Gestione dei thread

Introduzione…

Gli oggetti forniscono un modo per strutturare il programma in sezioni indipendenti. Spesso, però, è necessario dividere il programma, in parti (*subtask*) eseguibili indipendentemente e contemporaneamente.

Ogni subtask, chiamato *thread*, definisce un diverso percorso di esecuzione. Un thread è un singolo flusso di controllo sequenziale in un programma. Pertanto ne consegue che il multithreading può essere considerato una specializzazione del multitasking.

Infatti è importante distinguere il multitasking basato su processi da quello basato su thread.

…Introduzione.

Nel primo caso il sistema operativo è capace di eseguire uno o più processi (programmi) alla volta distribuendo periodicamente l'uso della CPU a ciascun processo. Ciò significa che il processo è l'unità minima di esecuzione. Infatti in tutti i sistemi operativi è possibile compilare un programma e allo stesso tempo utilizzare un editor.

Nel secondo abbiamo che l'unita di esecuzione più piccola è il thread quindi in un programma è possibile avere thread multipli in esecuzione.

Per esempio un editor di testi può formattare del testo e stamparne un altro contemporaneamente.

Java supporta il multithreading.

Uso della classe Thread...

Il modo più semplice per creare un thread è quello di derivare la classe ***Thread*** che ha tutti i meccanismi necessari per creare ed eseguire i thread.

Il metodo più importante della classe *Thread* è *run()* che rappresenta il punto di entrata per il thread.

Esempio

In questo esempio verranno creati dei thread a cui verranno assegnati un numero unico generato da una variabile *static*.

primo thread.doc

...Uso della classe Thread…

Oltre al metodo ***run()*** la classe *Thread* definisce altri metodi per la gestione dei *Thread*.

- *getName* ottiene il nome di un thread
- *getPriority* ottiene la priorità di un thread
- *isAlive* determina se un thread è ancora in esecuzione
- *join* attende che un thread termini
- *sleep* sospende un thread per un determinato periodo di tempo
- *start* avvia un thread chiamando il relativo metodo di esecuzione
- *suspend* sospende un thread

...Uso della classe Thread...

Quando un programma Java viene avviato, è già in esecuzione un thread che viene solitamente denominato *thread principale* (*main thread*) del programma.

Questo thread è importante per due motivi:

- è il thread da cui saranno generati gli altri *thread*;
- deve essere l'ultimo thread a terminare l'esecuzione dopodiché il programma termina.

...Uso della classe Thread…

E’ possibile controllare il thread principale, mediante un oggetto *Thread*, che è creato automaticamente all’avvio del programma.

A tale scopo viene fornito un metodo statico pubblico della classe *Thread* denominato *currentThread()*.

Questo metodo restituisce un riferimento al thread in cui viene chiamato permettendo di gestirlo come un qualsiasi altro thread.

...Uso della classe Thread.

In generale un thread può essere creato generando una istanza della classe *Thread*. E' possibile fare ciò in due modi:

- Implementando l'interfaccia *Runnable;*
- estendendo la classe *Thread*.

Implementazione dell'interfaccia *Runnable*...

Per implementare l'interfaccia Runnable, una classe deve implementare il metodo run().

Il metodo run() può chiamare altri metodi, utilizzare altre classi, dichiarare variabili come accade nel thread principale. La differenza sta nel fatto che run() stabilisce il punto di entrata per un nuovo thread di esecuzione all'interno del programma.

…Creazione di thread multipli.

Lo svantaggio sta nel fatto che il thread non potrebbe accedere ai membri privati o protetti della classe principale.

Esempio

multithread2.doc

Condivisione di risorse in Java…

Con il multithreading si può verificare la possibilità che due o più thread cercano di accedere ad una risorsa limitata *condivisa*. In tal caso è necessario prevedere dei metodi che siano in grado di garantire l'utilizzo esclusivo della risorsa da parte di un thread per volta in modo da evitare i *fenomeni di collisione*.

Il processo appena esposto viene chiamato ***sincronizzazione***.

Se si è già utilizzata la sincronizzazione con altri linguaggi (C, C++) si è certamente consapevoli della complessità che la caratterizza.

…Condivisione di risorse in Java.

Infatti, visto che la maggior parte dei linguaggi di programmazione non supporta la sincronizzazione, i programmi devono utilizzare le primitive del sistema operativo per sincronizzare i thread.

Java implementa la sincronizzazione attraverso gli elementi del linguaggio, liberandola in tal modo da gran parte della sua complessità.

Sincronizzazione di metodi

Tipicamente i membri dato di una classe sono privati alla classe stessa, per cui si può controllare l'uso della risorsa semplicemente sincronizzando l'invocazione dei metodi che vi accedono. In molte situazioni questo può bastare.

Java fornisce lo specificatore ***synchronized*** per quanto riguarda i metodi.

Solo **un** thread alla volta può chiamare un *metodo synchronized* per l'oggetto sul quale è invocato (sebbene un thread possa chiamare più metodi sincronizzati per l'oggetto).

Eliminazione delle collisioni…

Esempi di definizione dei metodi sincronizzati sono:

```
synchronized void f() {/* . . .*/}
synchronized void g() {/* . . .*/}
```

Ciascun oggetto contiene un singolo sistema di bloccaggio (chiamato anche ***monitor***) che è parte dell'oggetto stesso.

Quando viene chiamato un metodo sincronizzato su un oggetto, quell'oggetto è bloccato e non può essere chiamato alcun altro metodo sincronizzato dell'oggetto finché il primo non termina e rilascia il blocco.

…Eliminazione delle collisioni…

Per esempio supponiamo di avere i metodi sincronizzati *f()* e *g()*. Se viene invocato il metodo *f()* per un dato oggetto, il metodo *g()* non può essere chiamato per lo stesso oggetto finché *f()* non è completato.

Quindi se si vuole proteggere delle risorse da un accesso simultaneo di thread multipli si può obbligare l'accesso a tali risorse solo attraverso metodi sincronizzati.

…Eliminazione delle collisioni…

Esempio

uso di syncronized.doc

Nell'esempio il thread della classe SincType tramite il metodo ***run()*** incrementa due contatori. In più c'è un altro thread della classe ***Watcher*** che stampa il contenuti dei due contatori.

Senza la parola chiave ***syncronized*** potrebbe accadere che di tanto in tanto i due contatori diventano diversi.

Ma aggiungendo la parola syncronized a run() il Watcher non riesce mai a chiamare syntest fino a quando la risorsa non è sbloccata

…Eliminazione delle collisioni …

È possibile limitare la sincronizzazione alla sola sezione critica relativa ad un oggetto.

```
synchronized(syncObject) {
// codice accessibile da
// un thread alla volta
}
```

L'esempio precedente può essere modificato rimuovendo la parola chiave **synchronized** dal metodo **run( )** e ponendo invece le due linee critiche in un blocco **synchronized**.

Su quale oggetto si dovrebbe effettuare il blocco?

Quello già coinvolto in **synchTest(**), che è l'oggetto corrente **(this**).

…Eliminazione delle collisioni …

```
public  void run() {
while (true) {
synchronized(this){
a++;
try {
sleep(500);
} catch(InterruptedException e) {
System.err.println("Interrupted");
}
b++;
}
try {
sleep(500);
} catch(InterruptedException e) {
System.err.println("Interrupted");
}


}
}
```

Problemi di ***syncronized***

L'uso di ***syncronized*** è molto svantaggioso dal punto dell'efficienza, infatti si potrebbero verificare dei colli di bottiglia.

D'altra parte una condivisione non corretta delle risorse potrebbe portare a eventi disastrosi per il programma.

Si conclude che la sincronizzazione deve essere effettuata solo quando è necessaria.

Blocking…

Un thread può trovarsi in uno dei seguenti stati:

- *New*: l'oggetto thread è stato creato ma non è stato ancora avviato quindi non può essere in esecuzione. Il sistema di scheduling non sa ancora della sua esistenza.
- *Runnable*: un thread può essere in esecuzione quando il meccanismo di *time-slicing* ha dei cicli di CPU accessibili ai thread. Di conseguenza un thread potrebbe o meno essere in esecuzione, nulla gli impedisce di essere eseguito se richiesto dallo scheduler. Non è in alcun modo bloccato o in uno stato *dead*.

…Blocking…

- *Dead*: il classico modo per terminare un thread è quello di uscire dal metodo *run()*. Tuttavia è possibile anche farlo mediante l'invocazione dei metodi *stop()* o *destroy()*. In ogni caso conviene non utilizzare gli ultimi due metodi quando il thread deve eseguire operazioni particolari, in quanto potrebbe lasciare il sistema in uno stato inconsistente. È preferibile forzare l'arresto di un thread attraverso un flag che permetta di uscire al metodo *run()*.

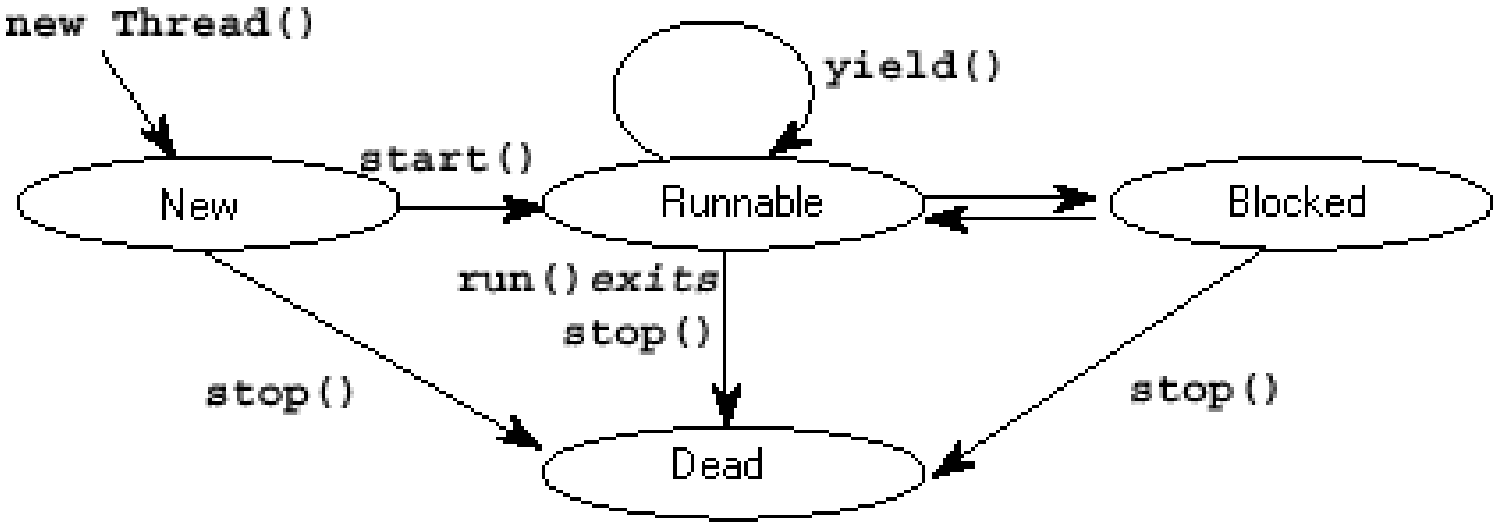
…Blocking…

- *Blocked*: il thread potrebbe essere in esecuzione ma qualcosa lo impedisce. Quando un thread è bloccato lo scheduler lo salterà evitando di concedergli del tempo CPU. Finché il thread non ritorna nello stato *Runnable* non effettuerà alcuna operazione.

Diagramma

Il seguente diagramma mostra graficamente i possibili stati di un thread.

Diagramma di stato di un thread



…Blocking…

Un thread può portarsi nello stato *blocked* per le seguenti ragioni:

- Si è volontariamente sospeso il thread mediante l'invocazione del metodo ***sleep(milliseconds)*** pertanto non continuerà l'esecuzione prima che sia passato il tempo specificato.
- È stata sospesa l'esecuzione del thread con il metodo ***suspend()***. Il thread non tornerà nello stato ***Runnable*** finché non si ha un messaggio di ***resume()***.
- È stata sospesa l'esecuzione del thread con il metodo ***wait()*** pertanto non tornerà nello stato ***Runnable*** finchè non si ha un messaggio di ***notify()*** oppure ***notifyAll()***.

…Blocking.

- Il thread sta aspettando che sia completato un operazione di I/O.
- Il thread sta cercando di chiamare un metodo sincronizzato oppure un altro oggetto non accessibile.

È possibile anche invocare il metodo ***yield()*** per cedere volontariamente la CPU in modo tale che gli altri thread possono essere eseguiti.

Uso di ***suspend()*** e ***resume()***

La classe ***Thread*** ha il metodo ***suspend( )*** per fermare temporaneamente l'esecuzione di un thread fino alla chiamata del metodo ***resume( )*** che fa ripartire il thread dal punto in cui era stata interrotta l'esecuzione.

resume( ) deve essere chiamata da un thread esterno.

wait() e ***notify()***

Sia ***sleep( )*** che ***suspend( )*** non rilasciano la risorsa dopo averla bloccata, in pratica nessun thread può accedervi fino a quando il thread che l'ha bloccata non la rilascia.

Il comportamento di ***wait( )*** è diverso, infatti altri metodi ***synchronized*** dell'oggetto thread possono essere invocati mentre è chiamata una ***wait( )***.

Usando ***wait( )*** è necessario usare ***notify()*** e ***notifyAll()*** per riportare il thread nello stato runnable.

Deadlock…

Siccome i thread possono essere bloccati e siccome gli oggetti possono essere *sincronizzati*, è possibile che un thread sia bloccato in attesa di un altro. La situazione limite si ha nel momento in cui tutti i thread aspettano un altro thread.

In questo caso si verifica una situazione di stallo chiamata *deadlock*.

È abbastanza difficile che si verifichino tali situazioni. Ma quando accadono è praticamente impossibile effettuare il debug.

…Deadlock…

Un tipico caso di deadlock è il seguente:

thread1 esegue il seguente codice:

```
synchronized (a)
  { ... synchronized (b) { ... }
 ... }
```

Contemporaneamente thread2 esegue il seguente codice:

```
synchronized (b)
 { ... synchronized (a) { ... }
 ... }
```

…Deadlock.

Per ridurre il problema del deadlock in Java 1.2 sono stati deprecati i metodi ***stop( )***, ***suspend( )***, ***resume( )***, and ***destroy( ).***

Per ***suspend( )*** e ***resume( )*** il motivo sta nel fatto che volendo riesumare il thread, bisogna attendere che esso sia rilasciato dal thread che l'ha bloccato, potendo generare, così, una situazione di *deadlock.*

Priorità

La priorità di un thread fornisce allo scheduler informazioni sull'ordine di esecuzione.

Nel caso ci sono un certo numero di thread bloccati, in attesa di esecuzioni, lo scheduler lancerà quello con priorità più alta. Ciò non significa che i thread con bassa priorità non verranno mai eseguiti (situazione di deadlock) ma solo che saranno eseguiti meno frequentemente.

Leggere e impostare le priorità…

Per impostare la priorità di thread si deve utilizzare il metodo *setPriority()* della classe *Thread*.

La sintassi generale è la seguente:

```
final void setPriority(int livello)
```

Il livello specifica la nuova impostazione per il thread chiamante. Il valore assunto deve variare nell'intervallo MIN_PRIORITY e MAX_PRIORITY. Attualmente questi valori sono rispettivamente 1 e 10.

Per ripristinare il valore di default, si deve specificare NORM_PRIORITY il cui valore è 5.

…Leggere e impostare le priorità.

Per ottenere l'impostazione attuale della priorità, occorre chiamare il metodo *getPriority()* di *Thread* nel modo seguente:

```
final int getPriority()
```