Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

Programmazione Concorrente e Distribuita
Thread e MultiThread

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it

Differenze tra programma e processo

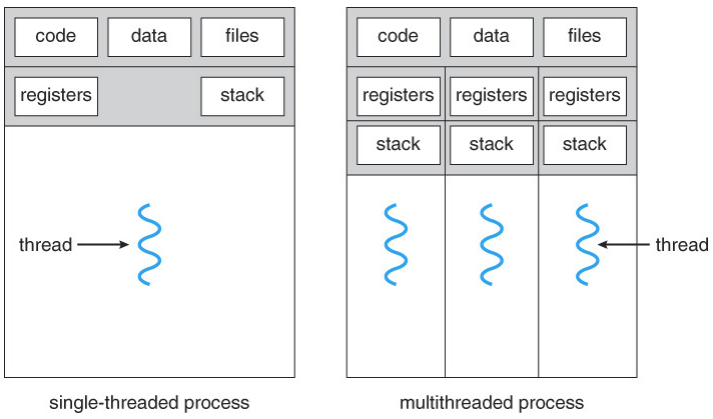
- Un programma è semplicemente un insieme di istruzioni di alto livello o istruzioni in linguaggio macchina
- Un processo è un programma in esecuzione.

Differenze tra processo e thread

- Quando i processi condividono lo stesso spazio degli indirizzi, allora vengono chiamati processi leggeri o thread.
- Quando i processi hanno il proprio spazio degli indirizzi, allora vengono chiamati processi pesanti o semplicemente processi.

Differenze tra processo e thread

- In Java i thread creano dei flussi di esecuzione concorrente all'interno del singolo processo rappresentato dal programma in esecuzione.



Il thread main

- In Java ogni programma in esecuzione è un thread
- Il metodo **`main()`** è associato al thread main
- Per poter accedere alle proprietà del thread main è necessario ottenerne un riferimento tramite il metodo **`currentThread()`**

```
public class ThreadMain {
  public static void main(String args []) {
    Thread t = Thread.currentThread( );
    System.out.println("Thread corrente: " + t );
    t.setName("Mio Thread");
    System.out.println("Dopo cambio nome: " + t );
  }
}
```

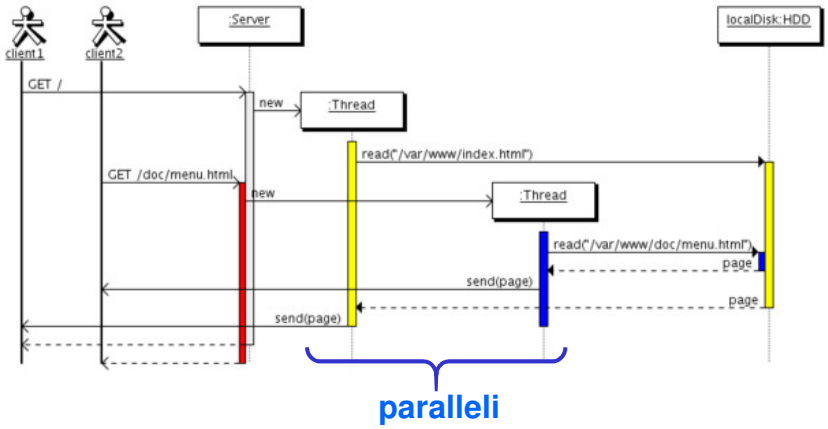
Thread[Nome Thread, Priorità, Gruppo di appartenenza del thread]

Output:

```
Thread corrente: Thread[main,5,main]
Dopo cambio nome: Thread[Mio Thread,5,main]
```

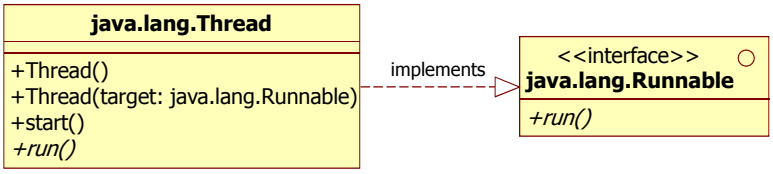

Programmazione concorrente

- Con il termine programmazione concorrente si indica la pratica di implementare dei programmi che contengano più flussi di esecuzione (Threads)



La classe principale per i Thread in Java

- La classe **`java.lang.Thread`**



- NB: il metodo **`run()`** è vuoto, sia nella classe che nell'interfaccia!

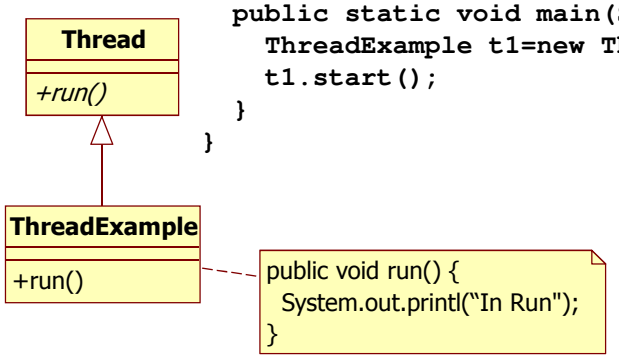
La classe principale per i Thread in Java

Il modo più semplice per creare un Thread è

1. Estendere la classe **`java.lang.Thread`** (che contiene un metodo **`run()`** vuoto)
2. Riscrivere (ridefinire, override) il metodo **`run()`** nella sottoclasse
   - Il codice eseguito dal thread è incluso nel metodo metodo **`run()`** e nei metodi invocati direttamente o indirettamente da **`run()`**
   - Questo è il codice che verrà eseguito in parallelo a quello degli altri thread
3. Creare un'istanza della sottoclasse
4. Richiamare il metodo **`start()`** su questa istanza
   - NB: spesso si mette **`start()`** nel costruttore: in tal modo creare l'istanza della sottoclasse fa anche partire il thread

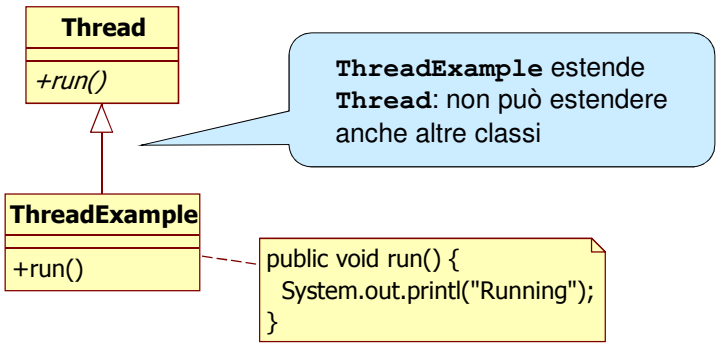
Estensione di Thread

```
public class ThreadExample extends Thread {
  public void run() {
    System.out.println("In run");
  }
  public static void main(String arg[]){
    ThreadExample t1=new ThreadExample();
    t1.start();
  }
}
```



Limitazione dell'estensione di `Thread`

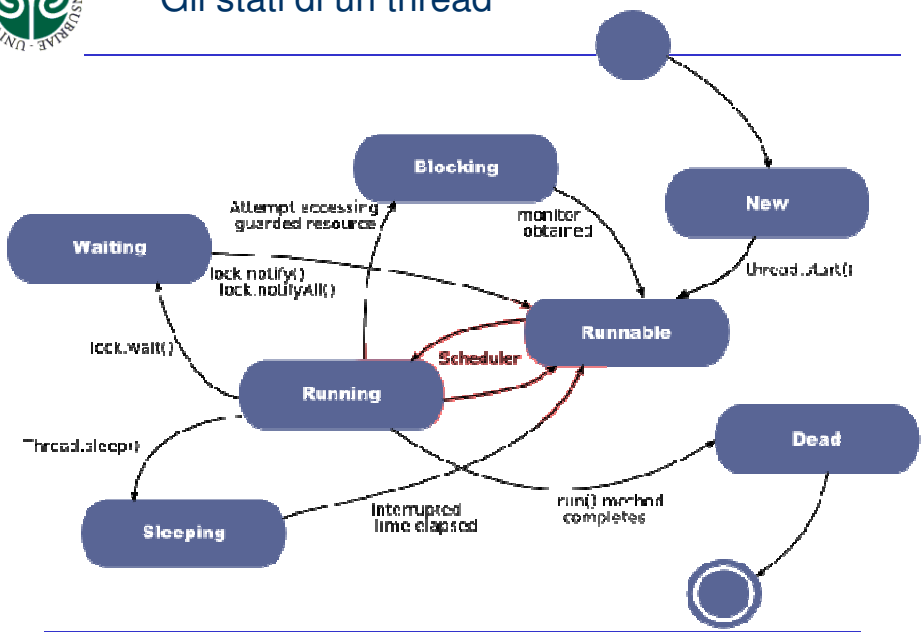
- Inconveniente:
  - Le classi che estendono **`Thread`** non possono estendere altre classi



Il metodo `run()`

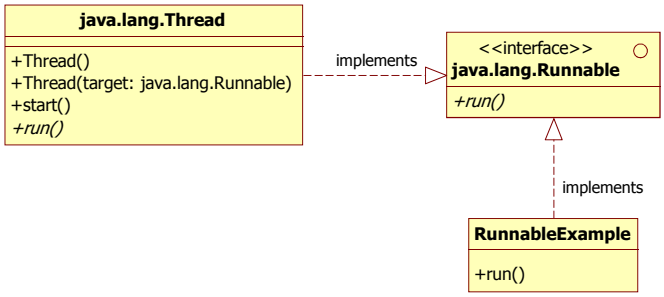
- Il metodo **`run()`** costituisce l'entry point del thread:
  - Un thread è considerato **alive** finché il metodo **`run()`** non ritorna
  - Quando **`run()`** ritorna, il thread è considerato **dead**
- Una volta che un thread è "morto" non può essere rieseguito (pena un'eccezione **`IllegalThreadStateException`**): se ne deve creare una nuova istanza.
- Non si può far partire lo stesso thread (la stessa istanza) più volte

Gli stati di un thread



Approccio alternativo alla creazione di un thread

- Si possono creare thread usando l'Interfaccia **`java.lang.Runnable`**
  1. Definire una implementazione di Runnable
  2. Realizzare il metodo **`run()`** nella classe creata
  3. Creare un'istanza di questa classe
  4. Instanziare un nuovo **`Thread`**, passando al costruttore l'istanza della classe che implementa **`Runnable`**
  5. Richiamare il metodo **`start()`** sull'istanza di **`Thread`**



Esempio usando Runnable

Questo perché l'oggetto Thread si aspetta di essere in grado di chiamare il metodo **`run()`** su questo oggetto quando il suo metodo **`start()`** viene chiamato

```
public class RunnableExample implements Runnable{
  public void run() {
    System.out.println("In run");
  }
  public static void main(String arg[]){
    RunnableExample re=new RunnableExample();
    Thread t1=new Thread(re);
    t1.start();
  }
}
```

This 'registers' the RunnableExample object re with the thread object t1

Programmi concorrenti e sequenziali

- I programmi concorrenti hanno delle proprietà molto diverse rispetto ai più comuni programmi sequenziali con i quali i programmatori hanno maggiore familiarità.
- Esempio
  - Un programma sequenziale eseguito ripetutamente con lo stesso input produce lo stesso risultato ogni volta
    - eventuali bug saranno riproducibili
  - Lo stesso non vale per i programmi concorrenti, in cui il comportamento di un thread dipende fortemente dagli altri thread.

Programma Sequenziale e Concorrente

```
public class ProceduralExample {
  public void run() {
    System.out.println("In run");
  }
  public static void main(String arg[]){
    ProceduralExample pe=new ProceduralExample();
    pe.run();
  }
}
public class RunnableExample implements Runnable{
  public void run() {
    System.out.println("In run");
  }
  public static void main(String arg[]){
    RunnableExample re=new RunnableExample();
    Thread t1=new Thread(re);
    t1.start();
  }
}
```

Run e start

- Il metodo **`run()`** può essere chiamato direttamente più volte

```
public class Example {
  public void run() {
    System.out.println("Ciao!");
  }
  public static void main(String arg[]){
    lExample e=new Example();
    e.run();
    e.run();
  }
}
```

Output:
`Ciao!`
`Ciao!`

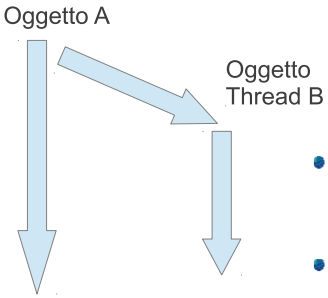
Run e start

- Il metodo **`start()`** può essere chiamato solo una volta.
- Una seconda chiamata genera l'eccezione **`IllegalThreadStateException`**

```
public class RunnableExample implements Runnable{
        public void run() {
                System.out.print(" Ciao!\n");
        }
        public static void main(String[] args) {
          RunnableExample re=new RunnableExample();
          Thread t = new Thread(re);
          t.start();
          t.start();
        }
}
```

genera l'eccezione **`IllegalThreadStateException`**

Flusso di controllo



- Da qui in poi ci sono due flussi di esecuzione, l'oggetto A non aspetta che termini l'esecuzione dell'oggetto B
- Il programma termina quando tutti i suoi thread (NON-daemon) terminano.
  - Se ci sono thread non demoni in esecuzione, il programma non termina
- Un thread daemon fornisce un servizio generale e non essenziale in background mentre il programma esegue altre operazioni.

Thread daemon

- I thread daemon di Java sono un particolare tipo di thread con le seguenti caratteristiche:
  - priorità molto bassa
    - eseguiti quando nessun altro thread dello stesso programma è in esecuzione
  - Normalmente utilizzati come fornitori di servizi per i thread normali.
    - Esempio tipico: Java garbage collector.
- JVM termina il programma terminando i thread daemon, quando questi sono gli unici thread in esecuzione nel programma.

Thread daemon

- Tipicamente si creano inserendo l'istruzione **`setDeamon(true)`** nel costruttore di un thread.

```
public class DaemonThread extends Thread {
  public DaemonThread( ) {
     setDaemon (true);
  }
  public void run() {
     int count=0;
     while (true) {
        System.out.println("Hello " + count++);
        try {
          Thread.sleep(3500) ;
        } catch(InterruptedException e) {}
      }
   }
}
```

Quando il parametro è vero il thread è un daemon thread, e termina quando il main completa l'esecuzione del proprio codice. Quando il parametro è falso, il thread continua indefinitamente e il main non può terminare.

Thread daemon

```
public class Example {
        public static void main(String[] args) {
           DaemonThread dt = new DaemonThread();
           dt.start();
           try {
                 Thread.sleep(7500);
           } catch(InterruptedException e) {}
           System.out.println("Main thread ends.\n");
        }
}
```

Output quando il thread è deamon:

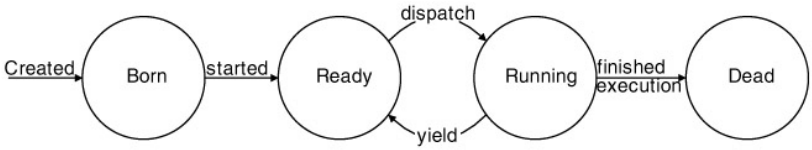
```
Hello 0
Hello 1
Hello 2
Main thread ends.
[il programma ha terminato]
```

Output quando il thread **non** è deamon:

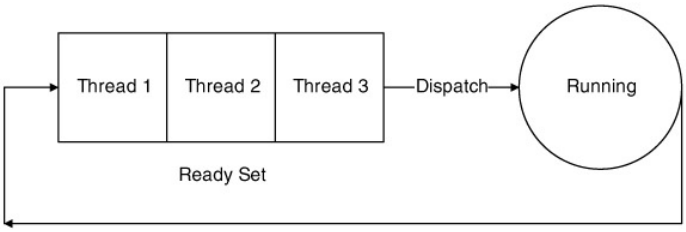
```
Hello 0
Hello 1
Hello 2
Main thread ends.

Hello 3
Hello 4
Hello 5
...
```

Stati in cui può trovarsi un Thread (versione semplificata)



- Quando invochiamo il metodo **`start()`** su un thread, il thread non viene eseguito immediatamente, ma si porta nello stato di Ready.
- Quando lo **scheduler** lo seleziona passa allo stato Running, ed esegue il metodo **`run()`** (la prima volta dall'inizio, poi da dov'era rimasto).



JAVA Thread Scheduling

- Come funziona esattamente lo scheduler dipende dalla specifica piattaforma in cui viene eseguita la VM. In generale
  - La JVM schedula l'esecuzione dei thread utilizzando un algoritmo di scheduling preemptive e priority based
  - Tutti i thread Java hanno una priorità e il thread con la priorità più alta tra quelli ready viene schedulato per essere eseguito.
  - Con il diritto di preemption lo scheduler può sottrarre la CPU al processo che la sta usando per assegnarla ad un altro processo

La politica dello Scheduler

- Java non precisa quale tipo di politica debba essere adottata dalla macchina virtuale
  - Dipende dal Sistema Operativo
- Facciamo un piccolo test per verificare se la politica è preemptive
  - Creiamo due thread
    a) Uno che procede indefinitamente, senza fare I/O o chiamate di sistema
    b) Uno che fa output
  - Facciamo partire prima il thread a): se il sistema non è preemptive, questo non cederà mai la CPU e non vedremo mai le uscite del thread b).

Test: preemptive?

```
public class BusyThread extends Thread{
  public void run() {
    int a=0;
    while(true){
      if(a>100){ a=a+1; }
      else { a=a-1; }
    }
  }
}
public class MyThread extends Thread{
  public void run() {
    String str=Thread.currentThread().getName();
    while(true){
      System.out.println(str);
    }
  }
}
```

Test: preemptive?

```
public class Esempio {
  public static void main(String arg[]) {
    System.out.println("Main: inizio");
    Thread t1 = new BusyThread();
    t1.start();
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
       e.printStackTrace();
    }
    Thread t2 = new MyThread();
    t2.setName("ciao");
    t2.start();
  }
}
```

Dorme un po'per accertarsi che t1 vada in esecuzione

La politica dello Scheduler

- I due thread hanno la stessa priorità.
- In presenza di scheduling non preemptive eseguirà solo il thread lanciato per primo che non fa I/O.
- Se vanno entrambi, lo scheduling è certamente preemptive (caso Windows e Linux).
  - Quando scade il quanto di tempo del thread la CPU gli viene forzosamente sottratta, e passata all'altro thread.

Programma procedurale con due chiamate a `run()`

```
public class Procedural {
  private int myNum;
  public Procedural(int myNum) {
    this.myNum = myNum;
  }
  public static void main(String argv[]) {
    Procedural a = new Procedural(1);
    Procedural b = new Procedural(2);
    a.run();
    b.run();
    try {
      Thread.sleep((int)(Math.random() * 100));
      System.out.println("in main");
      Thread.sleep((int)(Math.random() * 100));
      System.out.println("in main");
    } catch(InterruptedException e) { }
  }
  public void run() {
    try {
      Thread.sleep((int)(Math.random() * 100));
      System.out.println("in run, myNum = "+ myNum);
      Thread.sleep((int)(Math.random() * 100));
      System.out.println("in run, myNum = "+ myNum);
    } catch(InterruptedException e) { }
  }}
```

Out prodotto (**sempre**):

```
1  in run, myNum = 1
2  in run, myNum = 1
3  in run, myNum = 2
4  in run, myNum = 2
5  in main
6  in main
```

A questo punto il programma è terminato.

Programma concorrente con due Thread

```
public class Concurrent extends Thread {
  private int myNum ;
  public Concurrent (int myNum) {  this.myNum = myNum ;  }
  public static void main (String argv[]) {
    Concurrent a = new Concurrent(1);
    Concurrent b = new Concurrent(2);
    Thread t1 = new Thread(a);
    Thread t2 = new Thread(b);
    t1.start();
    t2.start();
    try { Thread.sleep(100*(int) Math.random());  System.out.println("in main ");
          Thread.sleep(100*(int) Math.random()); System.out.println("in main ");
    } catch (InterruptedException e) { }
  }
  public void run ( ) {
    try { Thread.sleep(100*(int) Math.random());  System.out.println("in run "+ myNum);
          Thread.sleep(100*(int) Math.random()); System.out.println("in run "+ myNum);
    } catch (InterruptedException e) { }
  }
}
```

Una possibile sequenza di esecuzione

Stato main	Stato t1	Stato t2	Istruz.	output
esec	new →ready	new	t1.start()	
esec	ready	new →ready	t2.start()	
esec	ready	ready	stampa	in main
esec → sleep	ready	ready→esec	sleep	
sleep	ready	esec	stampa	in run 2
sleep	ready→esec	esec → sleep	sleep	
sleep	esec	sleep	stampa	in run 1
sleep	esec → sleep	sleep	sleep	
sleep	sleep	sleep		
sleep	sleep →ready →esec	sleep		
sleep	esec	sleep	stampa	in run 1
sleep	esec	sleep →ready		

Una possibile sequenza di esecuzione (cont.)

Stato main	Stato t1	Stato t2	Istruz.	output
sleep	esec → sleep	ready	sleep	
sleep	sleep	ready→esec		
sleep	sleep	esec	stampa	in run 2
sleep →ready	sleep	esec		
ready	sleep	esec → sleep	sleep	
ready →esec	sleep→ready	sleep		
esec	ready	sleep	stampa	in main
esec→dead	ready	sleep	*fine*	
dead	ready→esec	sleep		
dead	esec	sleep →ready		
dead	esec→dead	ready→esec	*fine*	
dead	dead	esec→dead	*fine*	

Un'altra possibile sequenza di esecuzione

- Un'altra possibile sequenza di esecuzione genera il seguente output:

```
in main
in main
in run 2
in run 1
in run 2
in run 1
```

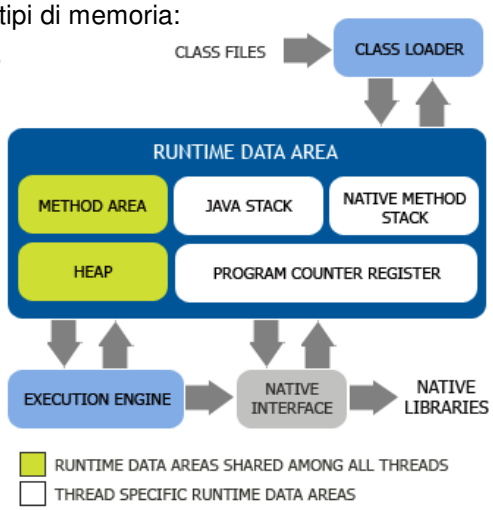
- Esercizio: dedurre la sequenza di eventi che ha determinato l'output.
- Esercizio: rimuovere le istruzioni sleep e vedere che succede.

Modello di Esecuzione Semplificato

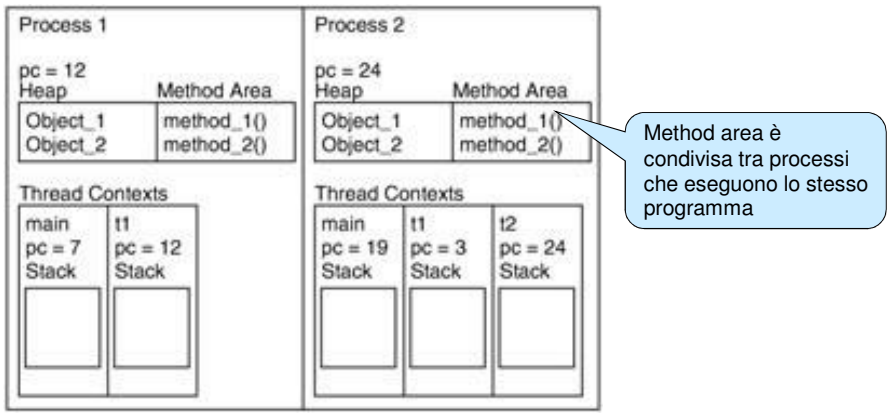
- Come un programma viene istanziato ed eseguito su un computer è un argomento complesso
- Per gli scopi di questo corso, il comportamento di base di programmi concorrenti può essere spiegato usando un'architettura di computer semplificata
- Svilupperemo un modello per spiegare il comportamento dei programmi concorrenti

Modello di Memoria Semplificato (SMM)

- Il modello di memoria semplificato utilizzato dalla Macchina Virtuale Semplificata (SVM) utilizza diversi tipi di memoria:
  - heap: utilizzato per memorizzare tutti gli oggetti e i loro dati
  - method area: contiene le definizioni delle classi e le istruzioni compilate
  - program context: informazioni uniche per ogni thread, come le stack e il program counter (PC)



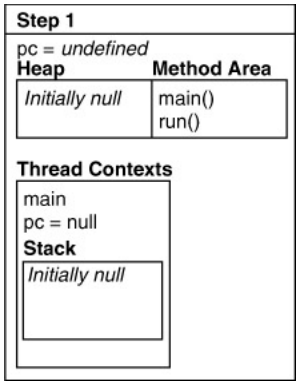
Modello di Memoria Semplificato



- La figura mostra la SMM di un computer che esegue due processi,
  - il primo con thread main e t1
  - il secondo con thread main, t1 e t2

SMM durante l'esecuzione di un programma single thread

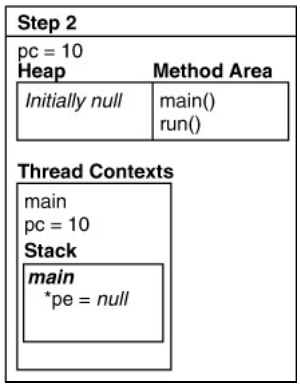
```
1  public class PExec {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    PExec pe = new PExec ( ) ;
11    pe.run();
12    return;
13   }
14  }
```



- La SVM crea lo heap e il thread context.
- Crea anche la method area e ci carica i metodi per la classe Pexec.
- Infine, crea il PC con valore per il momento non definito.

SMM durante l'esecuzione di un programma single thread

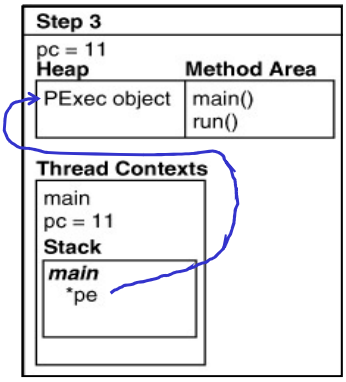
```
1  public class PExec {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    PExec pe = new PExec ( ) ;
11    pe.run();
12    return;
13   }
14  }
```



- La SVM inserisce un activation record per il metodo main nello stack.
  - Contenente un riferimento all'oggetto PExec.
- La SVM assegna la linea 10 al process PC e al thread PC (la prima riga eseguibile nel metodo main)

SMM durante l'esecuzione di un programma single thread

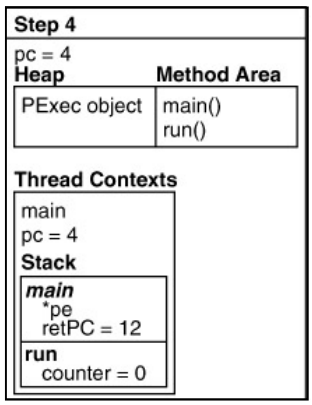
```
1  public class PExec {
2    public void run ( ) {
3      int counter = 0;
4      System.out.println("In run, counter= " + counter);
5      counter++;
6      System.out.println("In run, counter= " + counter);
7      return;
8    }
9    public static void main (String args[]) {
10     PExec pe = new PExec ( ) ;
11     pe.run();
12     return;
13   }
14 }
```



- La SVM esegue la riga 10, che crea una istanza della classe PExec nello heap.
- Quindi l'SVM aggiorna il PC, che indica la riga seguente, cioè la 11.

SMM durante l'esecuzione di un programma single thread

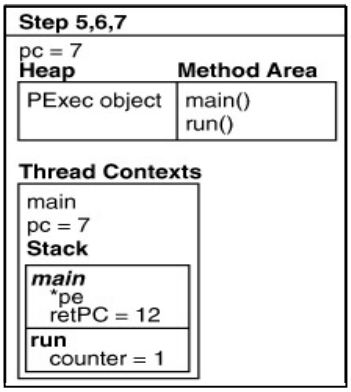
```
1  public class PExec {
2    public void run ( ) {
3      int counter = 0;
4      System.out.println("In run, counter= " + counter);
5      counter++;
6      System.out.println("In run, counter= " + counter);
7      return;
8    }
9    public static void main (String args[]) {
10     PExec pe = new PExec ( ) ;
11     pe.run();
12     return;
13   }
14 }
```



- La SVM chiama il metodo run.
  - Il PC della riga successiva viene memorizzato sulla pila in retPC.
- La SVM crea quindi un nuovo activation record per il metodo run, contenente la variabile locale counter, già inizializzata a zero.
- Infine, aggiorna il PC, che diventa 4.

SMM durante l'esecuzione di un programma single thread

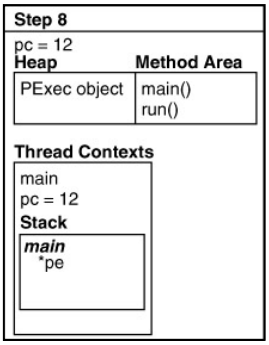
```
1  public class PExec {
2    public void run ( ) {
3      int counter = 0;
4      System.out.println("In run, counter= " + counter);
5      counter++;
6      System.out.println("In run, counter= " + counter);
7      return;
8    }
9    public static void main (String args[]) {
10     PExec pe = new PExec ( ) ;
11     pe.run();
12     return;
13   }
14 }
```



- Negli step 5, 6 e 7, la SVM esegue le righe da 4 a 6 comprese
  - stampa la variabile counter, l'incrementa e la stampa nuovamente.
- Alla fine il PC punta alla linea 7, e counter vale 1.

SMM durante l'esecuzione di un programma single thread

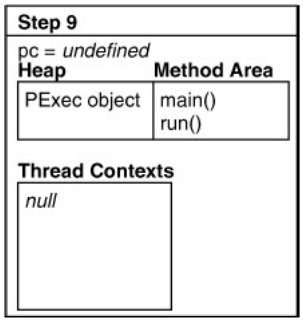
```
1  public class PExec {
2    public void run ( ) {
3      int counter = 0;
4      System.out.println("In run, counter= " + counter);
5      counter++;
6      System.out.println("In run, counter= " + counter);
7      return;
8    }
9    public static void main (String args[]) {
10     PExec pe = new PExec ( ) ;
11     pe.run();
12     return;
13   }
14 }
```



- La SVM esegue il return. Questo causa l'eliminazione dell'activation record del metodo run dallo stack, e al PC viene assegnato il valore di retPC.

SMM durante l'esecuzione di un programma single thread

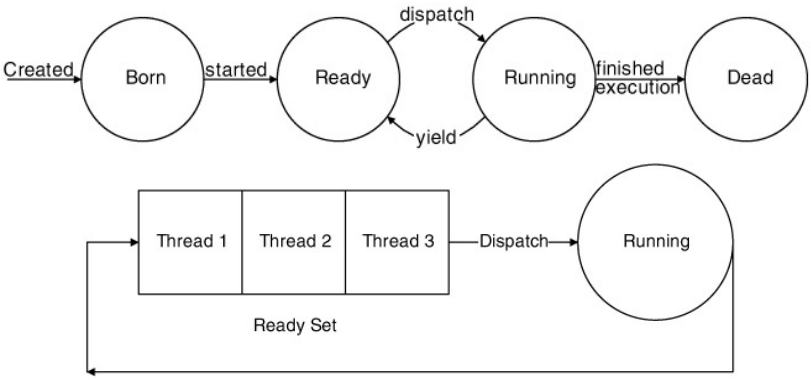
```
1  public class PExec {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    PExec pe = new PExec ( ) ;
11    pe.run();
12    return;
13   }
14  }
```



- Il metodo main esegue il return, e anche l'activation record del metodo main viene eliminato dallo stack.
  - Il riferimento all'oggetto PExec viene cancellato, ma l'oggetto stesso no: verrà eliminato in seguito dal garbage collection.

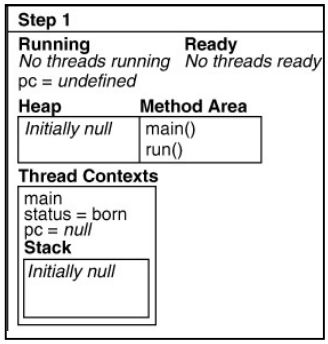
SVM Thread States

- Poiché un programma concorrente ha più thread, ognuno con il suo contesto e PC, la SVM deve scegliere quale thread mandare in esecuzione.
- La decisione su quale Thread mandare in esecuzione dipende da molti fattori. . .



SMM durante l'esecuzione di un programma multithread

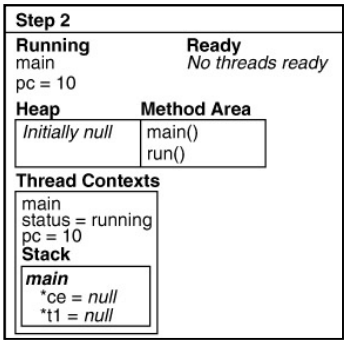
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    CExec ce = new CExec ( ) ;
11    Thread t1 = new Thread(ce);
12    t1.start();
13    System.out.println("In main");
14    return;
15   }
16  }
```



- La SVM crea lo heap e il thread context.
- Crea anche la method area e ci carica i metodi per la classe CExec.
- Infine, crea il PC, con valore per il momento non definito

SMM durante l'esecuzione di un programma multithread

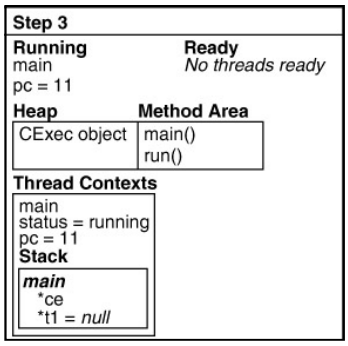
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    CExec ce = new CExec ( ) ;
11    Thread t1 = new Thread(ce);
12    t1.start();
13    System.out.println("In main");
14    return;
15   }
16  }
```



- La SVM inserisce un activation record per il metodo main nello stack, contenente un riferimento per l'oggetto ce e uno per il thread t1
- La SVM assegna la linea 10 al process PC e al thread PC (la prima riga eseguibile nel metodo main)

SMM durante l'esecuzione di un programma multithread

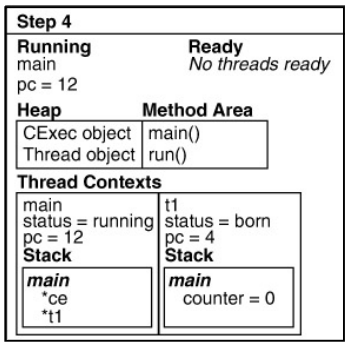
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    CExec ce = new CExec ( ) ;
11    Thread t1 = new Thread(ce);
12    t1.start();
13    System.out.println("In main");
14    return;
15   }
16  }
```



- La SVM esegue la riga 10, che crea una istanza della classe PExec nello heap (il riferimento nell'area di attivazione viene aggiornato)
- Il PC diventa 11

SMM durante l'esecuzione di un programma multithread

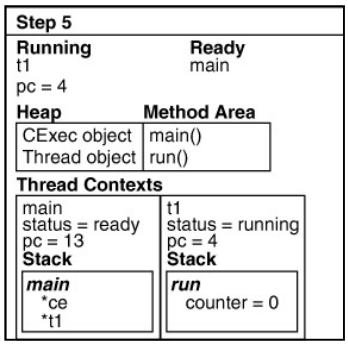
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    CExec ce = new CExec ( ) ;
11    Thread t1 = new Thread(ce);
12    t1.start();
13    System.out.println("In main");
14    return;
15   }
16  }
```



- La SVM esegue la linea 11, creando
  - un'istanza del Thread nello heap (e aggiornando il riferimento nel record di attivazione)
  - un nuovo context per il thread (status = born), completo di record di attivazione per il metodo run.

SMM durante l'esecuzione di un programma multithread

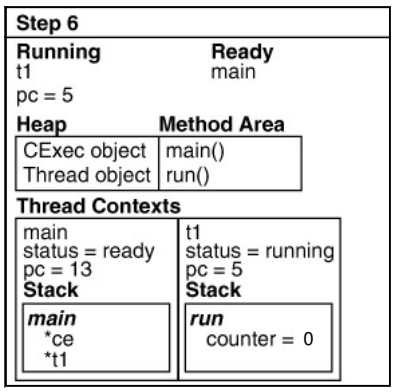
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10   CExec ce = new CExec ( ) ;
11   Thread t1 = new Thread(ce);
12   t1.start();
13   System.out.println("In main");
14   return;
15  }
16 }
```



- La SVM esegue la linea 12: il thread t1 diventa ready.
  - Quindi si possono eseguire sia main sia t1.
- Lo scheduler sceglie il thread t1, con conseguente context switch dal thread main a t1.
  - Il PC di processo assume il valore del PC del thread t1

SMM durante l'esecuzione di un programma multithread

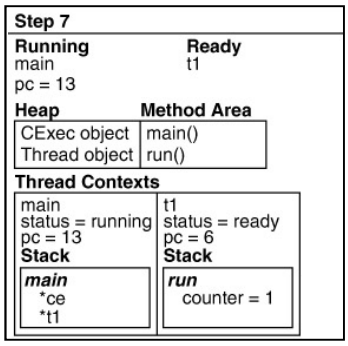
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10   CExec ce = new CExec ( ) ;
11   Thread t1 = new Thread(ce);
12   t1.start();
13   System.out.println("In main");
14   return;
15  }
16 }
```



- Viene eseguita l'istruzione 4 del thread t1.
- L'SVM deve nuovamente scegliere quale thread eseguire e sceglie ancora t1

SMM durante l'esecuzione di un programma multithread

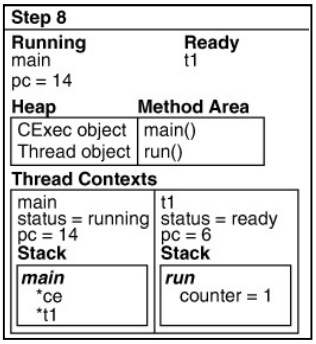
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    CExec ce = new CExec ( ) ;
11    Thread t1 = new Thread(ce);
12    t1.start();
13    System.out.println("In main");
14    return;
15   }
16 }
```



- Viene eseguita la linea 5: counter viene incrementato nel record di attivazione di run.
- La SVM adesso sceglie di eseguire il thread main ed esegue quindi un nuovo context switch.
  - Il PC di processo diventa = al PC di main, che vale 13

SMM durante l'esecuzione di un programma multithread

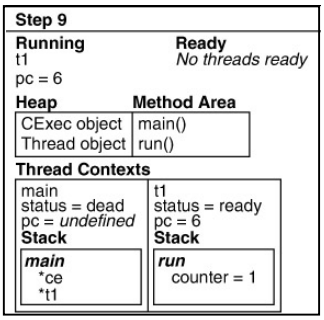
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    CExec ce = new CExec ( ) ;
11    Thread t1 = new Thread(ce);
12    t1.start();
13    System.out.println("In main");
14    return;
15   }
16 }
```



- Viene eseguita la linea 13.
- L'SVM sceglie di continuare con il thread main, aggiornando il PC alla linea 14.

SMM durante l'esecuzione di un programma multithread

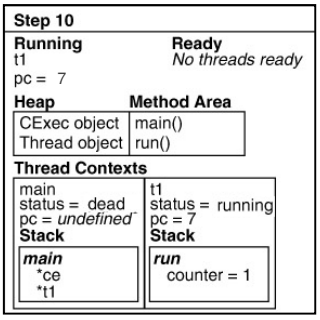
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    CExec ce = new CExec ( ) ;
11    Thread t1 = new Thread(ce);
12    t1.start();
13    System.out.println("In main");
14    return;
15   }
16 }
```



- Eseguendo la linea 14 il thread main viene completato e passa nello stato dead ma non può terminare.
  - È il parent thread di t1, e deve quindi esistere fino a quando t1 non diventa dead (questo vale in generale per tutti i figli).
- Si ritorna ad eseguire t1, che è l'unico thread ready.
  - Il process PC diventa uguale al PC di t1

SMM durante l'esecuzione di un programma multithread

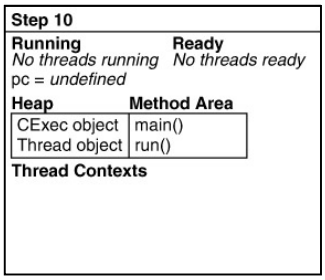
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3    int counter = 0;
4    System.out.println("In run, counter= " + counter);
5    counter++;
6    System.out.println("In run, counter= " + counter);
7    return;
8   }
9   public static void main (String args[]) {
10    CExec ce = new CExec ( ) ;
11    Thread t1 = new Thread(ce);
12    t1.start();
13    System.out.println("In main");
14    return;
15  }
16
```



- Viene eseguita la linea 6 del thread t1.
- La SVM continua ad eseguire il thread t1, aggiornando il PC.

SMM durante l'esecuzione di un programma multithread

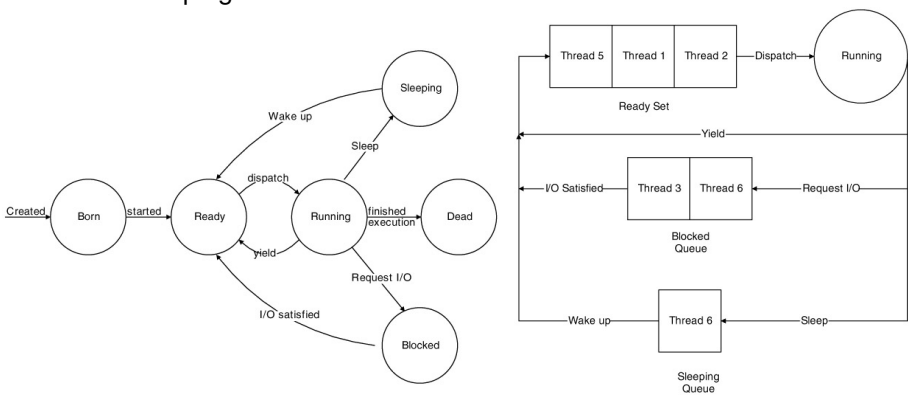
```
1  public class CExec implements Runnable {
2   public void run ( ) {
3     int counter = 0;
4     System.out.println("In run, counter= " + counter);
5     counter++;
6     System.out.println("In run, counter= " + counter);
7     return;
8   }
9   public static void main (String args[]) {
10    CExec ce = new CExec ( ) ;
11    Thread t1 = new Thread(ce);
12    t1.start();
13    System.out.println("In main");
12    return;
13  }
14 }
```



- L'esecuzione della linea 7 completa il thread t1 cambiando il suo stato a dead.
- Il thread main adesso non ha altri children, e il suo context viene cancellato. Il programma può terminare.

Sleeping and Blocking

- Il modello visto è sufficiente a spiegare un programma in cui tutti i thread sono sempre ready o running.
- Spesso un thread può essere sospeso per un certo periodo di tempo (es. **`Thread.Sleep`** o un I/O), in questo caso, il thread assume lo stato di sleeping o blocked.



Far partire i thread: `start()`

- Una chiamata `t.start()` rende il thread `t` pronto all'esecuzione.
- Il controllo ritorna al chiamante
- Prima o poi (quando lo scheduler lo riterrà opportuno) verrà invocato il metodo `run()` del thread `t`
- I due thread saranno eseguiti in modo concorrente ed indipendente

- Importante
  - L'ordine con cui ogni thread eseguirà le proprie istruzioni è noto, ma l'ordine in cui le istruzioni dei vari thread saranno eseguite effettivamente è indeterminato (nondeterminismo).

Un esempio di programma concorrente

- Che tipo di output produrrà questo esempio? E perché?

```
public class ThreadExample extends Thread {
  public void run() {
    for(int i=0; i<10; i++) {
      System.out.println("Nuovo thread");
    }
  }
  public static void main(String arg[]){
    ThreadExample t = new ThreadExample();
    t.start();
    for(int i=0; i<10; i++) {
      System.out.println("Main");
    }
  }
}
```

Un esempio di programma concorrente

- Che tipo di output produrrà questo esempio?

Main Main Main Main Main Main Main Main Main Main Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread	Se dopo t.start() esegue main Se dopo t.start() esegue il thread t	Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Nuovo thread Main Main Main Main Main Main Main Main Main Main

Sleep

- Se vogliamo essere sicuri che dopo t.start() inizi ad eseguire il task t, un modo sicuro consiste nel mandare in sleep il main:

```
public class ThreadExample extends Thread {
  public void run() {
    for(int i=0; i<10; i++) {
      System.out.println("Nuovo thread");
    }
  }
  public static void main(String arg[]){
    ThreadExample t = new ThreadExample();
    t.start();
    Thread.sleep(1);
    for(int i=0; i<10; i++) {
      System.out.println("Main");
    }
  }
}
```

Output

- Naturalmente, nulla vieta che succeda questo:

```
Nuovo thread
Main
Main
Main
Main
Main
Main
Main
Main
Main
Nuovo thread
Nuovo thread
Nuovo thread
Nuovo thread
Nuovo thread
Nuovo thread
Nuovo thread
Nuovo thread
Nuovo thread
```


Fare una pausa: `sleep()`

```
public static native void sleep (long msToSleep)
    throws InterruptedException
```

- Il metodo **`sleep()`**
  - Non utilizza cicli del processore
  - è un metodo statico e mette in pausa il thread corrente
    - non è possibile per un thread mettere in pausa un altro thread
  - mentre un thread è in sleep può essere interrotto (via **`Interrupt`**) da un altro thread in tal caso viene sollevata un'eccezione **`InterruptedException`** quindi **`sleep()`** va eseguito in un blocco **`try/catch`** (oppure il metodo che lo esegue deve dichiarare di sollevare tale eccezione)
    - **`Interrupt`** dovrebbe essere usata con **`wait`**, non con **`sleep`**.

Un esempio di programma concorrente con `sleep()`

```
public class ThreadExample extends Thread {
  public void run() {
    for(int i=0; i<10; i++) {
      System.out.println("Nuovo thread");
      try {
        Thread.sleep(300);
      } catch (InterruptedException e)  {}
    }
  }
  public static void main(String args[]) {
    ThreadExample t = new ThreadExample();
    t.start();
    for(int i=0; i<10; i++) {
      System.out.println("Main");
      try {
        Thread.sleep(300);
      } catch (InterruptedException e) { }
    }
  }
}
```

- Che tipo di output produrrà questo esempio? e perché?

Output tipico

```
Main
Nuovo thread
Nuovo thread
Main
Nuovo thread
Main
Nuovo thread
Main
Nuovo thread
Main
Nuovo thread
Main
Nuovo thread
Main
Nuovo thread
Main
Main
Nuovo thread
Nuovo thread
Main
Nuovo thread
Main
```

Fare una pausa: busy loop?

- Per rallentare l'esecuzione dei due Thread potremmo creare un metodo privato che cicla a vuoto allo scopo di perdere del tempo

```
private void busyLoop(){
   // wait for 60 seconds
   long startTime = System.currentTimeMillis();
   long stopTime = startTime + 60000;

   while (System.currentTimeMillis() < stopTime) {
     // do nothing, just loop
   }
}
```

- È una buona soluzione?
  - Generalmente no: si sprecano cicli di processore mentre si potrebbero impiegare in qualche attività utile.
  - Inoltre, se ci sono molti thread, è possibile (probabile) che nel momento in cui si raggiunge il momento in cui bisogna uscire dal ciclo, il thread non sia in esecuzione. Con la conseguenza che esce in ritardo, rispetto al momento desiderato.

Il metodo `isAlive()`

- Può essere utilizzato per testare se un thread è "vivo"
- quando viene chiamato **`start()`** il thread è considerato "alive"
- il thread è considerato "alive" finche il metodo **`run()`** non ritorna

- Usare **`isAlive()`** per "attendere" un thread?
  - Si potrebbe utilizzare **`isAlive()`** per testare periodicamente se un thread è ancora "vivo", ma non è efficiente.

- Ad esempio il thread chiamante potrebbe attendere t1:

```
while(t1.isAlive()) {
   Thread.sleep(100) ;
}
```

Esempio: Attendere la terminazione di altri Thread

```
public class ThreadSleep extends Thread {
  public ThreadSleep(String s) {  super(s); }
  public void run() {
    for (int i=0; i<5; ++i) {
      System.out.println(getName() + ": in esec.");
      try {
        Thread.sleep(200);
      }
      catch (InterruptedException e) { }
    }
    System.err.println(getName() + ": finito");
  }
}
```


Esempio: Attendere la terminazione di altri Thread

```
public class ThreadTestExit {
  public static void main(String args[])
                              throws Exception {
    System.err.println("I thread stanno per partire");
    Thread t1 = new ThreadSleep("t1"); t1.start();
    Thread t2 = new ThreadSleep("t2"); t2.start();
    Thread t3 = new ThreadSleep("t3"); t3.start();
    System.err.println("I thread sono partiti\n");
    System.err.println("chiude l'applicazione");
    System.exit(0); //chiude l'applicazione in ogni caso
  }
}
```

I thread stanno per partire
I thread sono partiti

chiude l'applicazione
t1: in esecuzione.
t2: in esecuzione.
t3: in esecuzione.

Attenzione
L'applicazione esce prima della terminazione dei suoi task

Esempio: Attendere la terminazione di altri Thread

```
public class ThreadTestExit {
  public static void main(String args[])
                              throws Exception {
    System.err.println("I thread stanno per partire");
    Thread t1 = new ThreadSleep("t1"); t1.start();
    Thread t2 = new ThreadSleep("t2"); t2.start();
    Thread t3 = new ThreadSleep("t3"); t3.start();
    System.err.println("I thread sono partiti\n");
    while (t1.isAlive() || t2.isAlive() || t3.isAlive()){
      Thread.sleep(100);
    }
    System.err.println("chiude l'applicazione");
    System.exit(0); //chiude l'applicazione in ogni caso
  }
}
```


Output

```
I thread stanno per partire
I thread sono partiti

t1: in esecuzione.
t2: in esecuzione.
t3: in esecuzione.
t1: in esecuzione.
t3: in esecuzione.
t2: in esecuzione.
t1: in esecuzione.
t2: in esecuzione.
t3: in esecuzione.
t3: in esecuzione.
t2: in esecuzione.
t1: in esecuzione.
t2: in esecuzione.
t1: in esecuzione.
t3: in esecuzione.
t2: finito
t3: finito
t1: finito
chiude l'applicazione
```

Attendere la terminazione: `join()`

- Il metodo `join()` attende la terminazione del thread sul quale è richiamato
- Il thread che esegue `join()` rimane così bloccato in attesa della terminazione dell'altro thread
- Il metodo `join()` può lanciare una `InterruptedException`

applicazione

Attendere la terminazione: `join()`

```
public class ThreadTestExitErr {
  public static void main(String args[])
                                  throws Exception {
    System.err.println("I thread stanno per partire");
    Thread t1 = new ThreadSleep("t1"); t1.start();
    Thread t2 = new ThreadSleep("t2"); t2.start();
    Thread t3 = new ThreadSleep("t3"); t3.start();
    System.err.println("I thread sono partiti\n");
    t1.join();
    t2.join();
    t3.join();
    System.err.println("chiude l'applicazione");
    System.exit(0); //chiude l'applicazione
  }
}
```

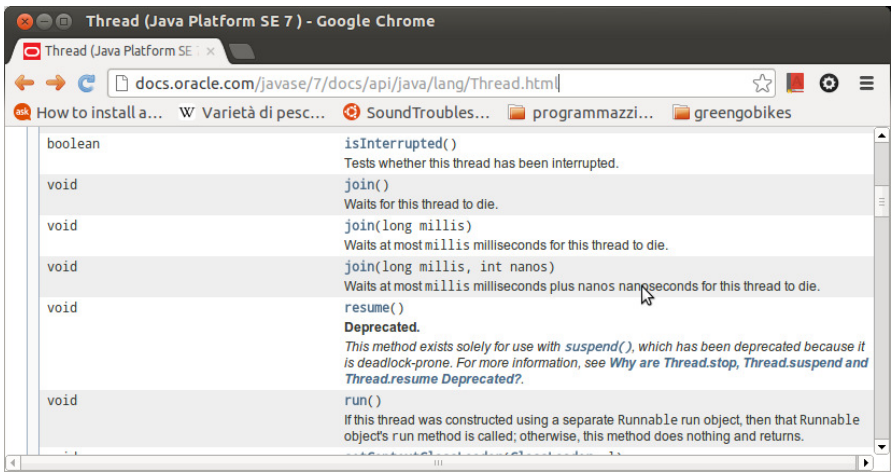
Output

```
I thread stanno per partire
I thread sono partiti

t1: in esecuzione.
t2: in esecuzione.
t3: in esecuzione.
t1: in esecuzione.
t2: in esecuzione.
t3: in esecuzione.
t1: in esecuzione.
t2: in esecuzione.
t3: in esecuzione.
t1: in esecuzione.
t2: in esecuzione.
t3: in esecuzione.
t1: in esecuzione.
t2: in esecuzione.
t3: in esecuzione.
t1: finito
t3: finito
t2: finito
chiude l'applicazione
```


Attendere la terminazione: `join()`

- Il metodo `join()` può essere chiamato con diversi parametri.
- Vedi JavaDoc...



How to Stop a Thread?

- Java è stato concepito fin dall'inizio come un linguaggio multi-thread. Purtroppo, i progettisti di Java non sono riusciti nel loro tentativo di fornire un mezzo sicuro ed efficace per fermare un thread dopo il suo avvio.
- La classe **`java.lang.Thread`** originale include i metodi, **`start()`**, **`stop()`**, **`stop(Throwable)`**, **`suspend()`**, **`destroy()`** e **`resume()`**, che avevano lo scopo di fornire le funzionalità di base per l'avvio e l'arresto di un thread.
- Di questi solo il metodo **`start()`** non è stato deprecato.

- Come fermare un thread è una questione ricorrente per i programmatori Java
  - Con il rilascio di Java V5.0 (o V1.5), la risposta definitiva è "un thread si interrompe utilizzando **`interrupt()`**".

Terminare un thread: **`interrupt()`**

- Il metodo **`interrupt()`** setta un flag di interruzione nel thread di destinazione e ritorna.
- Il thread che riceve l'interruzione non viene effettivamente interrotto (quindi al ritorno di **`interrupt()`** non si può assumere che il thread sia stato effettivamente interrotto).
- Il thread può controllare se tale flag è settato e nel caso uscire (dal **`run()`**)
- I metodi che mettono in pausa (**`sleep`**, **`join`**, ...) un thread controllano il flag di interruzione prima e durante lo stato di pausa
- Se tale flag risulta settato, allora lanciano un eccezione **`InterruptedException`** (e resettano il flag)
- Il thread che era stato interrotto intercetta l'eccezione e "dovrebbe" terminare l'esecuzione
  - Il fatto che il thread termini o no dipende da cosa viene scritto nel codice che gestisce l'eccezione.

Esempio: Interrompere i Thread con `interrupt()`

```
public class ThreadTestExitErr {
  public static void main(String args[]) throws Exception {
    System.err.println("\nI thread stanno per partire");
    Thread t1 = new ThreadSleep("t1"); t1.start();
    Thread t2 = new ThreadSleep("t2"); t2.start();
    Thread t3 = new ThreadSleep("t3"); t3.start();
    System.err.println("I thread sono partiti\n");
    t1.interrupt();
    t2.interrupt();
    t3.interrupt();
    t1.join();
    t2.join();
    t3.join();
    System.err.println("chiude l'applicazione");
    System.exit(0); //chiude l'applicazione in ogni caso
  }
}
```

Output:

Come prima!

Perché l'eccezione **`InterruptedException`** non è gestita.

Esempio: Attendere la terminazione di altri Thread

```
public class ThreadSleep extends Thread {
        public ThreadSleep(String s) { super(s); }
        public void run() {
                for (int i=0; i<5; ++i) {
                        System.out.println(getName() + ": in esecuzione.");
                        try { Thread.sleep(200); }
                        catch (InterruptedException e) {
                                System.err.println(getName() + ": interrotto");
                                break;
                        }
                }
                System.err.println(getName() + ": finito");
        }
}
```

Gestisce l'interruzione uscendo dal loop e quindi terminando

Esempio: Interrompere i Thread con `interrupt()`

```
public class ThreadTestExitErr {
  public static void main(String args[]) throws Exception {
    System.err.println("\nI thread stanno per partire");
    Thread t1 = new ThreadSleep("t1"); t1.start();
    Thread t2 = new ThreadSleep("t2"); t2.start();
    Thread t3 = new ThreadSleep("t3"); t3.start();
    System.err.println("I thread sono partiti\n");
    t1.interrupt();
    t2.interrupt();
    t3.interrupt();
    t1.join();
    t2.join();
    t3.join();
    System.err.println("chiude l'applicazione");
    System.exit(0); //chiude l'applicazione in ogni caso
  }
}
```

Output

```
I thread stanno per partire
I thread sono partiti

t1: interrottot1: in esecuzione.
t1: finito

t2: in esecuzione.
t2: interrotto
t3: in esecuzione.t2: finito

t3: interrotto
t3: finito
chiude l'applicazione
```


Problemi con `interrupt()`

- Il metodo **`interrupt()`** non funziona se il thread "interrotto" non esegue mai metodi di attesa (sleep, join, ...)
- Ad es. se un thread si occupa di effettuare calcoli in memoria, non potrà essere interrotto in questo modo
- I thread devono cooperare ad esempio controllando periodicamente il suo stato di "interruzione":
  - **`isInterrupted()`** controlla il flag di interruzione senza resettarlo
  - **`Thread.interrupted()`** controlla il flag di interruzione del thread corrente e se settato lo resetta

```
public void run() {
  while (condizione) {
    // esegue un'operazione complessa
    if (Thread.interrupted()) {
      break;
    }
  }
}
```

Collaborazione: `Thread.yield()`

- La chiamata al metodo statico **`Thread.yield( )`** è un suggerimento per il thread scheduler che dice,

  Ho fatto una parte importante del mio lavoro e questo sarebbe un buon momento per passare a un altro thread per un po'
- Permette ad un thread di lasciare volontariamente il processore ad un altro thread
- Utile nel caso di un thread che non esegue spesso operazioni che lo mettano in attesa

Collaborazione: `Thread.yield()`

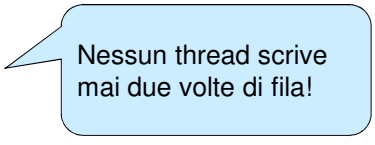
- Con **`yield()`** si cede il controllo allo scheduler, che sceglierà un altro thread (se esiste) da mandare in esecuzione al posto di quello che ha fatto **`yield()`**.
- Quando usare **`yield()`**?
  - Quando non c'è preemption.
    - Attenzione: se mi dimentico di fare **`yield()`** posso provocare starvation.
  - In casi molto particolari.
    - Ad es. quando tutto il programma è concepito in modo da rendere naturale l'uso di **`yield()`**
      - In un'organizzazione ciclica
- Ci sono alternative che possono essere preferibili in vari casi
  - se siete interessati a "usare solo parte della CPU", potete ad esempio stimare la quantità di CPU che il thread ha utilizzato nel suo ultimo blocco di elaborazione, quindi chiamare **`sleep()`** per una certa quantità di tempo per compensare

Esempio completo con `Thread.yield()`

```
public class Decollo implements Runnable {
  protected int countDown = 10;
  private static int taskCount = 0;
  private final int id = taskCount++;
  public String status () {
    return " "+id+"("+(countDown>0 ? countdown : "Decollo !")+") , ";
  }
  public void run() {
    while(countDown-- > 0) {
      System.out.println(status());
      Thread.yield();
    }
  }
}
public class YieldExample {
  public static void main(String[] args) {
    for (int i=0; i<5; i++) {
      new Thread (new Decollo()).start();
    }
    System.out.println("In attesa del decollo");
  }
}
```


Esempio completo con `Thread.yield()`

In attesa del decollo
4(9),
3(9),
4(8),
3(8),
4(7),
3(7),
4(6),
3(6),
4(5),
3(5),
4(4),
3(4),
4(3),
1(9),
2(9),
0(9),
2(8),
1(8),
...



Conclusioni

- La programmazione Thread-based in Java implica la comprensione di come creare i thread e come controllarli
- Questo aspetto non è banale, in quanto la presenza di thread produce un comportamento non deterministico nei programmi.
- La comprensione di cosa avviene in memoria durante l'esecuzione di un programma concorrente è molto importante
- Questa lezione introduttiva vi mette in grado di iniziare a creare i vostri primi programmi concorrenti

Riferimenti

- Molte delle informazioni presentate sono presenti
  - nel capitolo 2 di: Creating Components: Object Oriented, Concurrent, and Ditributed Computing in Java by Charles W. Kann
  - nel capitolo 'Concurrency' di: Thinking in Java by Bruce Eckel et. Al.