Gestione dei thread in Java

LSO 2008

Cos'è un Thread?

Si può avere la necessità di suddividere un programma in sotto-attività separate da eseguire indipendentemente l'una dall'altra

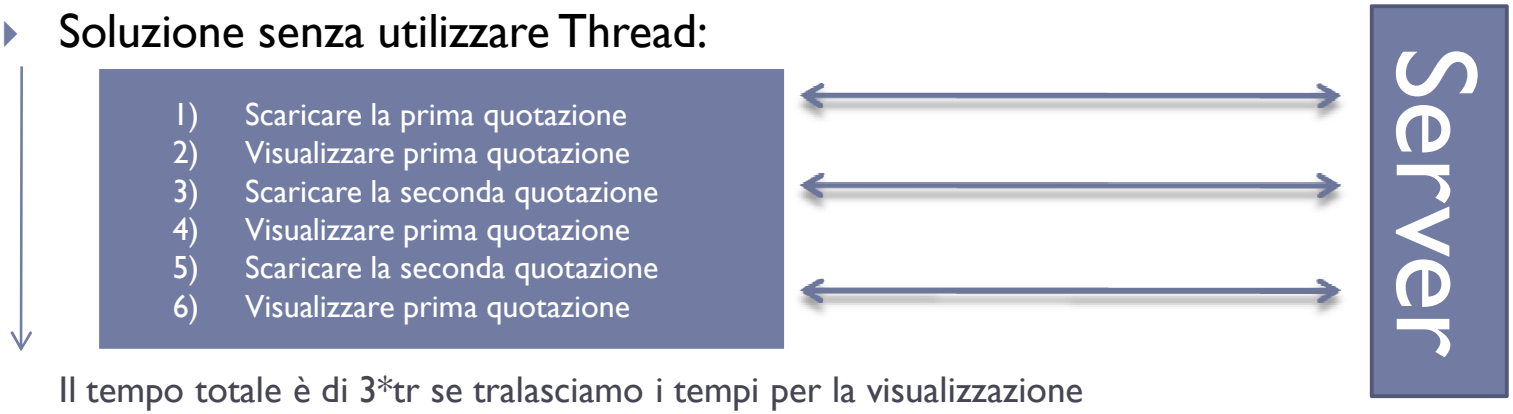
Queste sottoattività prendono il nome di *thread*: esse vengono progettate come entità separate.

Quindi:

- Col termine **processo** si intende un programma in esecuzione sul sistema;
- Col termine **thread** si intende una sotto-attività all'interno di un processo.

Esempio

- Si deve creare un programma che visualizzi l'andamento delle quotazioni di 3 titoli azionari. Le quotazioni vengono scaricate da un server e l'operazione richiede tempo t_r.
- Soluzione senza utilizzare Thread:

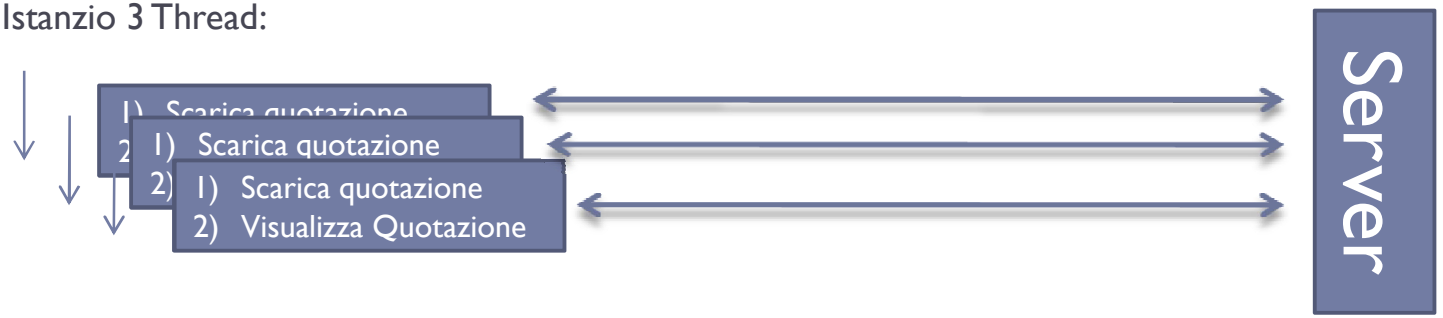


Il tempo totale è di 3*tr se tralasciamo i tempi per la visualizzazione

- Soluzione con l'utilizzo di Thread

Si usano tre Thread separati, che scaricano parallelamente le quote dei tre titoli azionari. Il tempo di esecuzione diminuisce in quanto i Thread verranno eseguiti parallelamente.

Istanzio 3 Thread:



Esempio(2)

- Senza utilizzare i thread, abbiamo un unico flusso di esecuzione;

- Utilizzando i thread, i flussi di esecuzione sono molteplici, all'interno dello stesso processo. Per operazioni complesse, bisognerà sincronizzare i vari thread per farli cooperare e raggiungere il risultato finale.

Concorrenza

Utilizzare il modello dei thread consente di ripartire il tempo di CPU fra tutti i thread in esecuzione. Ciascun thread ha la "sensazione" di avere per se tutta la CPU che in realtà viene ripartita in maniera automatica fra tutti i thread.

L'uso dei thread può in qualche caso ridurre le prestazioni dei programmi, ma in generale i vantaggi in fase di progettazione, uso delle risorse e tempi di risposta rendono il loro uso da preferirsi.

I sistemi che consentono la gestione nei programmi di più thread si dicono multithreading

Stati di un thread

Un thread può trovarsi nei seguenti stati:

1. **Nuovo**: il thread è stato creato con un'operazione di new() ma non è ancora stato mandato in esecuzione.
2. **Eseguibile**: il thread è pronto ad essere eseguito immediatamente appena gli viene assegnata la CPU.
3. **Morto**: il thread ha terminato la sua esecuzione ritornando dal metodo run.
4. **Bloccato**: il thread non può essere eseguito, per esempio potrebbe essere in attesa di avere a disposizione una risorsa.

Stato di avanzamento

Multithreading con java

- In Java i thread sono nativi, cioè supportati a livello di linguaggio. I thread si possono implementare in due modi:

Estendere Thread

```
class mioThread extends Thread {
mioThread(int n) {
                System.out.println("Creato miothread");
}
public void run() {
                // esegue istruzioni
                }
}I
```

Implementare Runnable

```
class mioRun implements Runnable {
mioRun() {
                System.out.println("Creato oggetto");
}
public void run() {
                // esegue istruzioni
}
}
Per creare il thread ed eseguirlo:
Thread t = new Thread(new mioRun());
t.start();
```

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;

public class SimpleThread extends Thread{

        private  int countDown = 10;
        private  static int threadCount = 0;

        public SimpleThread(){
                super(""+ ++threadCount);// crea nome thread
                start();     }// fa partire il thread

        public String toString(){
                return "#" + getName() + ": " + countDown;  }

        public void run(){
                while(true){
                        System.out.println(this);
                        if(--countDown == 0 ) return;
                }
        }

        public static void main(String args[]){
                for(int i=0; i<5;i++) new SimpleThread();          }
}
```

Funzionamento del programma

In quest'esempio si creano 5 thread che eseguono ciascuno un countdown da 10 a 1.

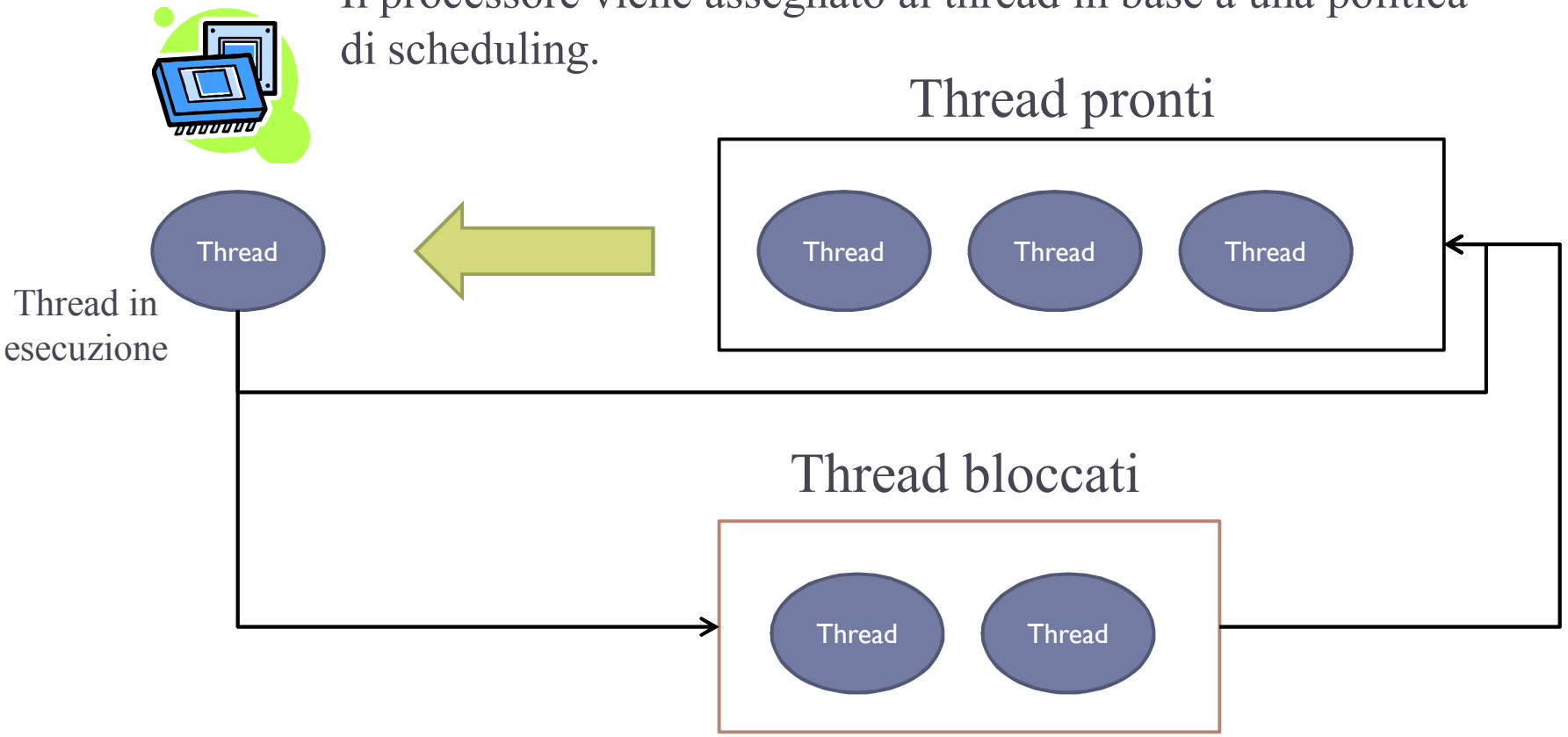
- I thread vengono creati nel main con l'istruzione **new()**.
- I thread vengono definiti e nominati invocando un costruttore della classe Thread con l'istruzione **super()**.
- Successivamente i thread vengono mandati in esecuzione con l'istruzione start che cede il flusso del programma al metodo **run()**.
- Il metodo **getName()** restituisce il nome del thread.

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;

public class SimpleThread extends Thread{

        private  int countDown = 10;
        private  static int threadCount = 0;

        public SimpleThread(){
                super(""+ ++threadCount);// crea nome thread
                start();     }// fa partire il thread

        public String toString(){
                return "#" + getName() + ": " + countDown;  }

        public void run(){
                while(true){
                        System.out.println(this);
                        if(--countDown == 0 ) return;
                }
        }

        public static void main(String args[]){
                for(int i=0; i<5;i++) new SimpleThread();          }
}
```

Non viene mantenuto un riferimento all'oggetto thread, che rimane in vita fino alla terminazione di RUN

Esempio di scheduling dei Thread

Il processore viene assegnato ai thread in base a una politica di scheduling.



Direttive per lo scheduling:yield

E' possibile dare dei "suggerimenti" al meccanismo di scheduling dei thread attraverso il medoto **yield()**.

Questo metodo ci consente di forzare il rilascio del processore richiedendo l'assegnazione ad un altro thread.

Nell'esempio precedentemente questa operazione si può inserire al termine di ciascun ciclo.

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;

public class SimpleThread extends Thread{

        private  int countDown = 10;
        private  static int threadCount = 0;

        public SimpleThread(){
                super(""+ ++threadCount);// crea nome thread
                start();     }// fa partire il thread

        public String toString(){
                return "#" + getName() + ": " + countDown;  }

        public void run(){
                while(true){
                        System.out.println(this);
                        if(--countDown == 0 ) return;
                        yield(); //chiedo rilascio del processore
                }
        }

        public static void main(String args[]){
                for(int i=0; i<5;i++) new SimpleThread();          }
}
```

Direttive per lo scheduling: yield

Con questa variazione l'output del programma *dovrebbe* risultare più ordinato.

Il metodo yield risulta utile solo il particolari situazioni, ma non ci si può affidare ad esso per gestire con precisione i meccanismi di scheduling. Nel caso precedente, se le operazioni precedenti alla yield fossero troppo lunghe, il sistema potrebbe anticipare l'operazione di yield e rilasciare il processore.

La yield inoltre non consente di indicare a quale thread passare il controllo.

Direttive per lo scheduling: sleep

E' possibile controllare il comportamento dei thread anche con il metodo **sleep()**, che forza la sospensione di un thread per un determinato numero di millisecondi. Questo metodo richiede la gestione dell'eccezione **InterruptedException.**

Nell'esempio l'operazione viene posta alla fine di un ciclo.

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;

public class SimpleThread extends Thread{

        private  int countDown = 10;
        private  static int threadCount = 0;

        public SimpleThread(){
                super(""+ ++threadCount);// crea nome thread
                start();     }// fa partire il thread
        public String toString(){
                return "#" + getName() + ": " + countDown;  }

        public void run(){
                while(true){
                        System.out.println(this);
                        if(--countDown == 0 ) return;
                        try{
                                sleep(100);
                        }catch(InterruptedException e){
                                throw new RuntimeException(e);}
                }
        }
       public static void main(String args[]){
                for(int i=0; i<5;i++) new SimpleThread();          }
}
```

Direttive per lo scheduling: sleep

In questo caso i thread sono tutti alternati:

#1: 10
#2: 10
#3: 10
#4: 10
#5: 10
#1: 9
….

Neanche il metodo sleep fornisce comunque garanzie sull'ordine di esecuzione dei thread: il thread è sospeso per *almeno* 100 millisecondi, ma potrebbe essere ritardato ulteriormente né si può specificare il thread a cui passare il controllo.

Direttive per lo scheduling: priorità

La priorità di un thread ne determina la sua importanza per lo scheduler, che tenterà di privilegiare quelli con priorità più alta. Quelli con priorità più bassa verranno ugualmente eseguiti, ma meno spesso.

Le priorità possono essere impostate col metodo **setPriority()** della classe Thread**.**

Il metodo **getPriority()** fornisce il valore della priorità di un thread.

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;

public class SimpleThreadPriority2 extends Thread{

	private  int countDown = 50;
	private  static int threadCount = 0;
	public SimpleThreadPriority2(int priority){
		setPriority(priority);
		start();
	}

	public String toString(){
		return super.toString() + ": " + countDown; }

	public void run(){
		while(true){
			System.out.println(this);
			if(--countDown == 0 ) return;
		}	}

	  public static void main(String args[]){
		for(int i=0; i<5;i++) {
	if(i==0){ new SimpleThreadPriority2(Thread.MAX_PRIORITY);}
	if(i!=0) {new SimpleThreadPriority2(Thread.MIN_PRIORITY);}
		}		}
}
```

Direttive per lo scheduling: priorità

Il risultato dell'esempio precedente è circa:

……

Thread[Thread-0,10,main]: 2

Thread[Thread-0,10,main]: 1

Thread[Thread-1,1,main]: 50

Il thread 0 finisce l'esecuzione senza essere interrotto, poi iniziano gli altri.

Vengono riportati: nome thread, priorità, gruppo di thread di appartenenza e valore di conteggio.

Le scale di priorità dei thread variano da sistema a sistema, benchè Java fornisca 10 livelli. Le constanti MAX_PRIORITY, MIN_PRIORITY e NORM_PRIORITY consentono la portabilità sui vari sistemi.

Thread Daemon

Un thread *daemon* fornisce un servizio generale e non essenziale in background mentre il programma esegue altre operazioni. Dunque il programma termina quando tutti i thread NON-daemon terminano.

Il metodo **setDaemon()** imposta un thread come daemon.
Il metodo **isDaemon()** verifica se un thread è un daemon.

Tutti i thread creati da un daemon sono a loro volta dei daemon.

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;

public class SimpleDaemon extends Thread{

	public SimpleDaemon(){
		setDaemon(true);
		start();
	}

	public void run(){
		while(true){
			try{
				sleep(100);
			}catch(InterruptedException e){
				throw new RuntimeException(e);}
			System.out.println(this);
		}
	}

	public static void main(String args[]){
		for(int i=0; i<10;i++) new SimpleDaemon();

	}
}
```

Imposto il thread come Daemon prima dell'operazioen di start.

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;

public class SimpleDaemon extends Thread{

        public SimpleDaemon(){
                setDaemon(true);
                start();
        }

        public void run(){
                while(true){
                        try{
                                sleep(100);
                        }catch(InterruptedException e){
                                throw new RuntimeException(e);}
                        System.out.println(this);
                }
        }

        public static void main(String args[]){
                for(int i=0; i<10;i++) new SimpleDaemon();

        }
}
```

Imposto il thread come Daemon

Quale è l'output di questo programma?

Boh!
Dipende se i thread riescono a stampare qualcosa prima che il thread principale termini

Join()

I thread possono chiamare il metodo **join()** su un altro oggetto thread per attendere la terminazione di quest'ultimo prima di proseguire con l'esecuzione.

Es: t.join sospende il thread che esegue questa operazione finchè t non termina. Join() accetta anche un parametro che specifichi un tempo massimo di attesa.

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;
public class SimpleJoin extends Thread{
    private  int flag; private SimpleJoin Dormiente;
    public SimpleJoin(int i){
        flag = i;       start();    }

    public SimpleJoin(int i, SimpleJoin j){
        flag = i;   Dormiente = j;  start();    }

    public void run(){
        if(flag==1){
            try{sleep(1500);
            }catch(InterruptedException e){
                throw new RuntimeException(e);}
            System.out.println(getName()+ " si sveglia");     }
        if(flag==2){
            try{Dormiente.join();
            }catch(InterruptedException e){
                throw new RuntimeException(e);}
            System.out.println(getName()+ " riprende l'esecuzione");
        }   }

     public static void main(String args[]){
            SimpleJoin Dormiente = new SimpleJoin(1);
            SimpleJoin Paziente = new SimpleJoin(2, Dormiente);   }
}
```

Join()

Il thread *Paziente* è creato tramite un costruttore che riceve come parametro il riferimento al thread su cui rimanere in attesa. Paziente invoca l'operazione join() e attende che il thread *Dormiente* si svegli e termini (questo appena termina la sleep). Flag è usato semplicemente per distinguere il codice di Paziente e Dormiente.

L'output del programma è:

Thread-0 si sveglia

Thread-1 riprende l'esecuzione

Accedere a risorse condivise

La programmazione concorrente con i thread pone il problema della gestione degli accessi a risorse condivise.

Java fornisce un meccanismo per gestire le collisioni basato sulla parola chiave **synchronized.** Quando un thread esegue parti di codice racchiuse da questa parola ha la garanzia di non essere interrotto da altri thread. Si possono proteggere in questo modo i metodi di accesso ad una classe dichiarandoli synchronized.

Accedere a risorse condivise

Per controllare l'accesso ad una risorsa si può incapsularla all'interno di un oggetto e dichiarare synchronized i metodi di accesso. In questo modo se un thread esegue **uno** dei metodi synchronized, vengono bloccati **altri** thread che cercano di eseguire **altri metodi synchronized**: la chiamata ad un metodo synchronized pone un blocco su tutti i metodi synchronized dell'oggetto. Gli altri thread vengono sbloccati quando è terminata l'esecuzione del primo thread.

synchronized void a(){….}

synchronized void b(){….}

Accedere a risorse condivise

Il blocco posto sull'oggetto non vale per il thread che ha effettuato la prima chiamata su codice synchronized: se questo infatti chiama altri metodi synchronized viene tenuta traccia del numero di blocchi richiesti. L'oggetto diviene di nuovo disponibile solo quando il thread rilascia tutti blocchi.

Thread bloccato

Un thread può trovarsi bloccato per vari motivi:

1. E' in attesa della scadenza di un certo intervallo di tempo – chiamata al metodo **sleep()**.
2. Il thread attende la terminazione di un'operazione di I/O.
3. Il thread sta cercando di accedere ad un oggetto o ad un blocco di codice **synchronized.**
4. Il thread è stato sospeso con una chiamata al metodo **wait()**.
5. Nelle versioni precedenti di Java 2, il thread poteva essere sospeso con una chiamata ai metodi **suspend()** e **resume()** ora deprecati.

Come gestire i thread

Per poter gestire l'esecuzione dei thread è importante disporre di metodi per bloccarne l'esecuzione o farla riprendere. Per questi scopi si può disporre dei metodi **wait(), notify(),** e **notifyAll()**.

Un altro metodo visto che può essere usato per bloccare un thread è **sleep(),** anche se sleep non rilascia le risorse prese da thread.

wait()

Quando un thread esegue una **wait()** da un oggetto, tale thread viene sospeso, e il blocco sull'oggetto viene rilasciato.

Il metodo wait() rilascia il blocco detenuto su un oggetto, quindi altri metodi synchronized posso essere richiamati da altri thread.

Wait() può essere chiamato in 2 modi:

1. Specificando un intervallo di tempo massimo (in millisecondi) di attesa.
2. Senza specificare l'intervallo. Il thread rimane in attesa indefinita.

notify() e notifyAll()

E' possibile per un thread riprendere l'esecuzione sospesa attraverso i metodi **notify** e **notifyAll().**

Sia wait() che notify e notifyAll devono essere chiamate all'interno di blocchi sincronizzati, altrimenti generano eccezioni di tipo **IllegalMonitorStateException.**

Terminazione dei thread

Con Java 2 sono stati dichiarati deprecati i metodi **stop(), suspend() e resume()** della classe thread.

- Terminando un thread con **stop** si forza il rilascio dei blocchi sugli oggetti, che possono comunque essere al momento in uno stato inconsistente e risultare poi visibili agli altri thread.
- **Suspend** sospende un processo senza rilasciare i blocchi ottenuti dal processo stesso (possibile causa di deadlock). **Resume** si usa insieme a **suspend**.

Terminazione dei thread

Il metodo **destroy()** termina un thread senza alcuna operazione di gestione della terminazione stessa.

Come terminare un thread

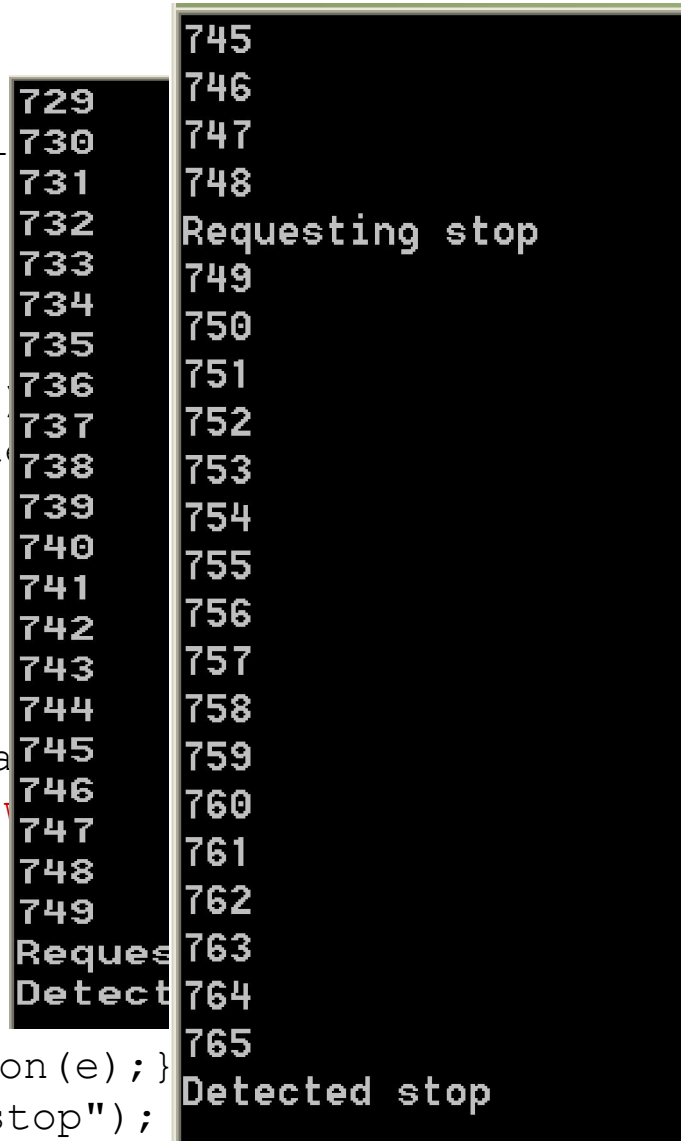
Per terminare un thread è consigliabile, al posto del metodo stop(), l'uso di un flag che indichi al thread di terminare uscendo dal suo metodo run().

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;
class CanStop extends Thread{

    private  volatile boolean stop = false;
    private  static int counter = 0;

    public void run(){
        while(!stop && counter<10000){
            System.out.println(counter++);}
        if (stop)System.out.println("Detected stop");
    }
    public void requestStop(){stop=true;}
}

public class Stopping{
        public static void main(String args[]){
            final CanStop stoppable = new CanStop();
        stoppable.start();
        try{
            Thread.sleep(500);
        }catch(InterruptedException e){
                throw new RuntimeException(e);}
        System.out.println("Requesting stop");
        stoppable.requestStop();
    }
}
```

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;
class CanStop extends Thread{

    private  volatile boolean stop = fal
    private  static int counter = 0;

    public void run(){
        while(!stop && counter<10000){
            System.out.println(counter++
        if (stop)System.out.println("Det
    }
    public void requestStop(){stop=true;
}

public class Stopping{
        public static void main(String a
            final CanStop stoppable = ne
        stoppable.start();
        try{
            Thread.sleep(500);
        }catch(InterruptedException e){
                throw new RuntimeException(e);}
        System.out.println("Requesting stop");
        stoppable.requestStop();
    }
}
```

```
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
Requesting stop
Detected stop
```

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;
class CanStop extends Thread{

    private  volatile boolean stop = fal
    private  static int counter = 0;

    public void run(){
        while(!stop && counter<10000){
            System.out.println(counter++
        if (stop)System.out.println("Det
    }
    public void requestStop(){stop=true;
}

public class Stopping{
        public static void main(String a
            final CanStop stoppable = ne
        stoppable.start();
        try{
            Thread.sleep(500);
        }catch(InterruptedException e){
                throw new RuntimeException(e);}
        System.out.println("Requesting stop");
        stoppable.requestStop();
    }
}
```



```
// ESEMPIO DI USO DEI THREAD
import java.io.*;
class CanStop extends Thread{

    private  volatile boolean stop = false;
    private  static int counter = 0;

    public void run(){
        while(!stop && counter<10000){
            System.out.println(counter++);}
        if (stop)System.out.println("Detected stop");
    }
    public void requestStop(){stop=true;}
}

public class Stopping{
        public static void main(String args[]){
            final CanStop stoppable = new CanStop();
        stoppable.start();
        try{
            Thread.sleep(500);
        }catch(InterruptedException e){
                throw new RuntimeException(e);}
        System.out.println("Requesting stop");
        stoppable.requestStop();
    }
}
```

Interrompere un thread bloccato

In alcune situazioni un thread rimane bloccato e non può verificare il flag di terminazione.

In questo caso si può utilizzare il metodo **interrupt()** della classe Thread per interrompere il thread bloccato.

Interrupt genera un'eccezione – per questo i metodi wait e sleep vanno inseriti all'interno di clausole try-catch().

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;

class Blocked extends Thread{

    public Blocked(){
        System.out.println("Starting Blocked");
        start();}
    public void run(){
        try{
            synchronized(this){wait();}
        }catch(InterruptedException e){
                System.out.println("Interrupted");     }
        System.out.println("Exit run()");  }
}

public class Interrupt{
    static Blocked blocked = new Blocked();
    public static void main(String args[]){
        try{
            Thread.sleep(500);
        }catch(InterruptedException e){
                throw new RuntimeException(e);}
        System.out.println("Preparing to interrupt");
        blocked.interrupt();
        blocked = null; }
}
```

```
// ESEMPIO DI USO DEI THREAD
import java.io.*;

class Blocked extends Thread{

    public Blocked(){
        System.out.println("Starting Blocked");
        start();}
    public void run(){
        try{
            synchronized(this){wait();}
        }catch(InterruptedException e){
                System.out.println("Interrupted");     }
        System.out.println("Exit run(
}

public class Interrupt{
    static Blocked blocked = new Bloc
    public static void main(String ar
        try{
            Thread.sleep(500);
        }catch(InterruptedException e
                throw new RuntimeExce
        System.out.println("Preparing to interrupt");
        blocked.interrupt();
        blocked = null; }
}
```

```
C:\Documents and Settings\
C:\Documents and Settings\
Starting Blocked
Preparing to interrupt
Interrupted
Exit run()

C:\Documents and Settings\
```