



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Programmazione parallela con i Java Thread – parte 1



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



Thread

Thread: flusso sequenziale di controllo (esecuzione di istruzioni) in un programma.

Nello stesso programma si possono far partire più *Thread* che sono eseguiti in modo concorrente.

Tutti i *Thread* condividono le stesse variabili del programma.

A differenza dai *processi* che hanno ciascuno il proprio contesto, PID, etc..

Un *Thread* viene visto come un *lightweight process*.

Nei computer a singola CPU la concorrenza viene simulata con una politica di *scheduling* che alterna l'esecuzione dei singoli *Thread*.



Una applicazione Java che usa i Thread può eseguire più attività contemporaneamente. Esempio: aggiornare l'informazione grafica sullo schermo e accedere alla rete.

In alcuni casi i Thread possono procedere in modo indipendente uno dall'altro (comportamento asincrono), in altri devono essere sincronizzati fra loro (es. produttore - consumatore).

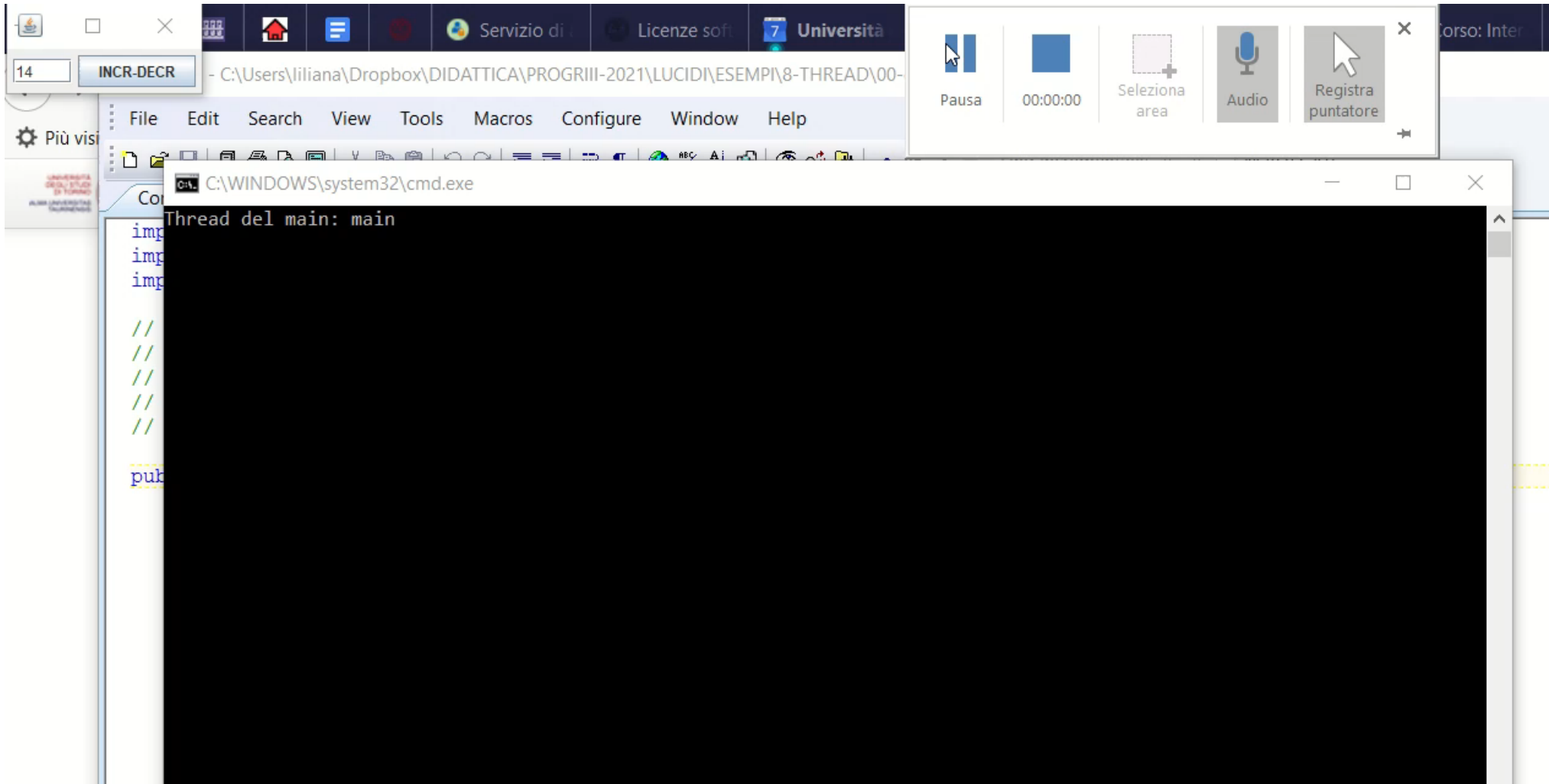
Ad esempio, quando un main crea una finestra (JFrame), viene attivato un **Thread** di interfaccia utente, diverso da quello del main().



Il costruttore del **JFrame** è eseguito dal Thread del main(), mentre il codice dei *listener* viene eseguito dal nuovo Thread (che gestisce tutti i listener).

I Thread sono indispensabili per realizzare interfacce grafiche che rispondano prontamente ai comandi dell'utente, anche se il programma sta eseguendo altre computazioni. In generale, li si usa per ottenere un uso ottimale della CPU (facendo in modo che un flusso di esecuzione si sospenda quando è in attesa di condizioni, I/O, etc., e lasciando la CPU a flussi che possono essere portati avanti).

Esecuzione di Contatore2

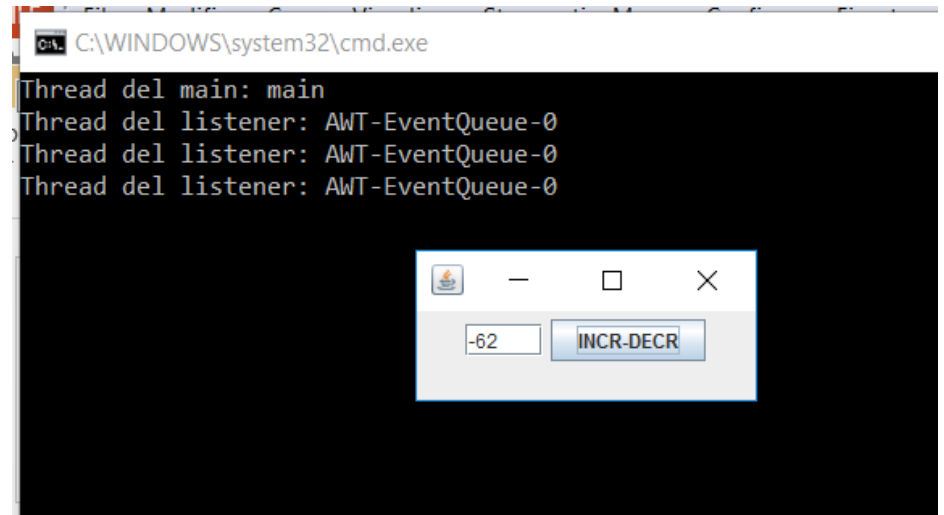




Esempio: Contatore 2

La label visualizza il valore della variabile intera *count*, mentre il bottone può cambiare il valore della variabile booleana *runFlag*.

Dopo aver creato la finestra, il `main()` entra in un ciclo infinito in cui la variabile *count* viene incrementata o decrementata a seconda del valore di *runFlag*. Nonostante il `main()` esegua una computazione che non termina, è possibile eseguire il metodo `actionPerformed()` del listener del bottone ogni volta che questo viene premuto.



```

public class Contatore2 {
    private static boolean runFlag = true;

    public static void main(String[] argv) {
        JButton onOff = new JButton("ON-OFF");
        JTextField t = new JTextField(4);
        JFrame f = new JFrame();
        int count = 0;
        setLayout(new FlowLayout());
        onOff.addActionListener(new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    runFlag = !runFlag;
                }
            });
        f.add(onOff);
        f.add(t);
        .....
        while (true) {
            try {Thread.sleep(500);}
            catch(InterruptedException e) {}
            if (runFlag) t.setText(Integer.toString(count++));
            else t.setText(Integer.toString(count--));
        }
    }
}

```



classe Thread



sleep(long millis) è un metodo di Thread che blocca l'esecuzione del Thread per il numero specificato di millisecondi.

Può generare una eccezione **InterruptedException**.

```
try {  
    Thread.sleep(500);  
} catch(InterruptedException e) {...}
```

sleep() è un metodo statico e quindi può essere usato con **Thread.sleep(...)** anche in una classe che non deriva da Thread, per far bloccare l'esecuzione del Thread che lo esegue.



Per sapere in quale Thread ci si trova si può usare il metodo statico `currentThread()`:

`Thread.currentThread().getName()`

Nell'esempio si può verificare che, quando si usa una interfaccia grafica, ci sono (almeno) due Thread. Uno per il `main()` e l'altro per l'interfaccia grafica.

Anche se si creano più finestre, c'è un solo Thread per l'esecuzione di tutte le finestre.



Esempio: Contatore 1 – non funziona bene

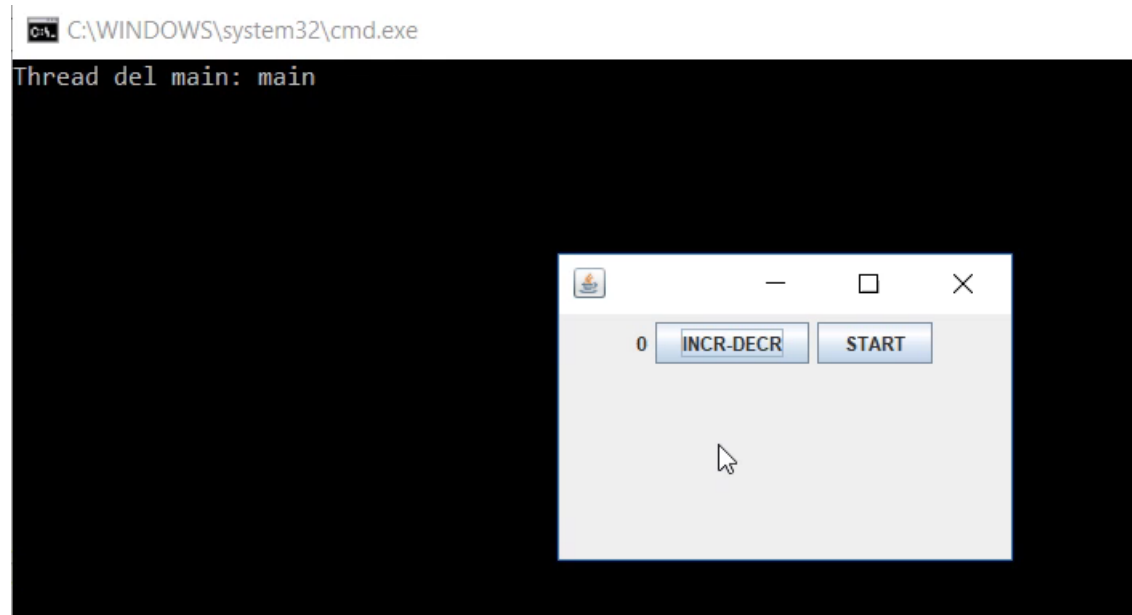
Vediamo adesso un altro esempio analogo al precedente (**Contatore1**).

In questo caso la computazione infinita viene eseguita dal listener di un bottone START, che impedisce qualunque altra computazione dell'interfaccia grafica, in particolare la gestione di altri eventi.

Dopo aver premuto START, la finestra non viene più rinfrescata.

Il bottone ON-OFF non ha effetto e non si riesce a chiudere la finestra e terminare l'esecuzione.

Esecuzione di Contatore1





```
public class Contatore1 extends JFrame
```

```
{    private int count = 0;
    private JButton onOff = new JButton("ON-OFF");
    private JButton start = new JButton("START");
    private JTextField t = new JTextField(4);
    private boolean runFlag = true;
    public Contatore1()
    { ..... }
```

```
class OnOffL implements ActionListener
```

```
{    public void actionPerformed(ActionEvent e)
    {    runFlag = !runFlag;
    }
}
```

```
class StartL implements ActionListener {
```

```
    public void actionPerformed(ActionEvent e) {
        while (true) {
            try {Thread.sleep(500);}
            catch(InterruptedException exc) {}
            if (runFlag) t.setText(Integer.toString(count++));
            else t.setText(Integer.toString(count--));
            System.out.println(count);
        }
    }
}
```

```
}}}}
```

Come si crea un Thread

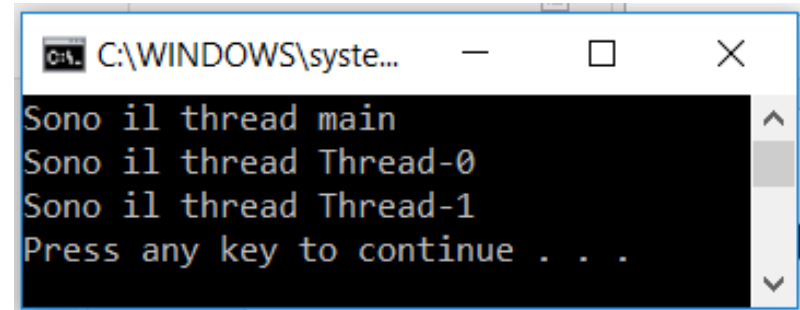


Si definisce una classe che eredita da **Thread** e che ridefinisce il metodo **run()** per specificare le operazioni che il Thread deve fare. Il metodo **run()** di **Thread** non fa niente.

```
class MiaClasse extends Thread {  
    public void run() {  
        System.out.println("Sono il thread " + getName());  
    }  
}
```

....

```
MiaClasse t1 = new MiaClasse();  
MiaClasse t2 = new MiaClasse();  
t1.start();  
t2.start();
```



Per avviare un Thread si deve eseguire il metodo **start()**, che manda in esecuzione un nuovo Thread e poi invoca **run()**; **start()** lancia una eccezione se viene invocato più di una volta. NB: se si invoca direttamente il metodo **run()**, questo viene eseguito ma non viene creato un nuovo Thread.

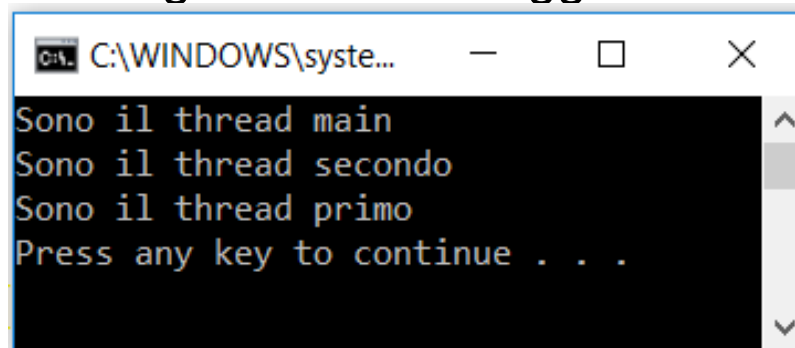


Altra formulazione: **start()** viene chiamato direttamente dal costruttore del **Thread**.

```
class MiaClasse extends Thread
{ public MiaClasse(String s)
  { super(s);
    start();
  }
  public void run()
  {System.out.println("Sono il thread " + getName());}
}

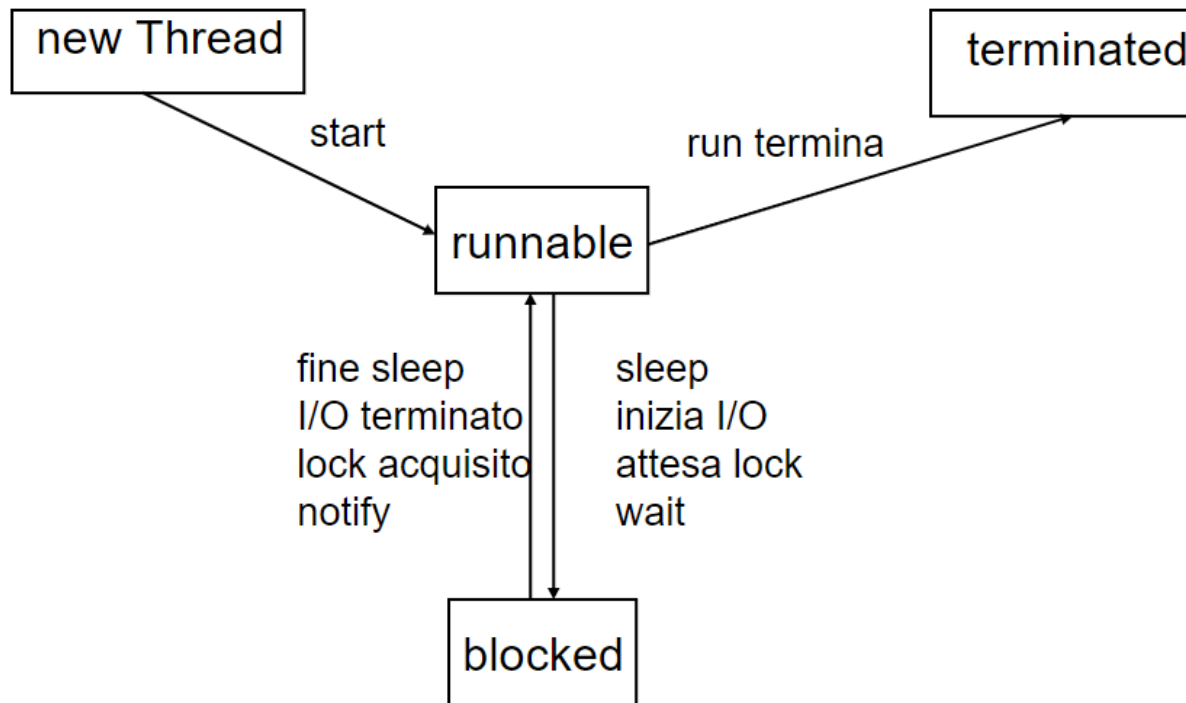
....
new MiaClasse("primo");
new MiaClasse("secondo");
```

NOTA Anche se l'oggetto MiaClasse non viene assegnato ad una variabile non ci sono problemi col Garbage Collector: l'oggetto rimane finché il Thread non termina.



```
C:\WINDOWS\system...
Sono il thread main
Sono il thread secondo
Sono il thread primo
Press any key to continue . . .
```

Ciclo di vita di un Thread





NB: runnable NON significa running!

Un Thread **runnable** *può* essere o meno in esecuzione.

Se c'è una sola CPU ci sarà al massimo un Thread in esecuzione ad ogni istante. L'effettiva esecuzione dipende dalla politica dello scheduler.

La scelta di quale Thread eseguire e per quanto tempo è arbitraria (es. preemptive scheduling).

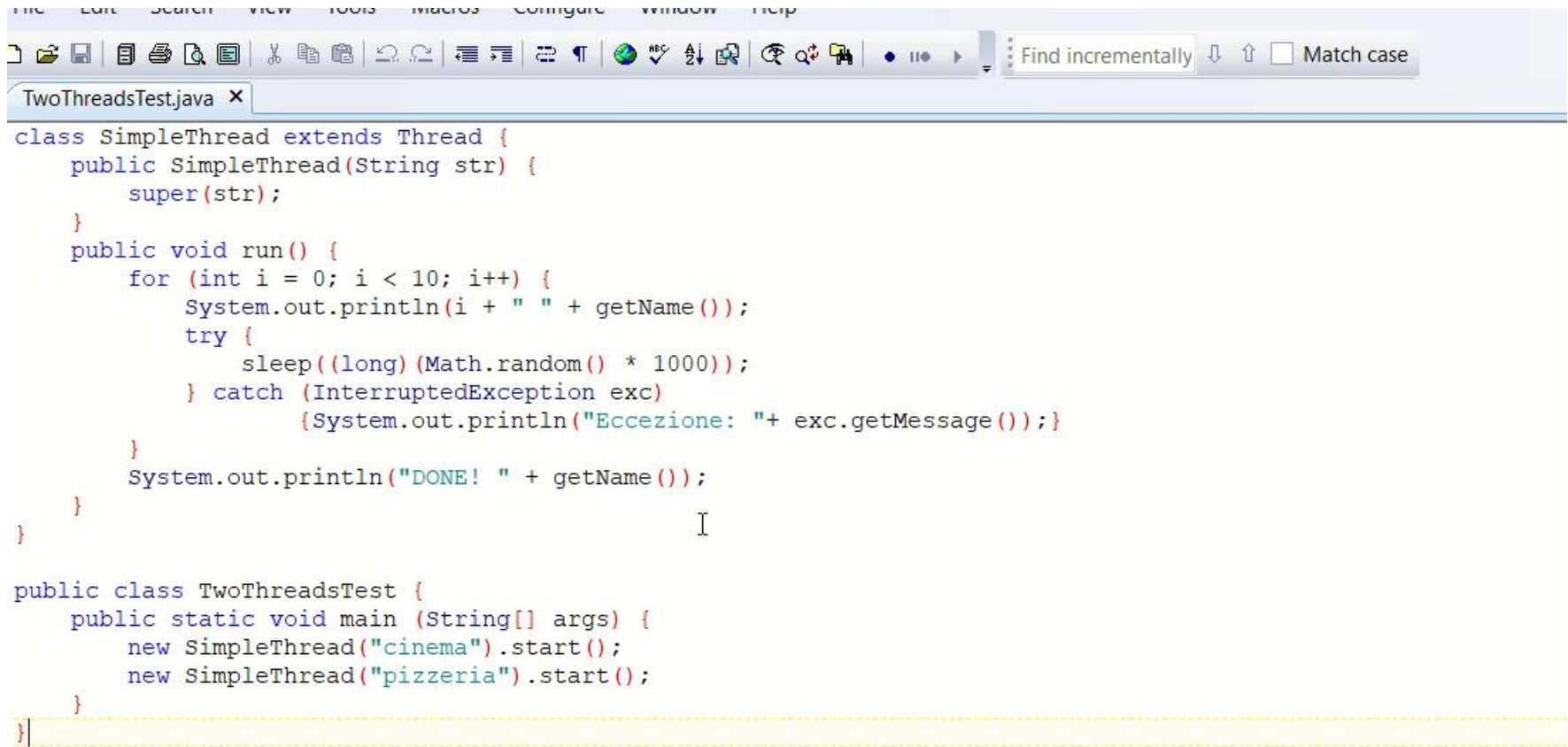
Esempio: TwoThreadsTest



```
class SimpleThread extends Thread {
    public SimpleThread(String str) {super(str);}
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {sleep((long)(Math.random() * 1000));}
            catch (InterruptedException exc)
                {System.out.println("Eccezione: " + exc.getMessage());}
        }
        System.out.println("DONE! " + getName());
    }
}

public class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("cinema").start();
        new SimpleThread("pizzeria").start();
    }
}
```

TwoThreadsTest - esecuzione



```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long) (Math.random() * 1000));
            } catch (InterruptedException exc) {
                System.out.println("Eccezione: " + exc.getMessage());
            }
            System.out.println("DONE! " + getName());
        }
    }
}

public class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("cinema").start();
        new SimpleThread("pizzeria").start();
    }
}
```



Ritorniamo all'esempio iniziale (Contatore1) in cui il listener di un bottone esegue una computazione infinita, bloccando il funzionamento del resto dell'interfaccia.

Si può risolvere il problema creando un nuovo Thread e eseguendo il ciclo infinito del listener in questo Thread.

In questo modo il Thread dell'interfaccia grafica può servire gli altri eventi.

ContConThread

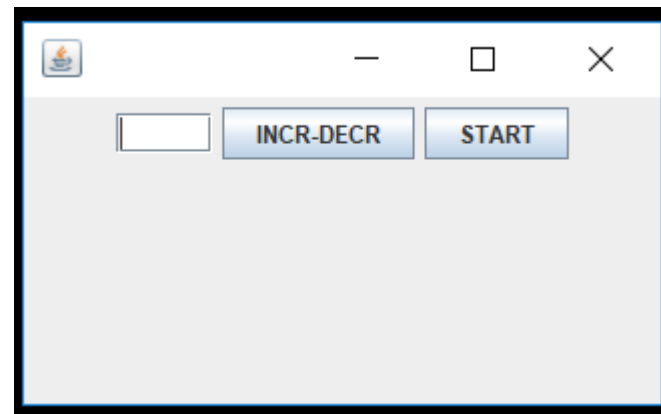


```
ThreadCont tc = new ThreadCont();
```

```
.....
```

```
class ThreadCont extends Thread {  
    public void run() {  
        while (true) {  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException exc) {}  
            if (runFlag) t.setText(Integer.toString(count++));  
            else t.setText(Integer.toString(count--));  
        }  
    }  
}
```

```
class StartL implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        tc.start();  
    }  
}
```



Gestione eccezioni nei Thread



Il metodo `run()` di un Thread non può fare throw di eccezioni controllate → vanno gestite tutte con opportuni handler (try e catch).

Se si verifica un'eccezione non controllata si blocca il Thread e si visualizza a video lo stack di esecuzione.

Volendo gestire le eventuali eccezioni non controllate: aggiungere al Thread un `UncaughtExceptionHandler`, implementando il suo metodo `uncaughtException()` per gestire le eccezioni. **Vd. EccezioneApp1**



EccezioneApp1 - I

```
public MyThread() {  
    super();  
    setUncaughtExceptionHandler(new UncaughtExceptionHandler() {  
        public void uncaughtException(Thread th, Throwable exc) {  
            System.out.println("Eccezione:" + exc.getMessage());  
        }  
    });  
}  
  
public void run() {  
    // genero eccezione non controllata  
    // (RuntimeException);  
    String s=null;  
    s.toString();  
}  
}
```

(IO CONTINUO... viene visualizzato dal main())

```
C:\WINDOWS\system...  
Eccezione:null  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
Press any key to continue . . .
```

EccezioneApp1 - II



```
public MyThread() {  
    super();  
    /* setUncaughtExceptionHandler(new UncaughtExceptionHandler() {  
        public void uncaughtException(Thread th, Throwable exc) {  
            System.out.println("Eccezione:" + exc.getMessage());  
        }  
    }); */  
}
```

```
public void run() {  
    // genero eccezione non controllata  
    // (RuntimeException);  
    String s=null;  
    s.toString();  
}
```

```
C:\WINDOWS\system32\cmd.exe  
Exception in thread "Thread-0" java.lang.NullPointerException  
    at MyThread.run(EccezioneApp1.java:15)  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
IO CONTINUO...  
Press any key to continue . . .
```



Interface Runnable

Si definisce una classe che implementa l'interfaccia **Runnable** che possiede il metodo **run()**.

```
class Esempio implements Runnable {  
    public void run() {...}}
```

Per attivare un Thread è necessario creare un **Thread** passando come parametro al costruttore un oggetto **Runnable**. Quando si fa partire il Thread con **start()**, inizia l'esecuzione del metodo **run()** nel nuovo Thread.

```
Esempio es = new Esempio();  
Thread t = new Thread(es),  
t.start();
```




```
class MiaClasse implements Runnable {  
    public void run() {  
        System.out.println("Sono il thread " +  
            Thread.currentThread().getName());  
    }  
}
```

....

```
MiaClasse mt = new MiaClasse();  
new Thread(mt).start();  
new Thread(mt).start();
```

Questo secondo modo di creare i Thread deve essere usato quando la classe contenente il metodo **run()** è già sottoclasse di un'altra classe (ereditarietà singola). **In generale, usare i Runnable è utile per separare le specifiche delle istruzioni da eseguire in un flusso parallelo rispetto all'esecutore, che è l'oggetto Thread.**

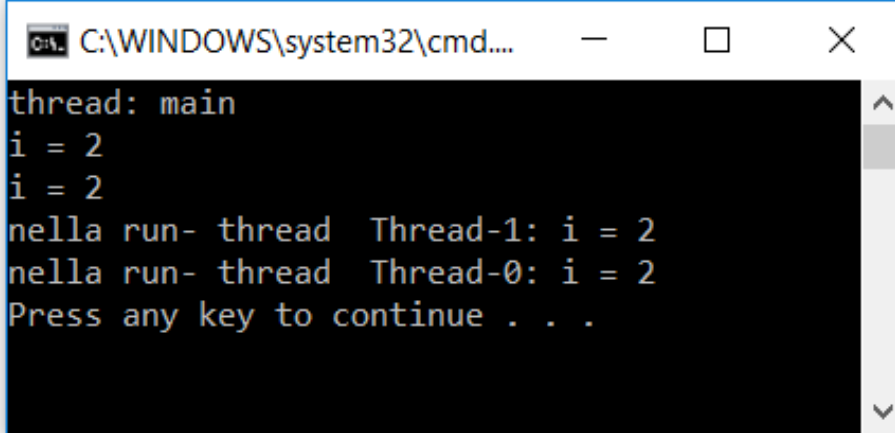
ProvaRunnable2 - I



```
class MiaClasse implements Runnable {  
    private int i = 0;  
    public void run() {  
        i++;  
        System.out.println(i);  
    }  
}
```

....

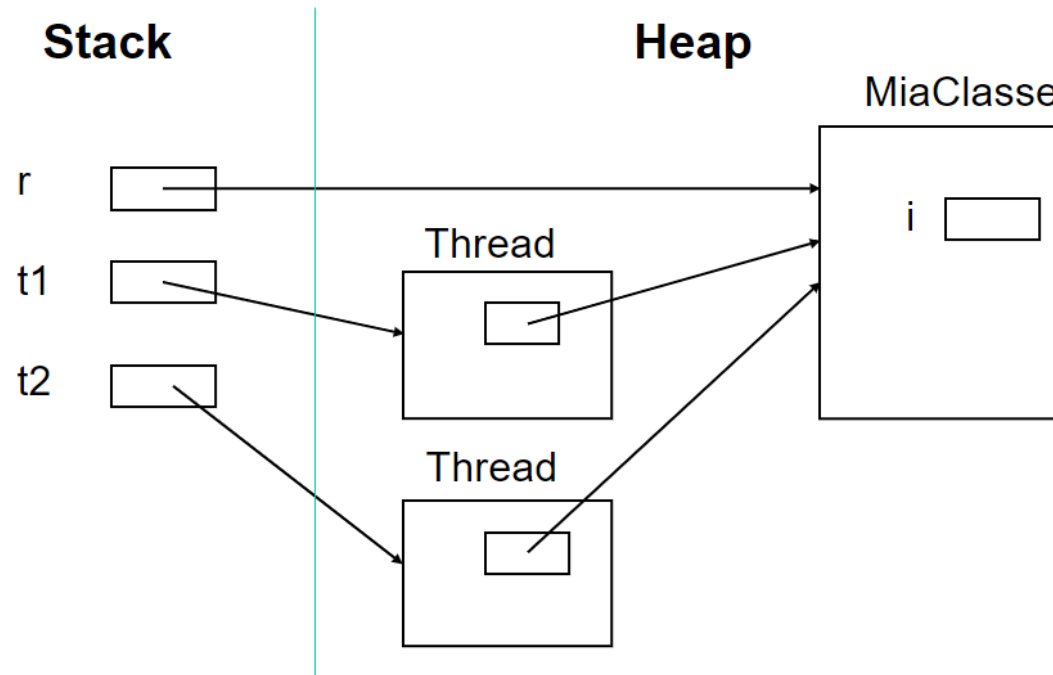
```
MiaClasse r = new MiaClasse();  
Thread t1 = new Thread(r);  
Thread t2 = new Thread(r);  
t1.start();  
t2.start();
```



```
C:\WINDOWS\system32\cmd...  
thread: main  
i = 2  
i = 2  
nella run- thread Thread-1: i = 2  
nella run- thread Thread-0: i = 2  
Press any key to continue . . .
```

La variabile *i* viene incrementata due volte. Infatti esiste un unico oggetto di **MiaClasse**, legato alla variabile *r*, che ha una variabile locale *i*. I Thread **t1** e **t2** eseguono entrambi il metodo **run()** di questo oggetto e incrementano la stessa variabile *i*.

ProvaRunnable2 - II





Per capire meglio il funzionamento dei Thread introduciamo un modello semplificato di esecuzione, che è una estensione di quello per l'esecuzione di programmi sequenziali (vedi slide su gestione della memoria).

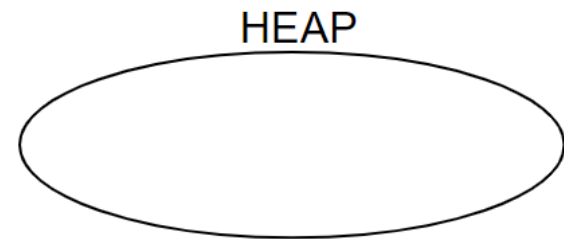
La differenza principale, se il programma contiene dei Thread, è dovuta al fatto che **ogni Thread ha un suo stack di esecuzione**. Invece lo heap, viene condiviso.

Inoltre, ad ogni passo, la macchina virtuale si dovrà ricordare quale è il Thread *running*.



```
class EsempioT extends Thread {  
    private int counter = 0;  
    public void run() {  
        incr();  
        return;  
    }  
    public void incr() {  
        counter++;  
        return;  
    }  
    public static void main(String[] args){  
        EsempioT t1 = new EsempioT();  
        EsempioT t2 = new EsempioT();  
        t1.start();  
        t2.start();  
    }  
}
```

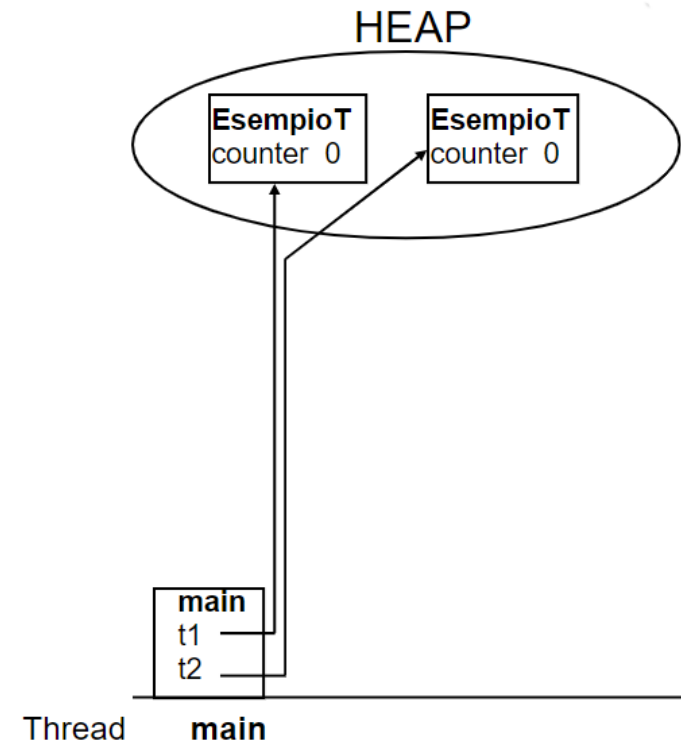
Running: Thread main
In **grassetto** l'istruzione
che viene eseguita.





```
class EsempioT extends Thread {  
    private int counter = 0;  
    public void run() {  
        incr();  
        return;  
    }  
    public void incr() {  
        counter++;  
        return;  
    }  
    public static void main(String[] args){  
        EsempioT t1 = new EsempioT();  
        EsempioT t2 = new EsempioT();  
        t1.start();  
        t2.start();  
    }  
}
```

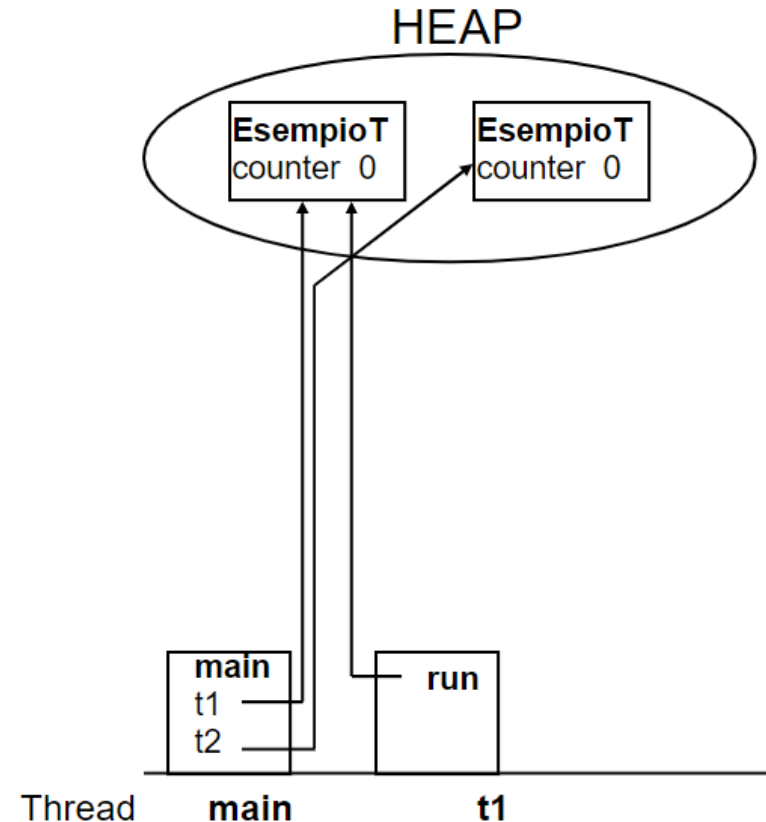
Running: Thread main





```
class EsempioT extends Thread {  
    private int counter = 0;  
    public void run() {  
        incr();  
        return;  
    }  
    public void incr() {  
        counter++;  
        return;  
    }  
    public static void main(String[] args){  
        EsempioT t1 = new EsempioT();  
        EsempioT t2 = new EsempioT();  
        t1.start();  
        t2.start();  
    }  
}
```

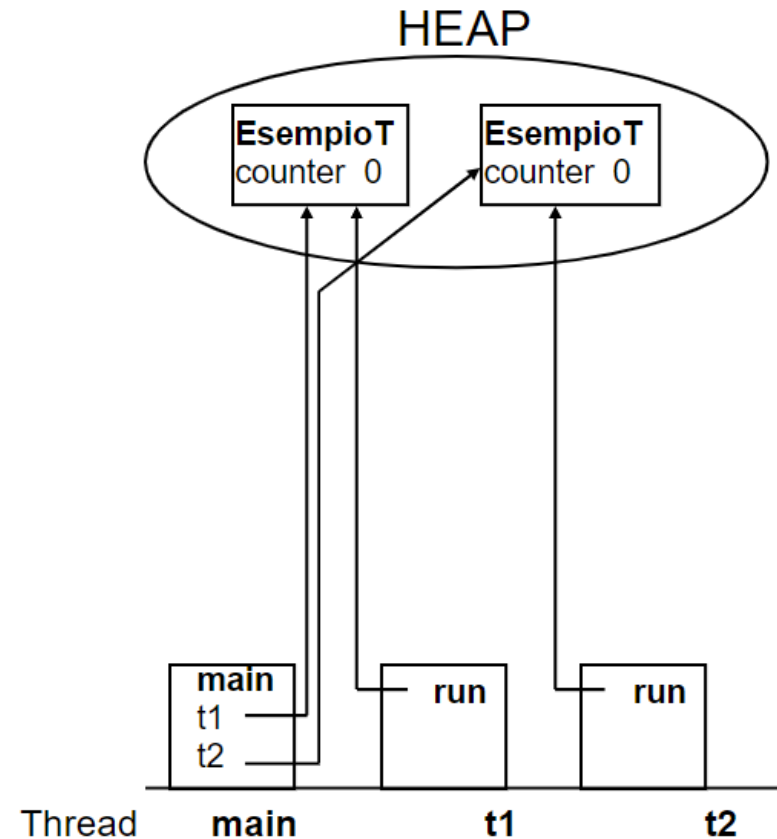
Running: Thread main





```
class EsempioT extends Thread {  
    private int counter = 0;  
    public void run() {  
        incr();  
        return;  
    }  
    public void incr() {  
        counter++;  
        return;  
    }  
    public static void main(String[] args){  
        EsempioT t1 = new EsempioT();  
        EsempioT t2 = new EsempioT();  
        t1.start();  
        t2.start();  
    }  
}
```

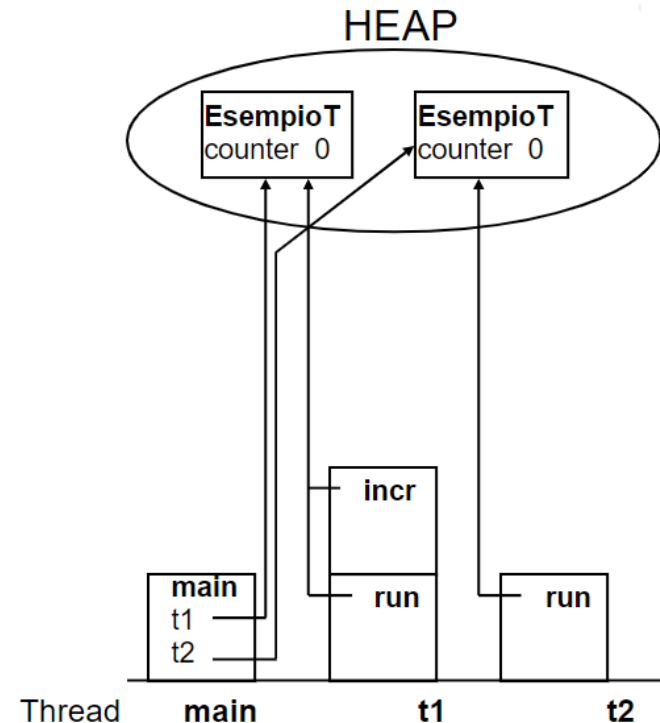
Running: Thread main





```
class EsempioT extends Thread {  
    private int counter = 0;  
    public void run() {  
        incr();  
        return;  
    }  
    public void incr() {  
        counter++;  
        return;  
    }  
    public static void main(String[] args){  
        EsempioT t1 = new EsempioT();  
        EsempioT t2 = new EsempioT();  
        t1.start();  
        t2.start();  
    }  
}
```

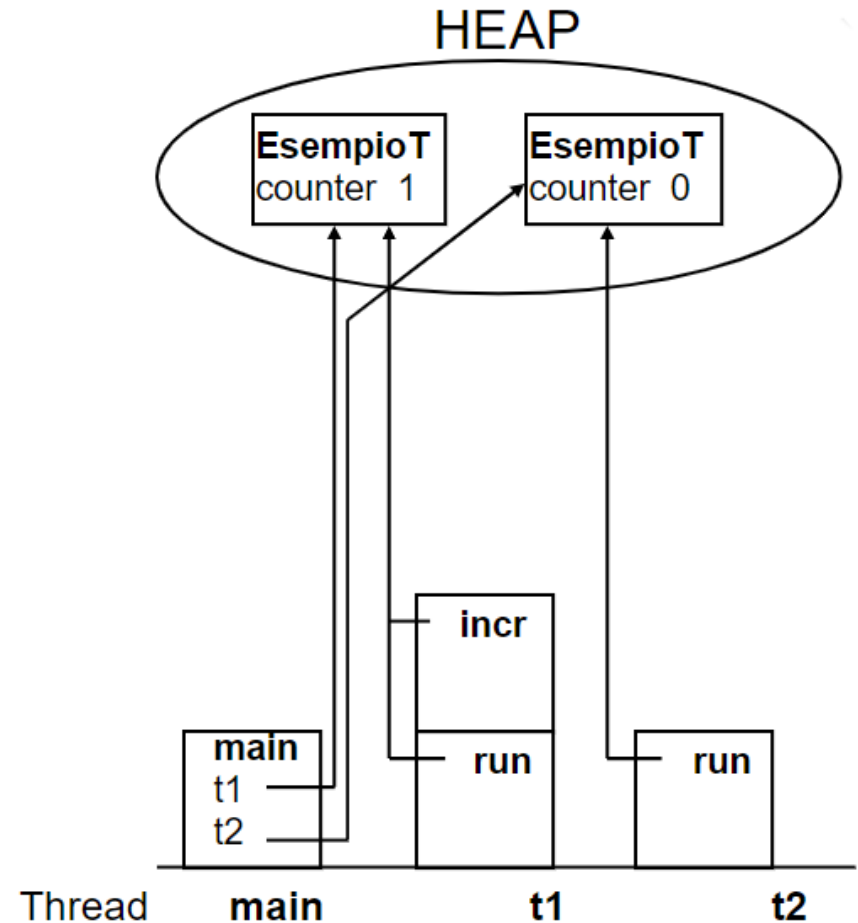
Running: Thread t1





```
class EsempioT extends Thread {  
    private int counter = 0;  
    public void run() {  
        incr();  
        return;  
    }  
    public void incr() {  
        counter++;  
        return;  
    }  
    public static void main(String[] args){  
        EsempioT t1 = new EsempioT();  
        EsempioT t2 = new EsempioT();  
        t1.start();  
        t2.start();  
    }  
}
```

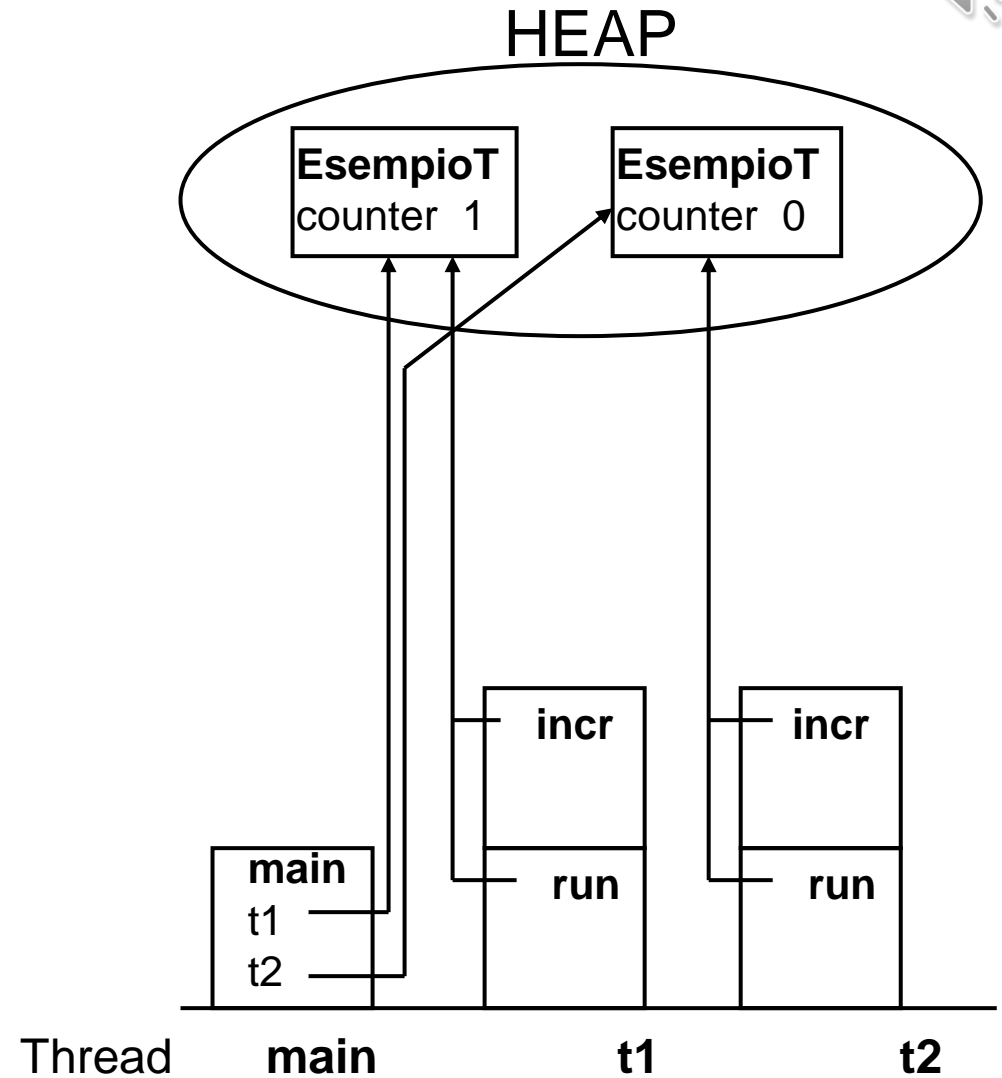
Running: Thread t1





```
class EsempioT extends Thread {  
    private int counter = 0;  
    public void run() {  
        incr();  
        return;  
    }  
    public void incr() {  
        counter++;  
        return;  
    }  
    public static void main(String[] args){  
        EsempioT t1 = new EsempioT();  
        EsempioT t2 = new EsempioT();  
        t1.start();  
        t2.start();  
    }  
}
```

Running: Thread t2





Naturalmente la computazione dei vari Thread può procedere con ordini diversi. Ad esempio, il Thread del **main()** può terminare prima dei Thread **t1** e **t2**, oppure viceversa.

Per avere la traccia di tutti gli stack in un qualunque punto della computazione si può usare il metodo statico **getAllStackTraces()** della classe **Thread**, che restituisce un **Map<Thread, StackTraceElement[]>**.

Ad ogni Thread attivo al momento della chiamata, viene associato un array di **StackTraceElement**.

EsempioT – visualizzazione degli stack dei Thread della Java Virtual Machine

[illegible]

Terminazione di Thread (metodi join())



Un Thread può chiamare il metodo **join()** su un altro Thread **t** per aspettare che **t** sia terminato prima di continuare la propria esecuzione.

Se un Thread chiama **t.join()** su un altro Thread **t**, il Thread chiamante viene sospeso finché **t** termina l'esecuzione del proprio metodo **run()**.

NB: il metodo **join()** va invocato DOPO aver fatto partire il Thread da attendere, altrimenti non è bloccante per chi lo invoca. Es:

```
Thread t1 = new MyThread();  
t1.start();  
try {t1.join();}  
catch (InterruptedException e)  
    {System.out.println(e.getMessage());}  
//... Istruzioni da eseguire dopo la terminazione di t1
```

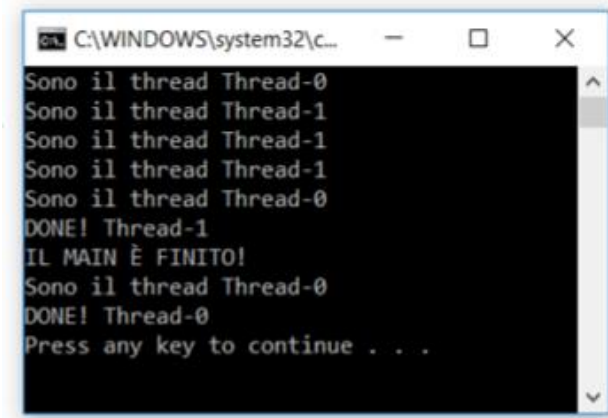
```

public class JoinApp {

    public static void main (String[] args) {
        Thread t1 = new MyThread(800);
        /* Se metto qui la join il main termina prima di t1 */
        /* try {
            t1.join();
        } catch (InterruptedException e) {System.out.println(e.getMessage());} */

        t1.start();
        Thread t2 = new MyThread(300);
        t2.start();
        try {
            t2.join();
        } catch (InterruptedException e) {System.out.println(e.getMessage());}
        System.out.println("IL MAIN È FINITO!");
    }
}

```



```

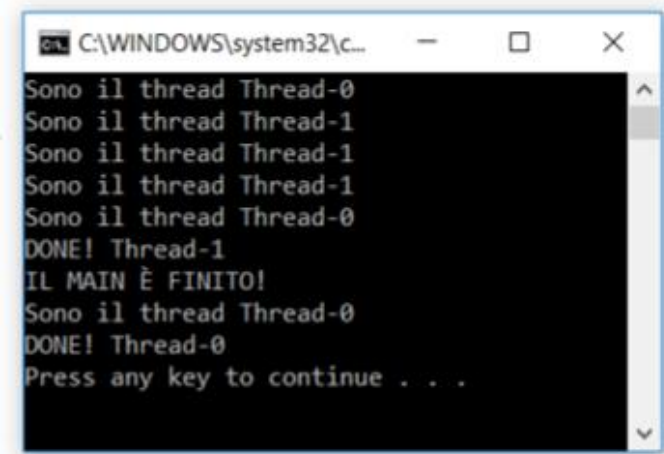
C:\WINDOWS\system32\c...
Sono il thread Thread-0
Sono il thread Thread-1
Sono il thread Thread-1
Sono il thread Thread-1
Sono il thread Thread-0
DONE! Thread-1
IL MAIN È FINITO!
Sono il thread Thread-0
DONE! Thread-0
Press any key to continue . . .

```



```
class MyThread extends Thread {  
    private int num;
```

```
    public MyThread(int num) {  
        super();  
        this.num = num;  
    }  
    public void run() {  
        for (int i = 1; i < 4; i++) {  
            System.out.println("Sono il thread " + getName());  
            try {  
                sleep(num);  
            } catch (InterruptedException e) {return;}  
        }  
        System.out.println("DONE! " + getName());  
    }  
}
```



```
C:\WINDOWS\system32\c...  
Sono il thread Thread-0  
Sono il thread Thread-1  
Sono il thread Thread-1  
Sono il thread Thread-1  
Sono il thread Thread-0  
DONE! Thread-1  
IL MAIN È FINITO!  
Sono il thread Thread-0  
DONE! Thread-0  
Press any key to continue . . .
```



Thread Demoni



L'esecuzione di un programma termina quando sono terminati *tutti i Thread* attivati. Un Thread può essere dichiarato come "**daemon**". Un *daemon* è un Thread normale, che però non influenza la terminazione del programma.

Un programma termina quando sono terminati *tutti i Thread non-daemon*. Se ci sono dei *daemon* attivi, la loro esecuzione viene terminata.

Per rendere un Thread Demone invocare il metodo `setDaemon(true)` prima del metodo `start()`. Es:

```
Class MyDaemon extends Thread {  
    public MyDaemon() {setDaemon(true);}  
    public void run() {...}  
}
```

Ringraziamenti



Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.