzionare come un Thread, e per far questo però, deve essere "agganciata" a un oggetto Thread (un'istanza della classe Thread o di una sua sottoclasse) passando un reference dell'oggetto Runnable al costruttore del Thread.

Dall'esempio fatto, però, l'interfaccia Runnable non risulta particolarmente utile, anzi sembra complicare inutilmente le cose: che bisogno c'è di rendere un altro oggetto Runnable se si può usare direttamente una sottoclasse di Thread?

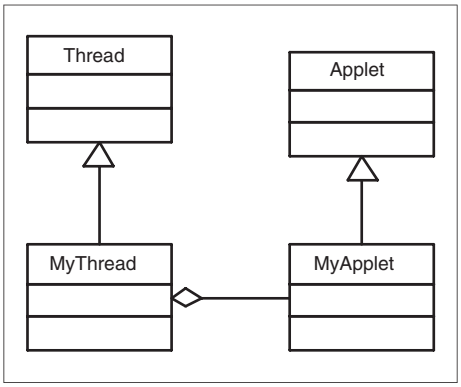
Per rispondere a tale quesito, si pensi ad una situazione in cui si voglia far sì che una certa classe implementi contemporaneamente la funzione di thread, ma che specializzi anche un'altra classe base (fig. 10.1).

Figura 10.1 – *Per creare una classe che sia contemporaneamente un Thread ma anche qualcos'altro, si può optare per una ereditarietà multipla. Tale soluzione non è permessa in Java.*



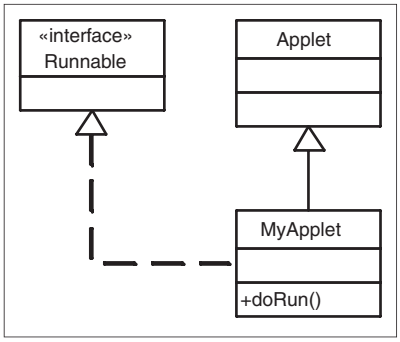
Ora dato che in Java non è permessa l'ereditarietà multipla, tipicamente una soluzione a cui si ricorre è quella di utilizzare uno schema progettuale differente, basato spesso sul pattern Delegation o sullo Strategy (come mostrato in fig. 10.2).

Figura 10.2 – *In alternativa, si può ereditare da un solo padre ed inglobare un oggetto che svolga la funzione che manca. Questo pattern, molto utilizzato, non risulta essere particolarmente indicato nel caso dei thread.*



Questa architettura non si adatta molto bene al caso dei thread, o comunque risulta essere troppo complessa, visto che l'interfaccia Runnable ne offre un'altra molto più semplice. Derivando dalla classe base e implementando l'interfaccia Runnable infatti si può sia personalizzare la classe base, sia aggiungere la funzione di thread (si veda la fig. 10.3).

Figura 10.3 – *Grazie all'utilizzo dell'interfaccia Runnable, si possono derivare classi al fine di specializzarne il comportamento ed aggiungere funzionalità di thread.*



Ecco con un esempio come si può implementare tale soluzione

```
class RunnableApplet extends Applet implements Runnable {
 String message;
```

```
 RunnableApplet(String s) {
    message = s;
 }

 public void init() {
    Thread t = new Thread(this);
    t.start();
 }

 public void run() {
    for (int i = 0; i < 100; i++)
       System.out.println(message);
}
```

Anche se l'esempio è forse poco significativo, riesce a far capire come l'oggetto può eseguire nel metodo run() dei compiti suoi propri, usando i suoi dati e i suoi metodi, e anche quelli ereditati dalla classe base, mentre l'oggetto Thread incapsulato viene usato solo per eseguire tutto questo in un thread separato.

Utilizzando una variabile di classe per il thread possiamo incrementare il controllo sul thread: aggiungendo un metodo start() è possibile far partire il thread dall'esterno, al momento voluto anziché automaticamente in fase di inizializzazione dell'Applet:

```
class RunnableApplet extends Applet implements Runnable {
 String message;
 Thread thread;

 RunnableApplet(String s) {
    message = s;
 }

 public void init() {
    thread = new Thread(this);
 }

 public void start() {
    t.start();
 }

 public void run() {
    for (int i = 0; i < 100; i++)



System.out.println(message);
}
```

Questo è il caso più tipico di utilizzo dell'interfaccia Runnable: un oggetto Thread viene inglobato in un oggetto già derivato da un'altra classe e utilizzato come "motore" per l'esecuzione di un certo codice in un thread separato.

Quindi si può dire semplicisticamente che l'uso dell'interfaccia Runnable al posto della derivazione da Thread si rende necessario quando la classe che si vuole rendere Runnable è già una classe derivata.

Negli altri casi si può scegliere il metodo che appare più conveniente.

Identificazione del thread

Ogni thread che viene creato assume un'identità autonoma all'interno del sistema: per facilitarne la successiva identificazione è possibile assegnare un nome al thread, passandolo al costruttore. Ad esempio:

```
SimpleRunnable sr = new SimpleRunnable("Buongiorno");
Thread t = new Thread (sr, "Thread che saluta");
```

con una successiva chiamata del metodo getName() è possibile conoscere il nome del thread.

In ogni caso se non è stato assegnato al momento della creazione, il runtime Java provvede ad assegnare a ciascun thread un nome simbolico che però non è molto esplicativo all'occhio dell'utente.

L'uso di nomi significativi è particolarmente utile in fase di debugging, rendendo molto più facile individuare e selezionare il thread che si vuol porre sotto osservazione.

Maggior controllo sui thread

Oltre alla gestione ordinaria dei thread, Java fornisce una serie di strumenti che permettono di gestire l'esecuzione di un thread fin nei minimi dettagli. Se da un lato questo permette una maggiore capacità di controllo del thread stesso, dall'altro comporta un miglior controllo sulle risorse che sono utilizzate durante l'esecuzione.

Di conseguenza migliora il livello di portabilità della applicazione, dato che si può sopperire a certe carenze del sistema operativo.

Una fine tranquilla: uscire da run()

Negli esempi precedenti sono stati presi in considerazione casi con thread che eseguono un certo compito per un lasso limitato di tempo (stampare un certo messaggio 100 volte). Finito il compito, il thread termina e scompare dalla circolazione.

Spesso accade però che un thread possa vivere per tutta la durata dell'applicazione e svolgere il suo compito indefinitamente, senza mai terminare; oppure continui finché il suo funzionamento non venga fatto cessare volutamente.

Un esempio tipico potrebbe essere quello che segue: un "thread-orologio" mostra in questo caso l'ora corrente aggiornandola periodicamente:

```
public class Clock extends Applet implements Runnable {
 private Thread clockThread = null;

 public void start() {
    if (clockThread == null) {
       clockThread = new Thread(this, “Clock”);
       clockThread.start();
    }
 }

 public void run() {
    while (clockThread != null) {
       repaint();
    try {
       // rimane in attesa per un secondo
       Thread.sleep(1000);
    }
    catch (InterruptedException e) {}
    }
 }

 public void paint(Graphics g) {
    // prende data e ora corrente
    Calendar systemTime = Calendar.getInstance();
    // formatta e visualizza l’ora
    DateFormat formatter = new SimpleDateFormat(“HH:mm:ss”);
    g.drawString(formatter.format(systemTime.getTime()), 5, 10);
 }

 public void stop() {
    clockThread = null;
 }
}
```

Al momento della creazione e inizializzazione della Applet (alla visualizzazione della pagina HTML nel browser) viene creato e fatto partire un thread.

Quando la pagina che contiene l’Applet viene lasciata, viene eseguito il metodo stop(), che mette la variabile clockThread a null. Ciò ha un doppio effetto: determina la terminazione del thread (quando la variabile ha il valore null il ciclo while del metodo run ha termine), e rende disponibile la memoria dell’oggetto Thread per la garbage collection.

Bisogno di riposo: il metodo sleep()

Riprendendo in esame la classe Clock, si può notare che nel metodo run() viene chiamato il metodo repaint(), che a sua volta determina l’esecuzione del metodo paint(), il quale infine visualizza l’ora di sistema.

Se non si utilizzassero ulteriori accorgimenti si avrebbe un grande spreco di risorse e un funzionamento tutt'altro che ottimale: l'ora infatti sarebbe continuamente aggiornata, senza alcuna utilità dal momento che vengono visualizzati solo i secondi, causando per di più un rallentamento del sistema e uno sfarfallio dell'immagine (effetto flickering).

Per evitare tutto questo viene in aiuto il metodo sleep(), che permette di sospendere l'esecuzione di un thread (facendolo passare allo stato blocked) per un periodo di tempo prefissato, specificato in millisecondi. Nel caso in questione, una sospensione della durata di un secondo è esattamente l'intervallo sufficiente per l'aggiornamento dell'orologio.

Una chiamata del metodo sleep() provoca una messa in attesa del thread corrente e l'esecuzione del primo thread in attesa, con conseguente cambio di contesto, non previsto dalla tabella dello scheduler. Questa operazione da una parte comporta un certo costo computazionale (che va tenuto presente), ma dall'altra libera una risorsa che talvolta, come nel nostro esempio, rimarrebbe inutilmente occupata.

Si tenga presente che la sospensione del thread per mezzo di una sleep può essere pericolosa nel caso in cui si implementi una qualche gestione sincronizzata delle variabili (vedi oltre), dato che non rilascia gli eventuali lock acquisti dal thread.

Il metodo sleep() è un metodo statico della classe Thread, e ha come effetto di sospendere l'esecuzione del thread corrente. Di conseguenza è possibile chiamarlo da qualunque classe, anche se non viene usato alcun oggetto di tipo Thread.

Gioco di squadra: il metodo yield()

Si è visto precedentemente, parlando dei processi, che esiste una forma di multitasking chiamato *cooperativo*, in cui ogni processo cede volontariamente il controllo del processore, dato che il sistema non gestisce lo scheduling dei processi. Si è anche detto che le specifiche della virtual machine non prevedono il time-slicing per cui, in presenza di thread di uguale priorità non è garantito che un thread che non rilasci le risorse di sua iniziativa non resti in esecuzione indefinitamente, impedendo di fatto agli altri thread di funzionare.

Per questi motivi, normalmente è buona norma non definire blocchi di istruzioni che possono richiedere molto tempo per essere eseguite ma, in alternativa, spezzare tali blocchi in entità più piccole. Lo scopo è quello di facilitare il compito dell'algoritmo di "schedulazione" in modo da evitare che un solo thread monopolizzi il processore per periodi troppo lunghi.

Anche nel caso in cui il sistema si faccia carico di partizionare il tempo di esecuzione, spesso lo scheduler non è in grado di stabilire in maniera automatica dove e quando risulti più opportuno interrompere un thread.

Il metodo yield() permette di gestire in maniera ottimale queste situazioni: esso consente infatti di cedere l'uso del processore a un altro thread in attesa con il grosso vantaggio che, nel caso in cui nessuno sia in attesa di essere servito, permette il proseguimento delle operazioni del thread invocante senza un inutile e costoso cambio di contesto.

L'invocazione di yield() non provoca un cambio di contesto (il thread rimane runnable), ma piuttosto viene spostato alla fine della coda dei thread della sua stessa priorità.

Ciò significa che questo metodo ha effetto solo nei confronti di altri thread di uguale priorità,

dato che i thread a priorità inferiore non prendono il posto del thread corrente anche se questo usa il metodo yield().

Utilizzando yield è il programmatore che stabilisce come e dove è opportuno cedere il processore, indipendentemente da quello che è poi il corso storico dei vari thread.

È bene eseguire una chiamata a tale funzione in quei casi in cui si ritiene che il thread possa impegnare troppo a lungo il processore, in modo da facilitare la cooperazione fra thread, permettendo una migliore gestione delle risorse condivise.

Il metodo yield() è un metodo statico della classe Thread, e ha effetto sul thread corrente. È possibile quindi chiamarlo da qualunque classe senza riferimento a un oggetto di tipo Thread.

La legge non è uguale per tutti: la priorità

Si è visto che la virtual machine adotta uno scheduling di tipo preemptive, basato sulla priorità: ogni volta quindi che un thread di priorità maggiore del thread in esecuzione diventa runnable, si ha un cambio di contesto; per questo in linea di massima il thread corrente è sempre un thread a priorità più alta.

Si è anche detto che la virtual machine non prevede necessariamente il time-slicing ma, se questo è presente, i thread a maggiore priorità dovrebbero occupare la CPU per un tempo maggiore rispetto a quelli a minore priorità. L'esempio che segue mostra questi aspetti dei thread, illustrando l'uso dei metodi setPriority() e getPriority(); la classe CounterThread rappresenta il thread di base, utilizzato in seguito dalla ThreadPriority.

```
private class CounterThread extends Thread {
 boolean terminated = false;
 int count = 0;

 public void run() {
    while (!terminated) {
       count++;
       for (int i = 0; i < 1000; i++) {
          ;


    }
 }

 public void terminate() {
    terminated = true;
 }

 public int getCount() {
    return count;
 }
}
```

La classe che implementa Runnable, oltre ad utilizzare il thread precedente, imposta anche le priorità.

```
public class ThreadPriority implements Runnable {
 CounterThread thread1 = new CounterThread();
 CounterThread thread2 = new CounterThread();
 Thread thisThread = new Thread(this);
 int duration;

 public ThreadPriority(int priority1, int priority2,
 int duration) {
    this.duration = duration;
    thisThread.setPriority(Thread.MAX_PRIORITY);
    thread1.setPriority(priority1);
    thread2.setPriority(priority2);
    thread1.start();
    thread2.start();
    thisThread.start();
 }


 public void run() {
    try {
       for (int i = 0; i < duration; i++){
          System.out.println("Thread1: priority: "
+ thread1.getPriority()
 + " count: " + thread1.count);
          System.out.println("Thread2: priority: "
+ thread2.getPriority()
 + " count: " + thread2.count);
          thisThread.sleep(1000);
       }
    }
    catch (InterruptedException e){}

    thread1.terminate();
    thread2.terminate();

 }

 public static void main(String[] args) {
    new ThreadPriority(Integer.parseInt(args[0]),
  Integer.parseInt(args[1]),
  Integer.parseInt(args[2]));
 }
}
```

La classe ThreadPriority crea due oggetti CounterThread e li manda in esecuzione; successivamente manda in esecuzione il suo thread collegato (si tratta di una classe che implementa Runnable), il quale stampa i valori delle priorità e del counter dei thread ogni secondo (scheduler permettendo), e alla fine termina i due thread.

I valori di priorità e la durata in secondi sono dati come argomenti del main sulla linea di comando. I valori di priorità devono essere numeri interi da 1 a 10.

Si può notare che si è assegnata una priorità massima a thisThread, che deve poter interrompere gli altri due thread per eseguire la stampa e le chiamate terminate(): per fare questo si è usata la costante Thread.MAX_PRIORITY, che ha un valore uguale a 10.

La classe CounterThread aggiorna un contatore dopo aver eseguito un ciclo vuoto di 1000 iterazioni (ovviamente consumando una quantità abnorme di tempo della CPU, ma ai fini dell'esempio sorvoliamo su quest'aspetto).

Mandando in esecuzione il programma si può notare che effettivamente dopo un certo tempo il programma termina e vengono stampate le informazioni, il che significa che i due CounterThread sono stati interrotti dall'altro thread a priorità massima.

Se il sistema operativo e la VM supportano il time-slicing, il numero raggiunto dal contatore è approssimativamente proporzionale alla priorità del thread. Bisogna tener presente che possono esserci variazioni anche notevoli dato che i thread possono essere gestiti secondo algoritmi abbastanza complessi e variabili da implementazione a implementazione.

Tuttavia si nota che comunque, aumentando la priorità, aumenta il valore del contatore, e viceversa.

L'uso della gestione diretta delle priorità risulta molto utile in particolare nei casi in cui si ha un thread che resta nello stato blocked per la maggior parte del tempo. Assegnando a questo thread una priorità elevata si evita che rimanga escluso dall'uso della CPU in sistemi che non utilizzano il time-slicing. Questo è particolarmente importante per operazioni temporizzate che devono avere una certa precisione.

In casi in cui un certo thread compie delle operazione che fanno un uso intenso della CPU, e di lunga durata, abbassando la priorità del thread si disturba il meno possibile l'esecuzione degli altri thread. Ciò, ovviamente, sempre a patto che sia possibile mettere in secondo piano tale task.

E l'ultimo chiuda la porta: il metodo join()

Il metodo join resta semplicemente in attesa finché il thread per il quale è stato chiamato non termina la sua esecuzione. Con questo metodo è quindi possibile eseguire una determinata operazione nel momento in cui un thread termina la sua esecuzione. Risulta pertanto utile in tutti i casi in cui un thread compie delle operazioni che utilizzano dei risultati dell'elaborazione di un altro thread.

Di seguito è riportato un breve esempio nel quale è mostrato come utilizzare tale metodo. In esso si è utilizzata una versione modificata della classe CounterThread, in cui è possibile specificare, come parametro del costruttore, il massimo valore raggiungibile dal counter. In tal modo possiamo limitare la durata di esecuzione del thread senza dover ricorrere al metodo terminate().

```
private class CounterThread extends Thread {
 boolean terminated = false;
 int count = 0;
 int maxCount = Integer.MAX_VALUE;

 public CounterThread() {
 }

 public CounterThread(int maxCount) {
    this.maxCount = maxCount;
 }

 public void run() {
    while (!terminated) {
       count++;
       for (int i = 0; i < 1000; i++) {
          // fai qualcosa
    }
 }

 public void terminate() {
    terminated = true;
 }

 public int getCount() {
    return count;
 }
}
```

La classe Chronometer misura il tempo di esecuzione, in minuti, secondi e millisecondi, di un thread che viene dato come argomento al metodo run(). Il metodo join() consente di determinare l'istante in cui termina l'esecuzione del thread (ovviamente con una certa approssimazione), e quindi di misurare il tempo trascorso dall'inizio dell'esecuzione.

```
public class Chronometer{
 Calendar startTime;
 Calendar endTime;

 public void run(Thread thread) {
    // registra l'ora di sistema all'inizio dell'esecuzione
    startTime = Calendar.getInstance();
    // manda in esecuzione il thread
    thread.start();
    try {
       // attende la fine dell'esecuzione
       thread.join();
    }
```

```
    catch (InterruptedException e) {}
    // registra l'ora di sistema alla fine dell'esecuzione
    endTime = Calendar.getInstance();
  }

  // calcola il tempo trascorso e restituisce
  // una stringa descrittiva
  public String getElapsedTime() {
    int minutes = endTime.get(Calendar.MINUTE)
   - startTime.get(Calendar.MINUTE);
    int seconds = endTime.get(Calendar.SECOND)
   - startTime.get(Calendar.SECOND);
    if (seconds < 0) {
      minutes--;
      seconds += 60;
    }
    int milliseconds = endTime.get(Calendar.MILLISECOND)
  - startTime.get(Calendar.MILLISECOND);
    if (milliseconds < 0) {
      seconds--;
      milliseconds += 1000;
    }
    return Integer.toString(minutes) + " minuti, " + seconds + " secondi, "  + milliseconds + " millisecondi";
  }

  public static void main(String[] args) {
    Chronometer chron = new Chronometer();
    // manda in esecuzione il thread per mezzo di Chronometer
    // dando come parametro al costruttore il numero massimo
    // raggiungibile dal counter, ricevuto a sua volta


 // come parametro di main, dalla linea di comando
    chron.run(new CounterThread(Integer.parseInt(args[0])));

    // stampa il tempo trascorso
    System.out.println(chron.getElapsedTime());
  }
 }
```

I metodi sleep() e join() sono metodi che hanno in comune la caratteristica di mettere un thread in stato di attesa (blocked). Ma mentre con sleep() l'attesa ha una durata prefissata, con join() l'attesa potrebbe protrarsi indefinitamente, o non avere addirittura termine. Alcune volte il protrarsi dell'attesa oltre un certo limite potrebbe indicare un malfunzionamento o comunque una condizione da gestire in maniera diversa che stando semplicemente ad aspettare.

In questi casi si può usare il metodo join(int millisecondi)che permette di assegnare un limite

massimo di attesa, dopo in quale il metodo ritornerà comunque, consentendo al metodo chiamante di riprendere l'esecuzione.

Sia sleep() che join() mettono in attesa il thread corrente, ma il metodo join() non è un metodo statico: viene chiamato per un oggetto specifico, che è quello di cui si attende la terminazione.

Interruzione di un thread

Un'altra caratteristica che accomuna thread e processi, è quella di essere soggetti a interruzioni. Come si è visto, l'interruzione è legata a un evento particolare, in qualche modo eccezionale, che determina cambiamenti tali nel contesto dell'esecuzione da richiedere (o poter richiedere) una gestione particolare dell'evento, ossia l'esecuzione di un codice specifico che fa fronte all'evento occorso.

In un thread l'interruzione ha luogo quando da un altro thread viene chiamato il metodo interrupt() per quel thread; formalmente è vero che un thread potrebbe interrompere se stesso, ma la cosa avrebbe poco senso.

L'aspetto più rilevante di questo metodo è che è in grado di interrompere uno stato di attesa causato da una chiamata a sleep() o join() (il discorso vale anche per il metodo wait() di cui si parlerà in seguito).

Se si ripensa per un momento agli esempi precedenti in cui vengono usati questi metodi, si nota che le chiamate sono all'interno di un blocco try e che nel catch viene intercettata un'eccezione del tipo InterruptedException anche se in questi casi l'eccezione non viene gestita.

Questo è appunto l'effetto di una chiamata al metodo interrupt(): se il thread interrotto è in stato di attesa, viene generata un'eccezione del tipo InterruptedException e lo stato di attesa viene interrotto.

L'oggetto Thread ha così l'opportunità di gestire l'interruzione, eseguendo del codice all'interno del blocco catch. Se il blocco catch è vuoto, l'effetto dell'interruzione sarà semplicemente quello di far riprendere l'esecuzione (non appena il thread, passato nuovamente allo stato runnable, sarà mandato in esecuzione dallo scheduler) dall'istruzione successiva alla chiamata sleep() o join().

Cosa accade se invece viene interrotto un thread che non è in attesa? In questo caso viene modificata una variabile di stato del thread facendo sì che il metodo isInterrupted() restituisca true. Questo permette al thread di gestire ugualmente l'interruzione controllando (tipicamente alla fine o comunque all'interno di un ciclo) il valore restituito da questo metodo:

```
public void run() {
 while (true) {
    doMyJob();
    if (isInterrupted())
      handleInterrupt();
 }
}
```

Purtroppo il flag di interruzione non viene impostato se l'interruzione ha luogo durante uno stato di attesa, per cui un codice del genere non funzionerebbe correttamente:

```
public void run() {
 while (true) {
    doMyJob();

    try {
       sleep(100);
    } catch (InterruptedException e) {}

    if (isInterrupted())
       handleInterrupt();
 }
}
```

Infatti, se l'interruzione ha luogo durante l'esecuzione di sleep(), viene generata una eccezione, ma il metodo isInterrupted() restituisce false.

Se si vuole gestire l'interruzione indipendentemente dal momento in cui si verifica, bisogna duplicare la chiamata a handleInterrupt():

```
public void run() {
 while (true) {
    doMyJob();

    try {
       sleep(100);
    } catch (InterruptedException e) {
       handleInterrupt()
    }


    if (isInterrupted())
       handleInterrupt();
 }
}
```

Il metodo interrupt() generalmente non interrompe un blocco dovuto ad attesa di I/O. In questi casi si deve agire direttamente sugli stream per interrompere lo stato di attesa. Il metodo interrupt è stato introdotto con Java 1.1 e non funziona con Java 1.0. Inoltre spesso non è supportato dalla VM dei browser, anche di quelli che dovrebbero supportare Java 1.1. Quindi per ora è opportuno evitarne l'uso nelle applet, a meno che non si faccia uso del Java plug-in.

Metodi deprecati

Il metodo stop(), che termina l'esecuzione di un thread, oltre a essere stato deprecato in Java 2, è sconsigliato: infatti il suo uso rischia di produrre malfunzionamenti causando una

interruzione "al buio" (cioè senza che il thread interrotto abbia il controllo delle modalità di terminazione. Per motivi analoghi sono deprecati i metodi `suspend()`, che mette il thread nello stato blocked, e `resume()` che lo sblocca, riportandolo allo stato runnable.

La sincronizzazione dei thread

Parlando dei processi, nell'introduzione, si è detto che questi hanno spazi di memoria separati e che possono condividere e scambiare dati tra loro solo con mezzi particolari appositamente concepiti per questo scopo. Si è inoltre detto che i thread che appartengono al medesimo processo condividono automaticamente lo spazio di memoria.

Come si è visto, una classe `Thread` o una basata sul `Runnable` funzionano come normali classi, e come tali hanno accesso a tutti gli oggetti che rientrano nel loro scope.

La differenza fondamentale è che, mentre le normali classi funzionano una alla volta, ossia eseguono il loro codice in momenti differenti, i thread vengono eseguiti in parallelo; questo significa che esiste la possibilità che thread diversi accedano contemporaneamente agli stessi dati. Anche se per "contemporaneamente" si intende sempre qualcosa basato su un parallelismo simulato, vi sono casi in cui questa "simultaneità" di accesso, per quanto relativa, può causare effettivamente dei problemi.

Sorge così l'esigenza di implementare una qualche tecnica di sincronizzazione dei vari thread. Prima di spiegare nei dettagli gli aspetti legati alla sincronizzazione, si ponga attenzione a i modi in cui diversi thread condividono oggetti e dati.

Condivisione di dati fra thread

Il caso più comune è quello di oggetti creati esternamente che vengono passati come parametri a un oggetto `Thread` o `Runnable`. Ad esempio:

```
public class PointXY {

 int x;
 int y;

 public PointXY(int x, int y) {
    this.x = x;
    this.y = y;
 }
}

public class PointThread extends Thread {

 PointXY point;

 public Run1(PointXY p) {
    point = p;
 }
```

```
 public void run() {
    // esegue operazioni con la variabile point
 }
}

public class PointRunnable implements Runnable {

 PointXY point;

 public Run2(PointXY p) {
    point = p;
 }

 public void run() {
    // esegue operazioni con la variabile point
 }
}

...

// dall'esterno si istanziano e si lanciano
// i thread relativi Run1 e Run2
PointXY point = new PointXY(10, 10);
PointThread thread1 = new PointThread(point);
PointRunnable runnable = new PointRunnable(point);
Thread thread2 = new Thread(runnable);

// start ed utilizzazione...
```

In questo modo si è permessa una condivisione di una variabile tra due thread, di cui uno funziona come sottoclasse di Thread, l'altro è collegato a una Runnable. Tutti e due gli oggetti hanno accesso alla stessa istanza dell'oggetto p di tipo PointXY.

Si sarebbero potute anche creare due istanze della stessa classe:

```
// condivisione di uno stesso oggetto point
// tra due istanze di uno stessa classe Thread
PointThread thread1 = new PointThread(point);
PointThread thread2 = new PointThread(point);
```

oppure due Thread collegati allo stesso Runnable:

```
// condivisione di uno stesso oggetto Runnable
// da parte di due Thread e di conseguenza
// condivisione dello stesso oggetto point
PointRunnable runnable = new PointRunnable(point);
Thread thread1 = new Thread(runnable);
Thread thread2 = new Thread(runnable);
```

In questi due casi i thread non solo condividono l'oggetto point, ma eseguono anche lo stesso codice in maniera indipendente ed eventualmente con differenti modalità. Nell'ultimo caso si è in presenza di una sola istanza di un oggetto Runnable, a cui si passa l'oggetto point, che viene "agganciato" a due thread differenti, mentre nel primo caso si creano due istanze di una sottoclasse di Thread, a cui si passa lo stesso oggetto point.

Competizione fra thread

Dopo aver accennato ad una delle configurazioni tipiche di accesso concorrente ad aree di memoria, si può passare a considerare quali siano i potenziali problemi derivanti dalla condivisione dei dati e dall'accesso parallelo a questi dati.

Si consideri ad esempio l'interfaccia Value che funge da contenitore (wrapper) di un valore intero:

```
public interface Value {
 public abstract int get();
 public abstract void set(int i);
 public abstract void increment();
}
```

La classe IntValue implementa l'interfaccia di cui sopra fornendo una gestione del valore contenuto come intero

```
public class IntValue implements Value {
 int value = 0;

 public int get() {
    return value;
 }

 public void set(int i) {
    value = i;
 }

 public void increment() {
    value++;
 }
}
```

La classe StringValue invece fornisce una gestione del valore come stringa

```
public class StringValue implements Value {
 String value = "0";

 public int get() {
```

```
    return Integer.parseInt(value);
  }

  public void set(int i) {
    value = Integer.toString(i);
  }

  public void increment() {
    int i = get();
    i++;
    set(i);
  }
}
```

Infine il thread permette di incrementare il valore contenuto in un generico wrapper, che viene passato al costruttore come interfaccia generica

```
public class ValueIncrementer extends Thread {
  Value value;
  int increment;

  public ValueIncrementer(Value value, int increment) {
    this.value = value;
    this.increment = increment;

}

  public void run() {
    for (int i = 0; i < increment; i++)
      value.increment();
  }

  public static void main(String[] args) {
    // crea un IntValue
    IntValue intValue = new IntValue();
    // crea due IntIncrementer a cui passa lo stesso IntValue
    // e lo stesso valore di incremento pari a 100000
    ValueIncrementer intIncrementer1  = new ValueIncrementer(intValue, 100000);
    ValueIncrementer intIncrementer2  = new ValueIncrementer(intValue, 100000);
    // ripete i passi precedenti
    // questa volta con un oggetto StringValue
    StringValue stringValue = new StringValue();
    ValueIncrementer stringIncrementer1  = new ValueIncrementer(stringValue, 100000);
    ValueIncrementer stringIncrementer2  = new ValueIncrementer(stringValue, 100000);

    try {
```

```
        // fa partire insieme i due thread che
        // incrementano lo IntValue
        intIncrementer1.start();
        intIncrementer2.start();
        // attende che i due thread terminino l'esecuzione
        intIncrementer1.join();
        intIncrementer2.join();
        // stampa il valore
        System.out.println("int value: " + intValue.get());

        // ripete i passi precedenti
        // questa volta con lo StringValue
        stringIncrementer1.start();
        stringIncrementer2.start();
        stringIncrementer1.join();
        stringIncrementer2.join();
        System.out.println("string value: " + stringValue.get());
      } catch (InterruptedException e) {}
    }
}
```

Tralasciando le considerazioni legate al modo in cui si effettua l'incremento, mandando in esecuzione l'esempio (impiegando una VM che utilizza il time-slicing), si può notare che mentre il valore per l'oggetto di tipo IntValue è quello che ci si aspetta, dovuto all'incremento di 100000 effettuato da due thread distinti, il valore dell'oggetto StringValue è inferiore, e varia da esecuzione a esecuzione.

Per capire cosa sia successo si esamini il codice del metodo increment() delle due classi IntValue e StringValue. Nella classe IntValue si ha

```
public void increment() {
 value++;
}
```

ovvero il metodo compie una semplice operazione di incremento di una variabile di tipo int. Invece, nella classe StringValue si trova

```
public void increment() {
 int i = get();
 i++;
 set(i);
}
```

Qui siamo in presenza di un algoritmo che, per quanto semplice, è formato da diverse istruzioni; a loro volta i metodi get() eset() chiamano metodi della classe Integer per convertire la stringa in int e viceversa, metodi che compiono operazioni di una certa complessità, ossia eseguono diverse istruzioni, ma possiamo anche prescindere da quest'ultima osservazione. Quello

che conta è che si tratta comunque di un'operazione complessa, divisa in più passi successivi. È questa complessità dell'operazione ciò che causa il problema. Infatti, se i due thread funzionano in parallelo in un sistema gestito con il time-slicing, è possibile che il passaggio da un thread all'altro avvenga durante l'esecuzione del metodo increment().

In questo caso, dato che i thread non fanno altro che eseguire increment() in un ciclo, e hanno la stessa priorità, le probabilità sono piuttosto alte. Di conseguenza, è possibile che il processore segua una sequenza di esecuzione come questa (per semplicità le chiamate get() e set() saranno ipotizzate istruzioni semplici):

1. Il primo thread esegue l'istruzione int i = get();
   Supponendo che il valore sia 100, questo valore viene assegnato alla variabile locale i

2. Il secondo thread esegue l'istruzione int i = get();
   Il valore è sempre 100, e viene assegnato all'altra variabile locale i (che è diversa per ciascun thread)

3. Il primo thread esegue l'istruzione i++;

4. Il valore della variabile locale diventa 101
   Il primo thread esegue l'istruzione set(i);
   La variabile di classe (condivisa fra i thread) diventa 101

5. Il secondo thread esegue l'istruzione i++;
   Il valore della variabile locale passa da 100 a 101

6. Il secondo thread esegue l'istruzione set(i);
   Alla variabile di classe viene assegnato il valore 101, che è lo stesso che già aveva, in seguito all'azione del primo thread.

Quindi, come risultato complessivo si ottiene un incremento di 1 e non di 2, come se uno dei due thread non avesse fatto nulla. Questa situazione di interferenza tra thread è quella che viene generalmente chiamata *race condition*.

Si noti come nel caso degli oggetti IntValue non si sia verificata nessuna alterazione: infatti il metodo increment() di questa classe compie una sola operazione non complessa, l'incremento della variabile interna, ossia un'istruzione che non può essere interrotta nel mezzo da un cambio di contesto. Le operazioni di questo tipo sono chiamate *atomiche*.

Alcune operazioni in apparenza atomiche, possono in realtà non esserlo: infatti se una variabile di tipo int è un rappresentata da un'area indivisibile di 32 bit, e quindi un'operazione di scrittura viene eseguita in una sola operazione non interrompibile, altrettanto non vale per un long, che occupa 64 bit di memoria, e in certi casi viene scritto o letto in due blocchi di 32 bit con due operazioni distinte. Se fra una operazione e la successiva si interrompe il thread, ci può

essere un'alterazione non voluta del valore della variabile. Si tenga presente che la frammentazione di una operazione di scrittura in sottoperazioni avviene a basso livello in maniera non visibile dal programma Java. Una race condition (condizione di competizione) è la situazione che si verifica quando due o più thread eseguono contemporaneamente operazioni di cui almeno una non atomica sugli stessi dati; l'ordine con il quale i vari passi di cui le operazioni sono composte vengono eseguiti dai diversi thread può portare ad alterazioni del risultato dell'intera operazione per uno o più thread.

Lock e sincronizzazione

Per risolvere una simile situazione è necessario che l'operazione complessa sia effettuata per intero da un thread alla volta, e che non venga interrotta da istruzioni relative alla stessa operazione eseguite da un altro thread. Per ottenere questo risultato generalmente si ricorre all'utilizzo dei cosiddetti lock.

Il lock può essere paragonato alla chiave di una toilette: alla toilette accede una sola persona alla volta e una volta entrata chiude la porta a chiave, dato che, anche in questo caso, sia pure per motivi differenti rispetto al caso dei thread, la condivisione della risorsa potrebbe portare a risultati indesiderati. Le altre persone che vogliono entrare, trovano la porta chiusa e devono pertanto attendere l'uscita dell'utente corrente della toilette.

Il lock può essere pensato come una semplice variabile booleana, visibile da tutti i thread. Ogni volta che un thread esegue un codice protetto da un lock, la variabile viene impostata a `true`, per indicare che il codice è già in esecuzione, e si dirà che il thread ha acquisito il lock su quel codice.

Il meccanismo di "schedulazione" si fa carico di garantire che, fin quando un thread è in possesso di un lock su un certo codice, nessun altro thread vi acceda. Eventuali thread che chiedono l'accesso al codice vengono così messi in attesa finché il thread corrente non ha rilasciato il lock.

In Java, però, tutto questo avviene "dietro le quinte", dal momento che il programmatore non usa direttamente dei lock, ma ricorre invece alla keyword `synchronized`. Il meccanismo è estremamente semplice.

Si riconsideri l'esempio di cui sopra facendo una piccola ma importante modifica al metodo `increment()`

```
public synchronized void increment() {
   int i = get();
   i++;
   set(i);
}
```

Se si prova adesso a eseguire l'applicazione, si potrà vedere che il risultato è corretto, per entrambe le classi `IntValue` e `StringValue`.

Quando un thread esegue un metodo synchronized, acquisisce il lock prima di eseguire le istruzioni, e lo rilascia al termine dell'esecuzione.

Ma acquisisce il lock su che cosa? La risposta è: acquisisce il lock sull'oggetto stesso. Più precisamente quando un oggetto è sottoposto a lock, nessuno dei suoi metodi sincronizzati è eseguibile se non dal thread che detiene il lock (cosa molto importante al fine di evitare deadlock).

Per capire meglio cosa questo significhi in pratica, si consideri una nuova classe ValueIncrementer2 identica alla classe ValueIncrementer, ma con una piccola modifica nel metodo run():

```
public void run() {
   for (int i = 0; i < increment; i++)
      value.set(value.get() + 1);
}
```

In questo caso l'incremento non è più ottenuto con una chiamata al metodo increment(), ma chiamando direttamente i metodi get() e set().

Se si prova ad eseguire contemporaneamente un oggetto ValueIncrementer e un oggetto ValueIncrementer2, nonostante la sincronizzazione del metodo increment() della classe StringValue, si otterrà una race condition con forti probabilità di funzionamento anomalo.

Il motivo di questa incomprensibile stranezza risiede nel fatto che increment() è l'unico metodo sincronizzato: ciò implica non solo che sia l'unico ad acquisire il lock, ma anche che sia l'unico a rispettarlo. In sostanza il lock non ha alcun effetto sui metodi non sincronizzati, in particolare sui metodi get() e set(), che quindi possono essere eseguiti in parallelo a increment() e causare i problemi che abbiamo visto.

Per evitare questi problemi, si devono definire come synchronized anche i metodi get() e set(). In tal modo, poiché il lock è sull'oggetto, sarà impossibile mandare contemporaneamente in esecuzione due metodi sincronizzati dello stesso oggetto; nel nostro caso i metodi increment(), get() e set() non potranno essere eseguiti in parallelo sullo stesso oggetto StringValue, ma dovranno attendere ognuno la fine dell'esecuzione dell'altro su un altro thread. Infine si tenga presente che una variabile non può essere direttamente sottoposta a lock, dato che si possono sincronizzare solo i metodi.

Quindi per permettere realmente la sincronizzazione sull'accesso concorrente a una variabile, oltre a definire sincronizzati tutti i metodi di gestione di tale variabile, si dovrà impedire l'accesso diretto per mezzo di una istruzione del tipo

```
oggetto.variabile = valore
```

Quindi tutte le variabili passibili di accesso condiviso devono essere protette e ad esse si deve accedere esclusivamente con metodi sincronizzati pubblici.

Un oggetto si dice thread-safe quando è protetto da malfunzionamenti causati da race condition e quindi è correttamente eseguibile anche contemporaneamente da thread differenti. Lo stesso termine può essere riferito anche a singoli metodi o a intere librerie.

Visibilità del lock

L'uso di synchronized fino ad ora è stato applicato a un intero metodo. Esiste anche la possibilità di circoscrivere l'ambito della sincronizzazione a un blocco di codice, ottenendo così un blocco sincronizzato. Ecco un'altra versione del metodo increment() di StringValue, che esemplifica questa modalità:

```
public void increment() {
   synchronized (this) {
      int i = get();
      i++;
      set(i);
   }
}
```

In questo caso le due versioni del metodo sono esattamente equivalenti, dato che il lock viene acquisito all'inizio del metodo e rilasciato alla fine.

Può capitare però che solo una porzione di codice all'interno di un metodo necessiti di sincronizzazione. In questi casi può essere opportuno usare un blocco sincronizzato piuttosto che un metodo sincronizzato, restringendo lo scope del lock.

Per visibilità di un lock (*scope* del lock) si intende il suo ambito di durata, corrispondente alla sequenza di istruzioni che viene eseguita tra il momento in cui il lock viene acquisito e quello in cui viene rilasciato.

Utilizzando il blocco sincronizzato si deve anche specificare l'oggetto di cui vogliamo acquisire il lock. Nell'esempio precedente l'oggetto è lo stesso di cui si sta eseguendo il metodo, e quindi è indicato con this, ma potrebbe essere anche un altro.

Questo significa che un lock su un oggetto può aver effetto anche su codice di altri oggetti, anche di classi differenti. Quindi, correggendo un'affermazione precedentemente fatta, dal contenuto ancora impreciso, si può dire che quando un oggetto è sottoposto a lock, nessuna area sincronizzata — intendendo sia blocchi che metodi — che richieda il lock per quel determinato oggetto è eseguibile se non dal thread che detiene il lock.

Tra brevissimo sarà preso in esame un esempio di uso di un blocco sincronizzato per un oggetto diverso da this. Ma quali sono i criteri in base ai quali scegliere lo scope appropriato? Bisogna naturalmente valutare caso per caso tenendo presente i diversi aspetti a favore e contro.

Da una parte, uno scope più esteso del necessario può causare inutili ritardi nell'esecuzione di altri thread, e in casi particolari può anche portare a una situazione di stallo, detta deadlock, di cui si dirà tra poco.

Dall'altra, acquisire e rilasciare un lock è un'operazione che consuma delle risorse e quindi, se si verifica troppo di frequente, rischia di influire negativamente sull'efficienza del programma. Inoltre, come vedremo, anche l'acquisizione di troppi lock può portare al verificarsi di deadlock.

Lo scope di un'area sincronizzata in Java non può estendersi al di là di un singolo metodo. Nel caso servano lock di scope più estesi (che vengano acquisiti in un metodo e rilasciati in un'altro, eventualmente di un'altro oggetto) occorre ricorrere a lock implementati *ad hoc* (ad esempio una classe Lock) la cui trattazione esula dagli scopi di questo capitolo.

Deadlock

Si supponga di scrivere una classe FileUtility che fornisca una serie di funzioni di utilità per il file system. Una delle funzioni è quella di eliminare da una directory tutti i files la cui data è precedente a una certa data fissata dall'utente, oppure esistenti da più di un certo numero di giorni. Un'altra funzione è di comprimere i files di una certa directory.

Si supponga di aver creato due classi:

la classe File, che tra l'altro contiene un metodo isOlder(Date d) che controlla se il file è antecedente a una certa data, e un metodo compress() che comprime il file;

la classe Directory, che contiene tra gli altri un metodo removeFile(File f), e dei metodi firstFile() e nextFile() utilizzabili per iterare sui files della directory, che sono mantenuti come una collezione di oggetti File all'interno dell'oggetto Directory.

La classe FileUtility, da parte sua, contiene removeOldFiles(Directory dir, Date date), un metodo che elimina i files "vecchi", e compressFiles(Directory dir), un metodo che comprime tutti i file di una directory.

Questa potrebbe essere una implementazione del metodo removeOldFiles:

```
public void removeOldFiles (Directory dir, Date date) {
 for (File file = dir.firstFile(); file != null; dir.nextFile()) {
synchronized (file) {
      if (file.isOlder(date)) {


      synchronized (dir) {
           dir.removeFile(file);
        }
      }
   }
 }
}
```

Questo è un tipico esempio di uso del blocco sincronizzato su un oggetto diverso da this: quello che serve è un lock sul file, per evitare che altri thread possano agire contemporaneamente sullo stesso file. Se il file risulta "vecchio" si utilizza il metodo removeFile() dell'oggetto Directory, ed anche in questo caso si deve ottenere il lock su tale oggetto, per evitare interventi contemporanei sulla stessa directory, che potrebbero interferire con l'operazione di cancellazione.

Questa potrebbe essere una possibile implementazione del metodo compressFiles():

```
public void compressFiles (Directory dir) {
 synchronized (dir) {
   for (File file = dir.firstFile(); file != null; dir.nextFile()) {
     synchronized (file) {
       file.compress();
     }
   }
 }
}
```

Anche in questa circostanza il thread deve acquisire i lock sull'oggetto Directory e sull'oggetto File per evitare interferenze potenzialmente dannose.

Si ipotizzi adesso che i due thread siano in esecuzione contemporaneamente e che si verifichi una sequenza di esecuzione come questa:

1. Il primo thread chiama il metodo compressFiles() per un certo oggetto Directory, acquisendone il lock;

2. Il secondo thread chiama il metodo removeOldFiles() per lo stesso oggetto Directory, verificando che il primo file è vecchio, e acquisisce il lock per il primo oggetto File;

3. Il secondo thread, per procedere alla rimozione del file, tenta di acquisire il lock sull'oggetto Directory, lo trova già occupato e si mette in attesa;

4. Il primo thread tenta di acquisire il lock per il primo oggetto File, lo trova occupato e si mette in attesa.

A questo punto i thread si trovano in una situazione di stallo, in cui ognuno aspetta l'altro, ma l'attesa non avrà mai termine. Si è verificato un deadlock.

Un deadlock è una situazione in cui due o più thread (o processi) si trovano in attesa l'uno dell'altro, in modo tale che gli eventi attesi non potranno mai verificarsi.

Per quanto riguarda la prevenzione dell'insorgenza di deadlock, non ci sono mezzi particolari messi a disposizione dal linguaggio, né regole generali e precise da seguire.

Si tratta di esaminare con attenzione le possibili interazioni fra thread e tenerne conto nell'implementazione delle classi. Ci sono naturalmente dei casi tipici, il cui esame va però al di là degli obiettivi di questo capitolo.

Class lock e sincronizzazione di metodi statici

La keyword synchronized può essere usata anche per metodi statici, ad esempio per sincronizzare l'accesso a variabili statiche della classe. In questo caso quello che viene acquisito è il lock della classe, anziché di un determinato oggetto di quella classe.

In realtà il lock si riferisce sempre a un oggetto, e precisamente all'oggetto Class che rappresenta quella classe. Quindi è possibile acquisire un lock della classe anche da un blocco sincronizzato, specificando l'oggetto Class:

```
public void someMethod() {
 synchronized (someObject.class) {
    doSomething();
 }
}
```

oppure:

```
public void someMethod() {
 synchronized (Class.forName("SomeClass")) {
    doSomething();
 }
}
```

Comunicazione fra thread

Dato che la programmazione per thread permette l'esecuzione contemporanea di più flussi di esecuzione autonomi fra loro, sorge abbastanza spontanea l'esigenza di mettere in comunicazione fra loro tali flussi in modo da realizzare qualche tipo di lavoro collaborativo. Il modo più semplice per ottenere la comunicazione fra thread è la condivisione diretta di dati, attraverso codice sincronizzato, come visto in precedenza. Ma ci sono situazioni in cui questo sistema non è sufficiente.

Condivisione di dati

Si consideri questo semplice esempio basato su le due classi Transmitter e Receiver, utilizzabili su thread differenti per lo scambio di dati:

```
public class Transmitter extends Thread {
 Vector data;

 public Transmitter(Vector v) {
    data = v;
 }

 public void transmit(Object obj) {
    synchronized (data) {
       data.add(obj);
    }
 }
```

```
  public void run() {
    int sleepTime = 50;
    transmit("Ora trasmetto 10 numeri");
    try {
      if (!isInterrupted()) {
        sleep(1000);
        for (int i = 1; i <= 10; i++) {
          transmit(new Integer(i * 3));
          if (isInterrupted())
            break;
          sleep(sleepTime * i);
        }
      }
    } catch (InterruptedException e) {}
    transmit("Fine della trasmissione");
  }
}
```

La classe Transmitter implementa un semplice meccanismo di condivisione dei dati attraverso un oggetto Vector, che viene passato come argomento del costruttore. Il metodo transmit() non fa altro che aggiungere un elemento al Vector, dopo aver acquisito un lock sul Vector stesso.

Questa operazione ha un reale effetto perché la classe Vector è una classe thread-safe, ossia è stata implementata usando dove necessario dei blocchi o dei metodi sincronizzati.

Il metodo run() trasmette un messaggio iniziale, attende un secondo, poi trasmette una sequenza di 10 numeri a intervalli di tempo crescenti, infine trasmette un messaggio finale.

In questo metodo viene anche esemplificata una gestione delle interruzioni: i messaggi iniziale e finale vengono comunque trasmessi; in caso di interruzione durante la trasmissione viene conclusa la trasmissione in corso, poi si esce dal ciclo; se l'interruzione arriva durante una chiamata a sleep(), questa causa un salto al blocco catch, vuoto, con un risultato equivalente.

```
public class Receiver extends Thread {
  Vector data;

  public Receiver(Vector v) {
    data = v;
  }

  public Object receive() {
    Object obj;
    synchronized (data) {
      if (data.size() == 0)
        obj = null;
      else {
        obj = data.elementAt(0);
```

```
            data.removeElementAt(0);
        }
    }
    return obj;
}

public void run() {
    Object obj;
    while (!isInterrupted()) {
        while ((obj = receive()) == null) {
            try {
                sleep(1000);
            } catch (InterruptedException e) {
                return;
            }
        }
        System.out.println(obj.toString());
    }
}
}
```

La classe Receiver riceve anch'essa un Vector come argomento del costruttore, e in tal modo ha la possibilità di condividere i dati con un Transmitter.

Il metodo receive() restituisce null se non trova dati; altrimenti restituisce il dato dopo averlo rimosso dal Vector.

Il metodo run() esegue un ciclo che può essere terminato solo da una chiamata a interrupt().

Anche in questo caso viene gestito sia la possibilità di interruzione in stato di attesa con una InterruptedException, sia quella di interruzione durante l'esecuzione.

Ad ogni ciclo si prova a ricevere un dato: se la ricezione ha luogo, stampa il dato sotto forma di stringa, altrimenti attende un secondo e riprende il ciclo.

```
public class ThreadCommunication{
 public static void main(String[] args) {
    Vector vector = new Vector();
    Transmitter transmitter = new Transmitter(vector);
    Receiver receiver = new Receiver(vector);
    transmitter.start();
    receiver.start();
    try {
        transmitter.join();
        Thread.sleep(2000);
    } catch (InterruptedException e) {}
    receiver.interrupt();
 }
}
```

La classe ThreadCommunication contiene soltanto un main() che mostra il funzionamento delle classi Transmitter e Receiver: in tale metodo viene creato un oggetto di tipo Vector e lo passa ai due oggetti Transmitter e Receiver, che poi provvede a far partire.

Quando il Transmitter ha terminato la sua esecuzione, attende 2 secondi per dare al Receiver il tempo di ricevere gli ultimi dati; successivamente termina il Receiver con una chiamata al metodo interrupt().

In questo caso il metodo interrupt() è usato per terminare il thread: questo è un sistema che può essere usato nei casi in cui non ci sia la necessità gestire le interruzioni diversamente, senza terminare il thread, ma eseguendo un determinato codice. La differenza, rispetto all'uso di un flag di stop, è che il thread termina immediatamente anche se si trova in stato di attesa.

Utilizzo dei metodi wait() e notify()

Per migliorare la sincronizzazione fra i thread, si può ricorrere all'utilizzo dei metodi wait() e notify().

Si tratta di metodi appartenenti alla classe Object e non alla classe Thread, per cui possono essere utilizzati per qualunque oggetto. Il funzionamento è semplice: se un thread esegue una chiamata al metodo wait() di un determinato oggetto, il thread rimarrà in stato di attesa fino a che un altro thread non chiamerà il metodo notify() di quello stesso oggetto.

Con poche variazioni al codice, si possono modificare le classi appena viste in modo che utilizzino wait() e notify().

Per prima cosa si può si può modificare il metodo transmit() della classe Transmitter in modo che sia effettuata una chiamata al metodo notify() per segnalare l'avvenuta trasmissione:

```
public void transmit(Object obj) {
 synchronized (data) {
    data.add(obj);
    data.notify();
 }
}
```

Nella classe Receiver invece il metodo run() diventa

```
public void run() {
 Object obj;
 while (!isInterrupted()) {
    synchronized (data) {
       while ((obj = receive()) == null) {
          try {
             data.wait();
          } catch (InterruptedException e) {
             return;
          }
```

```
            }
        }
        System.out.println(obj.toString());
    }
}
```

Come si può notare al posto della chiamata a sleep() si effettua una chiamata a wait(): in tal modo l'attesa si interromperà subito dopo la trasmissione, segnalata dalla chiamata notify(). Per il resto il funzionamento resta uguale.

Si può notare però un'altra differenza: all'interno del ciclo è stato inseritoun blocco sincronizzato, pur essendo il metodo receive() già sincronizzato.

Il motivo è che utilizzando direttamente la wait() è necessario prima ottenere il lock sull'oggetto, altrimenti si causa un'eccezione del tipo IllegalMonitorStateException, che darà il poco intuitivo messaggio current thread not owner, che sta a significare che il thread non detiene il lock sull'oggetto.

Più precisamente questo significa che, essendo la wait() un metodo della classe Object e non della Thread, è possibile invocare una wait() su tutti gli oggetti Java, non solo sui thread, anche se l'invocazione può avvenire solo se preventivamente si è acquisito il lock relativo. Discorso del tutto analogo per il metodo notify().

Questa è la tipica sequenza di eventi per la creazione di una sincronizzazione fra due thread; supponendo che il thread thread1 chiami wait() e il thread thread2 chiami notify():

1. il thread1 chiama wait() dopo aver acquisito il lock sull'oggetto; wait() per prima cosa rilascia il lock, poi mette il thread in attesa;

2. il thread2 a questo punto può acquisire il lock ed eseguire il blocco sincronizzato da cui chiama notify() dopodiché il lock sarà rilasciato…

3. …da thread1; wait(), ricevuta la notifica riacquisisce il lock ed esce; successivamente l'esecuzione del codice potrà continuare.

Tipicamente questo tipo di comunicazione si usa per aspettare/notificare il verificarsi di una certa condizione.

Nell'esempio appena visto la condizione è la presenza di dati ricevuti; in mancanza del meccanismo di sincronizzazione descritto sopra, e permettendo l'uso di wait() e notify() al di fuori di aree sincronizzate, si potrebbe verificare una sequenza di questo tipo:

1. il Receiver controlla se ci sono dati ricevuti; non ne trova;

2. il Transmitter trasmette un dato,

3. il Transmitter chiama notify(), ma la notifica non arriva a destinazione, dal momento che non c'è ancora nessun thread in attesa;

4. il Receiver si mette in attesa, ma ormai la notifica è andata persa.

È importante osservare che se un thread acquisisce il lock su un determinato oggetto, solo esso potrà eseguire l'operazione di rilascio e, finché non effettuerà tale operazione (uscendo dal blocco sincronizzato), il lock risulterà sempre occupato.

Questo fatto ha un'importante conseguenza: il thread messo in stato di attesa, che restituisce temporaneamente il lock, potrà essere riattivato solo se, dopo la chiamata a notify(), viene ad esso restituito il lock sull'oggetto che originariamente aveva acquisito. Non è detto quindi che un thread riprenda immediatamente la sua esecuzione dopo una chiamata a notify(), ma può trascorrere un periodo di tempo non precisato.

Ad esempio, se si scrive

```
synchronized (object){
 doSomething();
 object.notify();
 doSomethingElse();
}
```

fino a che non viene terminata l'esecuzione di doSomethingElse(), il thread che ha chiamato wait() non può riprendere l'esecuzione, anche se ne è stata richiesta la riattivazione con notify().

Il metodo wait() esiste anche nella versione wait(int millisecondi): in questo caso viene specificato un timeout scaduto il quale lo stato di attesa termina anche se non è stata ricevuta alcuna notifica.

Questo metodo può essere usato al posto di sleep() quando si vuole bloccare momentaneamente il thread rilasciando contemporaneamente il lock acquisito.

Il metodo notifyAll()

Se i thread in accesso concorrente sono più di uno, e tutti in attesa a causa di una wait(), allora una chiamata alla notify() avvertirà uno solo dei thread in attesa, senza la possibilità di sapere quale. In situazioni del genere il metodo notifyAll(), permette di eseguire la notifica nei confronti di tutti i thread in attesa.

Nel caso in cui si desideri che sia solo un particolare thread tra quelli in attesa a riprendere l'esecuzione, si deve implementare del codice *ad hoc* che gestisca la situazione, dato che il linguaggio Java non mette a disposizione nessuno costrutto particolare.

Deamon thread

I thread in Java possono essere di due tipi: *user thread* o *deamon thread*. Il termine *deamon* è stato usato inizialmente per designare un certo tipo di processi nei sistemi operativi multitasking (in particolare in ambiente Unix), ossia dei processi "invisibili" che girano "in background" e svolgono dietro le quinte dei servizi di carattere generale. In genere questi processi restano in

esecuzione per tutta la sessione del sistema. Il termine "demone" è stato usato probabilmente in analogia con "fantasma" a simboleggiare invisibilità e onnipresenza.

I deamon thread in Java sono qualcosa di molto simile ai processi deamon: sono infatti thread che spesso restano in esecuzione per tutta la durata di una sessione della virtual machine, ma soprattutto sono thread che si suppone che svolgano dei servizi per gli user thread, e che questa sia l'unica ragione della loro esistenza. In effetti l'unica differenza tra uno user thread e un deamon thread è che la virtual machine termina la sua esecuzione quando termina l'ultimo user thread, indipendentemente dal fatto che ci siano o meno in esecuzione dei deamon thread.

Spesso i deamon thread sono thread creati e mandati automaticamente in esecuzione dalla stessa virtual machine: un caso tipico è quello del garbage collector, che si occupa periodicamente di liberare la memoria allocata per oggetti non più in uso.

Ma un deamon thread può essere anche creato dall'utente, cioè dal programmatore: a tale scopo esiste il metodo setDeamon(boolean value) che permette di rendere deamon uno user thread o, viceversa, user un deamon thread.

Per default un thread, quando viene creato assume lo stato del thread "padre" da cui è stato creato. Con setDeamon() è possibile modificare questo stato, ma soltanto prima di mandare in esecuzione il thread con start().

Una chiamata durante l'esecuzione causerà un'eccezione. Per conoscere il deamon state di un thread si può usare il metodo isDeamon().

Se si creano dei thread di tipo deamon, occorre sempre tener presente che non devono svolgere delle operazioni che possano protrarsi oltre la durata di esecuzione degli user thread per cui svolgono i loro servizi. Questo rischierebbe di interrompere a metà queste operazioni, perché la Virtual Machine potrebbe terminare per mancanza di user thread in esecuzione.

I gruppi di thread

Ogni thread in Java appartiene a un gruppo; per default il gruppo di appartenenza è quello del thread padre. La virtual machine genera automaticamente dei thread groups, di cui almeno uno è destinato ai thread creati dalle applicazioni; questo sarà il gruppo di default per i thread creati da un'applicazione. Ogni applicazione può anche creare i suoi thread group, e assegnare i thread a questi gruppi.

I thread group sono organizzati secondo una struttura gerarchica ad albero: ciascun thread appartiene a un gruppo, il quale può appartenere a un altro gruppo; per questo ogni gruppo può contenere sia thread che gruppi di thread. La radice di quest'albero è rappresentata dal system thread group.

Per creare un thread group si deve creare un oggetto della classe ThreadGroup usando uno dei due costruttori:

```
ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name);
```

Come per i thread, se non viene specificato un thread group di appartenenza, il gruppo di appartenenza sarà quello del thread da cui è stato creato.

Per assegnare un thread a un gruppo si usa uno dei tre costruttori:

```
Thread(ThreadGroup group, String name),
Thread(ThreadGroup group, Runnable target),
Thread(ThreadGroup group, Runnable target, String name).
```

Una volta creato il thread come membro di un certo gruppo, non è possibile farlo passare ad un altro gruppo o toglierlo dal gruppo. Il thread sarà rimosso automaticamente dal gruppo una volta terminata la sua esecuzione.

Le funzionalità relative ai gruppi di thread si possono suddividere in quattro categorie: funzionalità di informazione, manipolazione collettiva dei thread appartenenti a un gruppo, funzioni relative alla priorità e funzioni legate alla sicurezza. Nei paragrafi successivi si analizzano tali funzionalità.

Informazioni sui thread e sui gruppi

Ci sono diversi metodi appartenenti alla classe ThreadGroup e alla classe Thread che forniscono informazioni sui thread e sui gruppi di thread.

Ci sono metodi che ci informano su quanti e quali sono i thread e i gruppi attualmente esistenti nella VM.

Il più importante è il metodo enumerate(), che fornisce la lista dei thread o dei thread group attivi, effettuando opzionalmente una ricorsione in tutti i sottogruppi.

Vi sono poi metodi che informano sui "rapporti di parentela" come getThreadGroup() della classe Thread o getParent() della ThreadGroup che permettono di conoscere il gruppo di appartenenza di un thread o di un gruppo.

Thread group e priorità

Con il metodo setMaxPriority(int priorità) è possibile assegnare a un gruppo una priorità massima. Se si tenta di assegnare a un thread del gruppo (o di sottogruppi) una priorità maggiore, questa viene automaticamente ridotta alla priorità massima del gruppo, senza che venga segnalato alcun errore.

La priorità massima può essere soltanto diminuita, e non aumentata. La priorità dei thread appartenenti al gruppo non viene in realtà modificata se si abbassa la priorità massima del gruppo, anche se è più alta di tale limite. La limitazione diviene attiva solo quando viene creato un nuovo thread o viene modificata la priorità di un thread con il metodo setPriority(). In questo caso non sarà possibile superare la priorità massima del gruppo.

Il valore della priorità di un gruppo può essere ottenuto con una chiamata al metodo getPriority().

Thread group e sicurezza

Le funzionalità più interessanti e più importanti legate ai thread group sono quelle relative alla sicurezza. Con i gruppi di thread è possibile consentire o interdire in maniera selettiva a interi gruppi di thread l'accesso ad altri thread e gruppi di thread.

Questa funzionalità è legata al funzionamento della classe java.lang.SecurityManager, la quale gestisce diverse funzioni legate alla sicurezza, tra cui alcune relative ai thread. In realtà queste funzioni sono riferite ai thread group, ma anche ai singoli thread.

Tuttavia i thread group assumono una particolare rilevanza perché consentono di discriminare l'accesso tra i thread sulla base dell'appartenenza ai gruppi, quindi accrescendo notevolmente le possibilità di organizzare i criteri di accesso secondo regole ben precise.

Il SecurityManager è quello che, ad esempio, si occupa di garantire che le Applet non possano accedere a determinate risorse del sistema. In questo caso si tratta di un SecurityManager fornito e gestito dal browser e dalla virtual machine del browser, a cui l'utente non ha accesso.

Ma per le applicazioni l'utente può invece creare e installare dei suoi SecurityManager. Prima di Java 2, le applicazioni non avevano nessun SecurityManager di default, quindi c'era solo la possibilità di usare dei SecurityManager creati dall'utente.

In Java 2 esiste anche un SecurityManager di default per le applicazioni, che può essere fatto partire con una opzione della VM, e configurato attraverso una serie di files di configurazione.

Tralasciando una descrizione complessiva del SecurityManager, le funzioni relative ai thread si basano in pratica su due soli metodi, o più precisamente su due versioni dello stesso metodo checkAccess(): questo infatti può prendere come parametro un oggetto Thread oppure un oggetto ThreadGroup.

A loro volta, le classi Thread e ThreadGroup contengono ciascuna un metodo checkAccess() che chiama i rispettivi metodi del SecurityManager.

Questo metodo viene chiamato da tutti i metodi della classe Thread e della classe ThreadGroup che determinano un qualsiasi cambiamento di stato nell'oggetto per cui vengono chiamati, per accertare che il thread corrente abbia il permesso di manipolare il thread in questione (che può essere lo stesso thread corrente o un altro thread).

Se le condizioni di accesso non sussistono, checkAccess() lancia una SecurityException che in genere viene semplicemente rilanciata dal metodo chiamante.

Le condizioni di accesso sono quindi stabilite dai metodi del SecurityManager, che può a tale scopo utilizzare tutte le informazioni che è in grado di conoscere sugli oggetti Thread o ThreadGroup di cui deve fare il check. Ad esempio può vietare l'accesso se il thread corrente e il thread in esame non appartengono allo stesso gruppo, o a seconda delle rispettive priorità, ecc.

I metodi che chiamano checkAccess() prima di compiere le loro operazioni sono:

1. nella classe Thread: Thread(), interrupt(), setPriority(), setDeamon(), setName(), più i metodi deprecati: stop(), suspend(), resume();

2. nella classe ThreadGroup: ThreadGroup(), interrupt(), setMaxPriority(), setDeamon(), destroy(), più i metodi deprecati: stop(), suspend(), resume().

Tutti questi metodi lanciano una SecurityException se il thread corrente non ha accesso al Thread o al ThreadGroup dell'oggetto this.

La classe ThreadLocal

Questa classe è stata introdotta con Java 2. Consente di avere una variabile locale al thread, cioè ciascun thread ha una sua istanza della variabile. Il valore della variabile è ottenuto tramite i metodi get() e set() dell'oggetto ThreadLocal. L'uso tipico di quest'oggetto è come variabile privata statica di una classe in cui si vuole mantenere uno stato o un identificatore per ogni thread.