



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

Programmazione Concorrente e Distribuita Thread e MultiThread parte B

Luigi Lavazza

Dipartimento di Scienze Teoriche e Applicate

luigi.lavazza@uninsubria.it



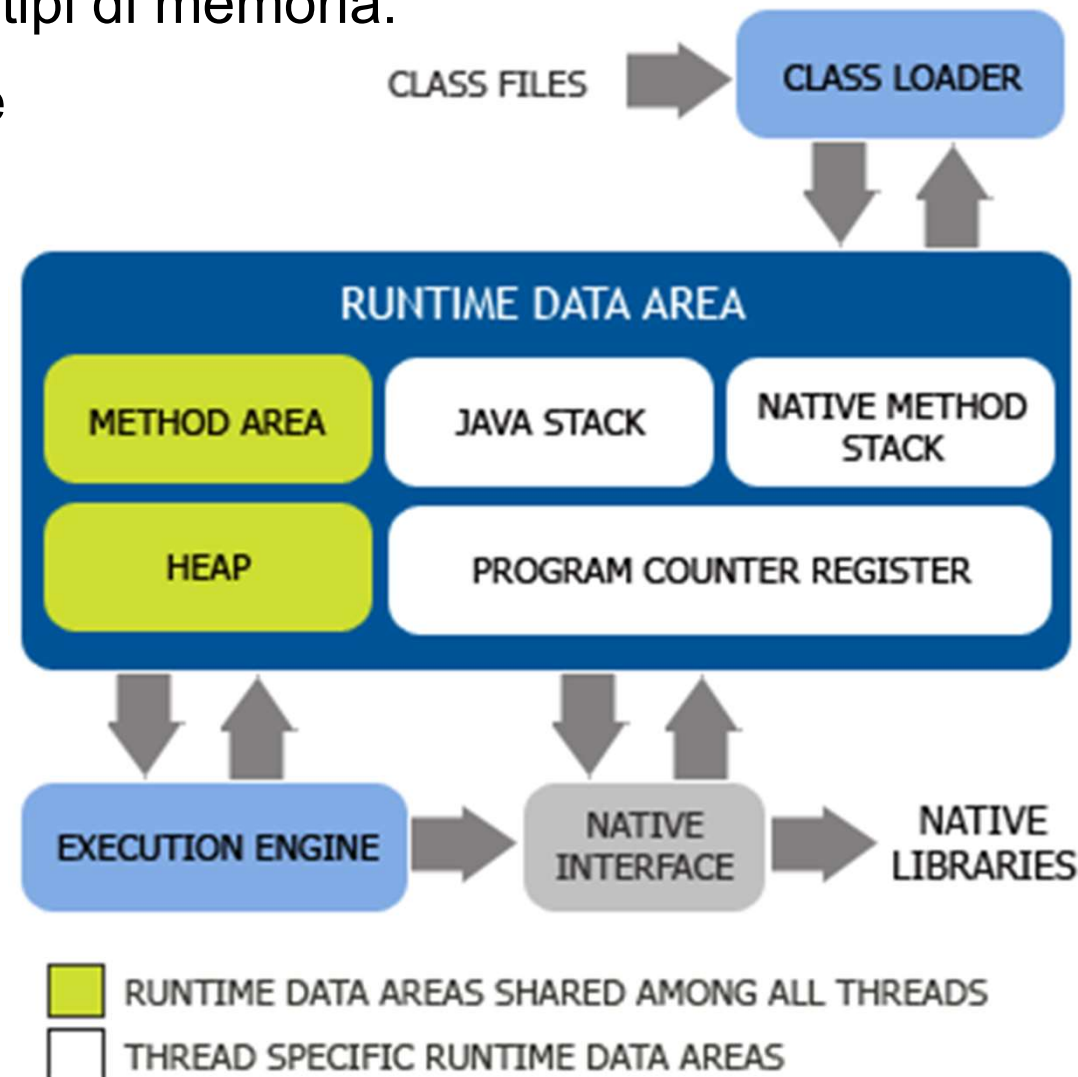
Modello di Esecuzione Semplificato

- 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

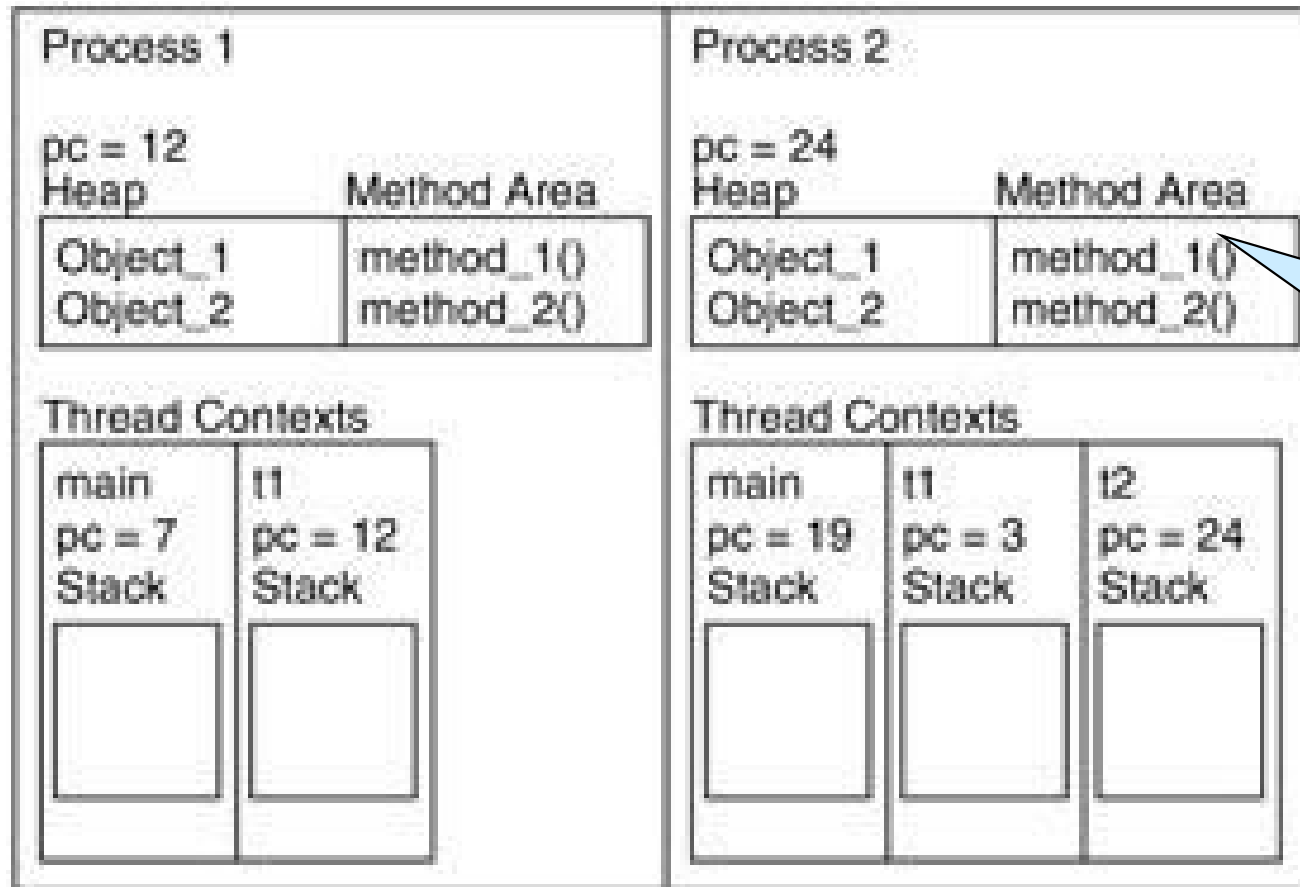
Modello di Memoria Semplificato (SMM)

- 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



Method area è condivisa tra processi che eseguono lo stesso programma

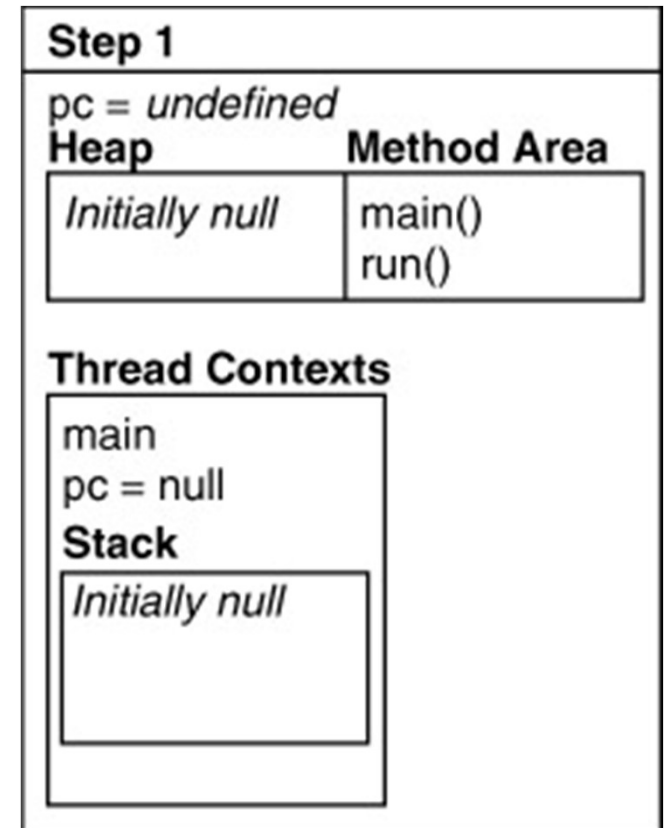
- La figura mostra la SMM di un computer che esegue due processi,
 - il primo con thread main e t1
 - il secondo con thread main, t1 e t2

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 }

```

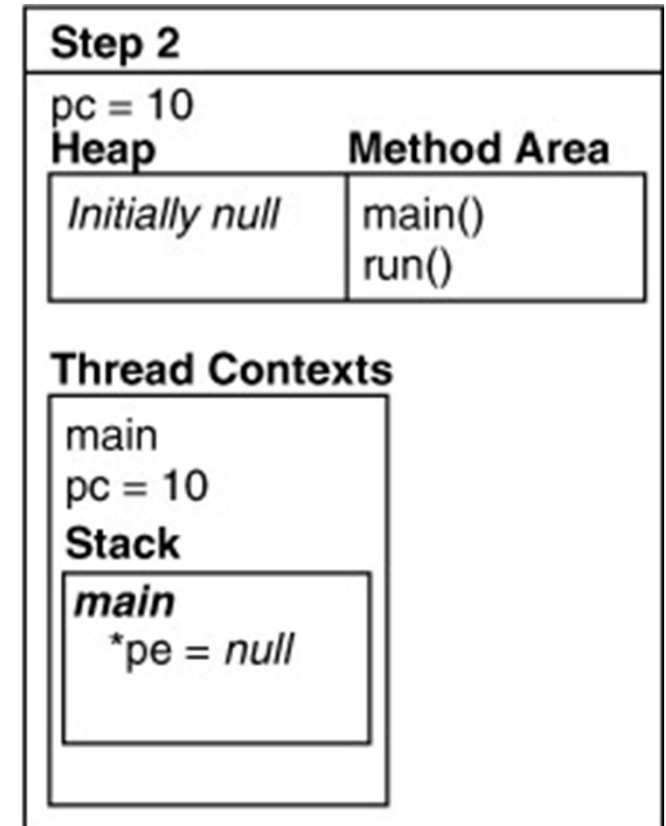


- La SVM crea lo heap e il thread context.
- Crea anche la method area e ci carica i metodi per la classe PExec.
- Infine, crea il PC con valore per il momento non definito.



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 }
```



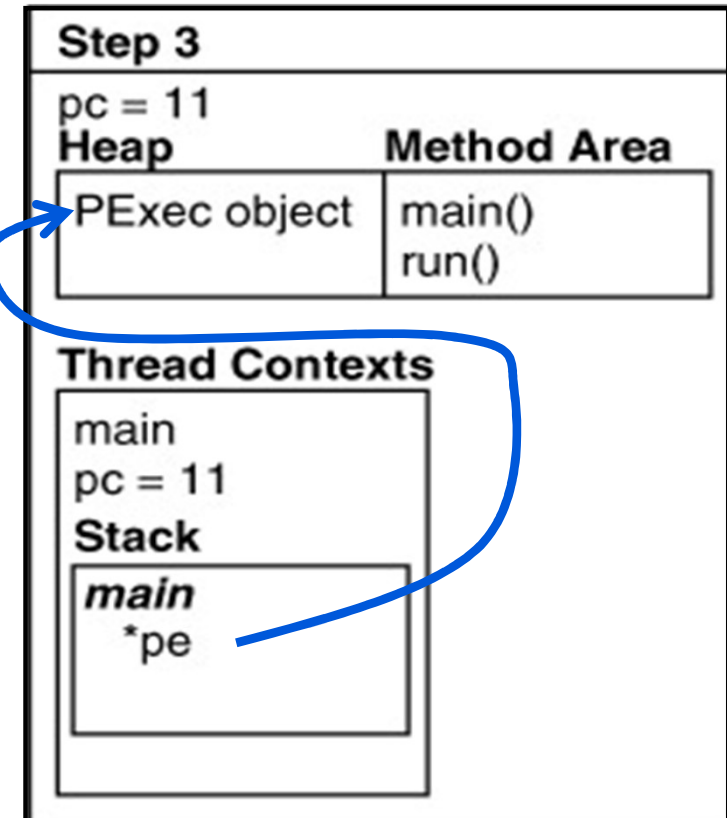
- La SVM inserisce un activation record per il metodo main nello stack.
 - Contenente un riferimento **pe** all'oggetto PExec.
- La SVM assegna la linea 10 al process PC e al thread PC (la prima riga eseguibile nel metodo main)

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 }

```



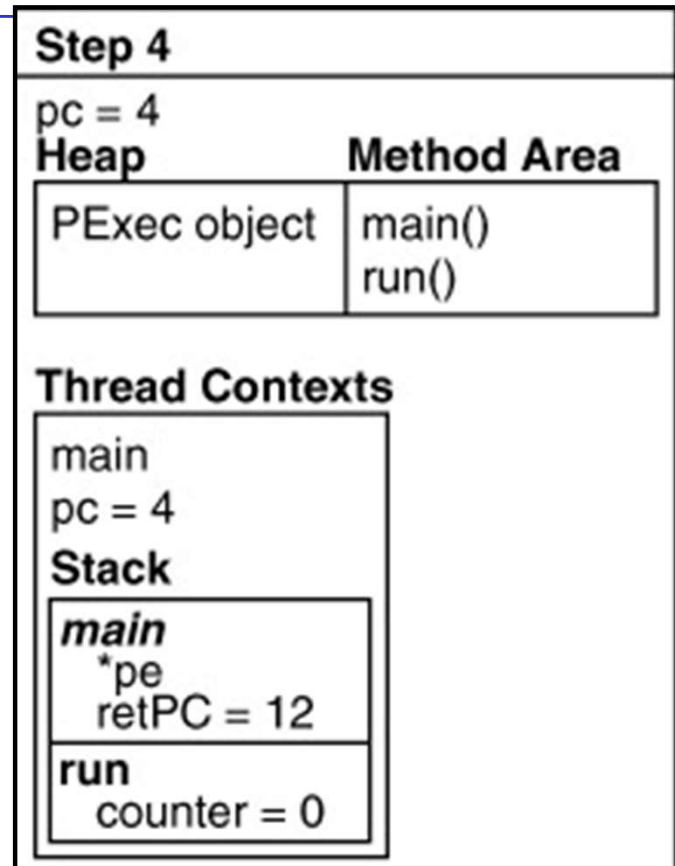
- La SVM esegue la riga 10, che crea una istanza della classe PExec nello heap. Il riferimento a questa classe è memorizzato in **pe**.
- Quindi l'SVM aggiorna il PC, che indica la riga seguente, cioè la 11.

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 }

```



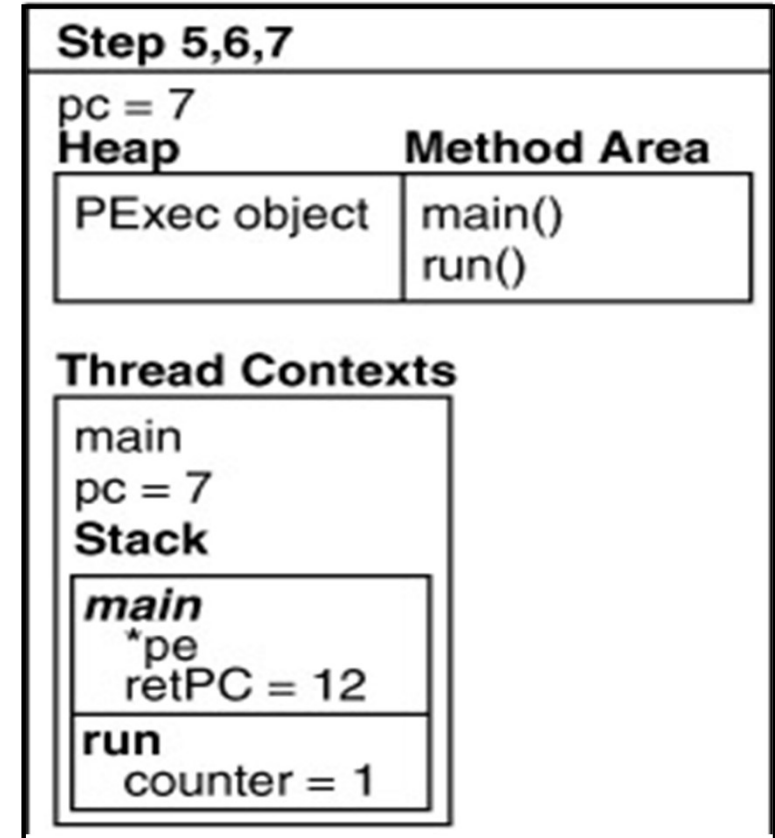
- La SVM esegue l'istruzione 11, chiamando il metodo **run**.
 - Il PC della riga successiva (12) viene memorizzato sulla pila in **retPC**.
- La SVM crea quindi un nuovo activation record per il metodo **run**, contenente la variabile locale **counter**, già inizializzata a zero.
- Infine, aggiorna il PC, che diventa 4 (indirizzo prima istruzione di **run**).

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 }

```

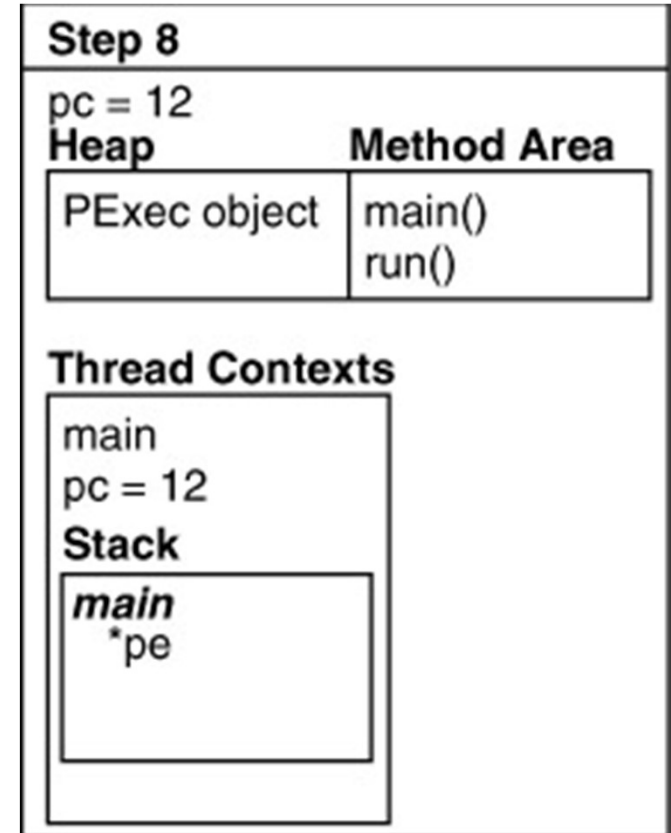


- Negli step 5, 6 e 7, la SVM esegue le righe da 4 a 6 comprese
 - ▶ stampa la variabile **counter**, l'incrementa e la stampa nuovamente.
- Alla fine il PC punta alla linea 7, e **counter** vale 1.



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 }
```

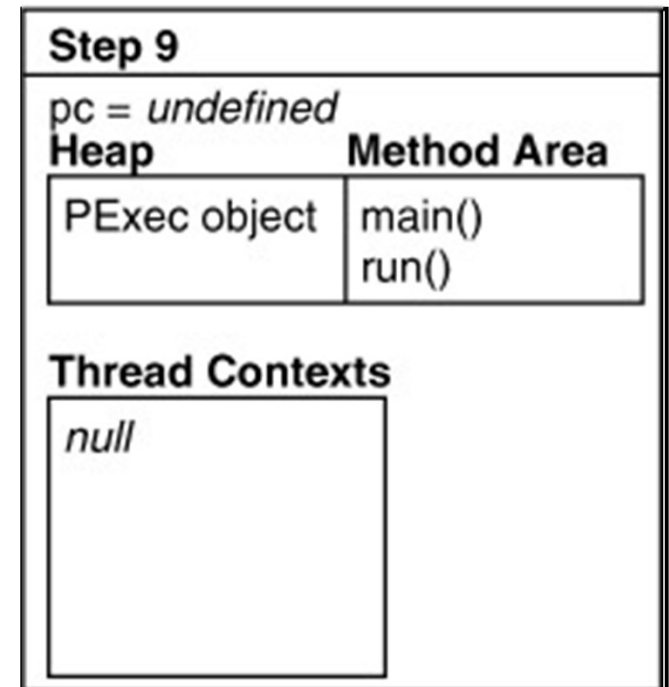


- La JVM esegue il **return**. Questo causa l'eliminazione dell'activation record del metodo **run** dallo stack, e al PC viene assegnato il valore di **retPC** (12).



SMM durante l'esecuzione di un programma single thread

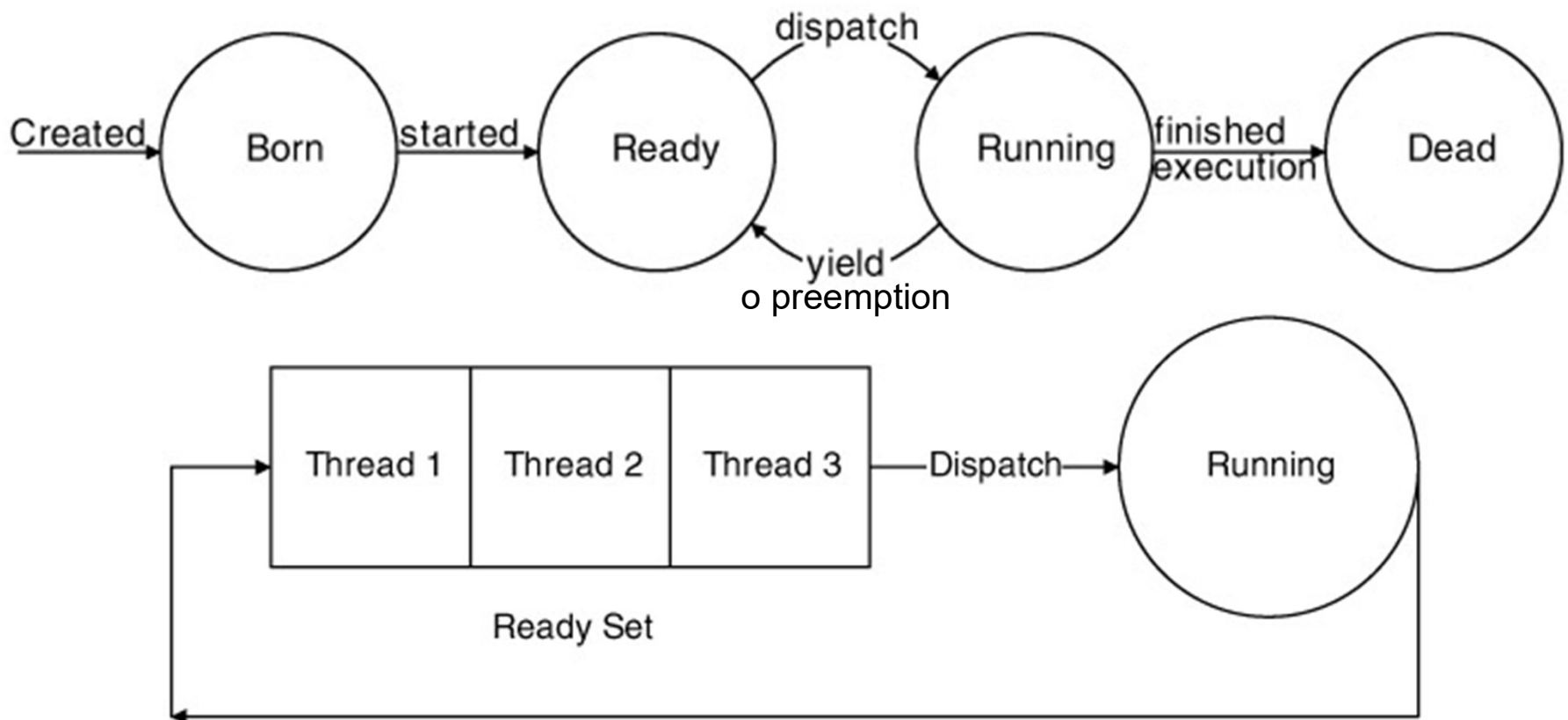
```
1 public class PExec {
2     public void run ( ) {
3         int counter = 0;
4         System.out.println("In run, counter= " + counter);
5         counter++;
6         System.out.println("In run, counter= " + counter);
7         return;
8     }
9     public static void main (String args[]) {
10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

SVM Thread States

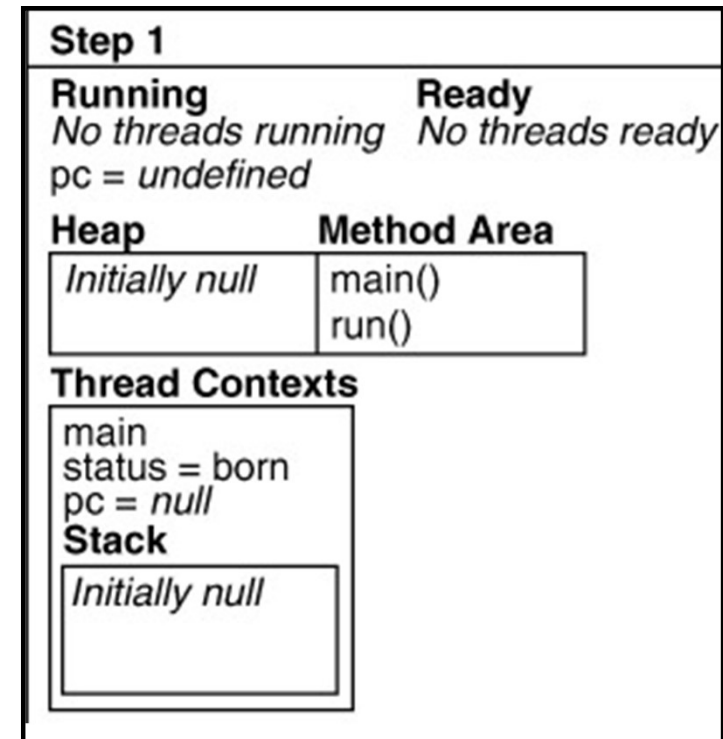
- Poiché un programma concorrente ha più thread, ognuno con il suo contesto e PC, la SVM deve scegliere quale thread mandare in esecuzione.
- La decisione su quale Thread mandare in esecuzione dipende da molti fattori.





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 }
```



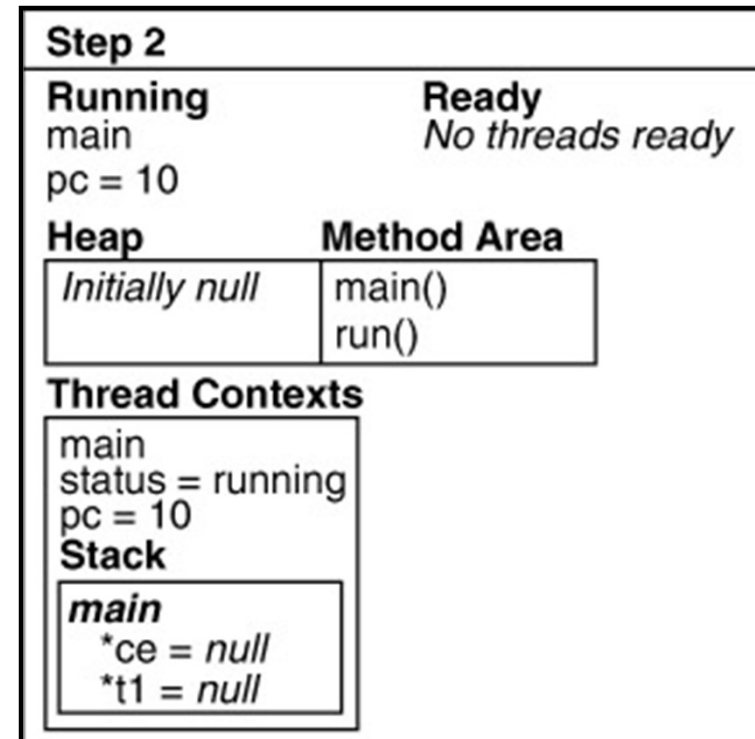
- La SVM crea lo heap e il thread context.
- Crea anche la method area e ci carica i metodi per la classe **CExec**.
- Infine, crea il PC, con valore per il momento non definito

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 }

```

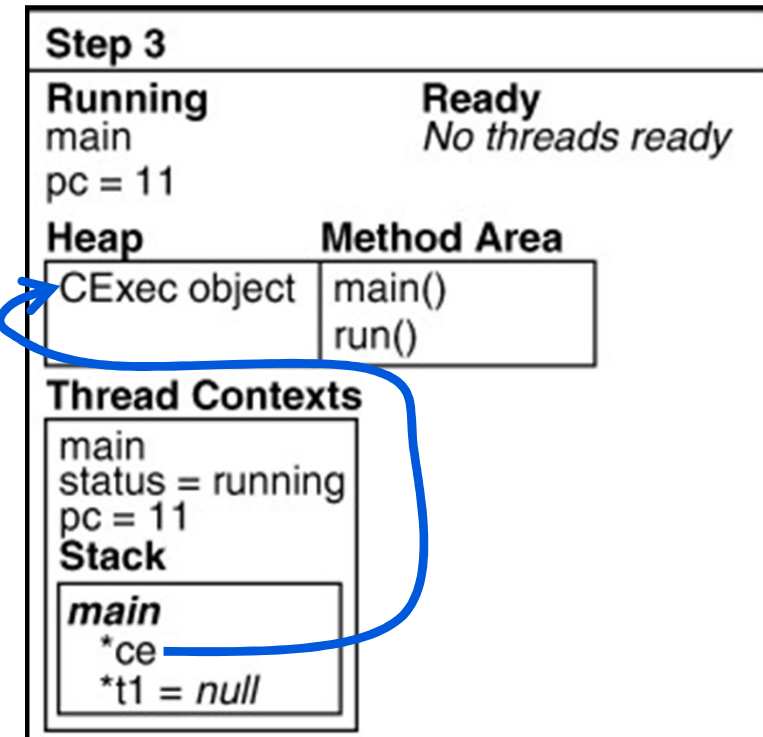


- La SVM inserisce un activation record per il metodo **main** nello stack, contenente un riferimento per l'oggetto **ce** e uno per il thread **t1**
- La SVM assegna la linea 10 al process PC e al thread PC (la prima riga eseguibile nel metodo main)



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

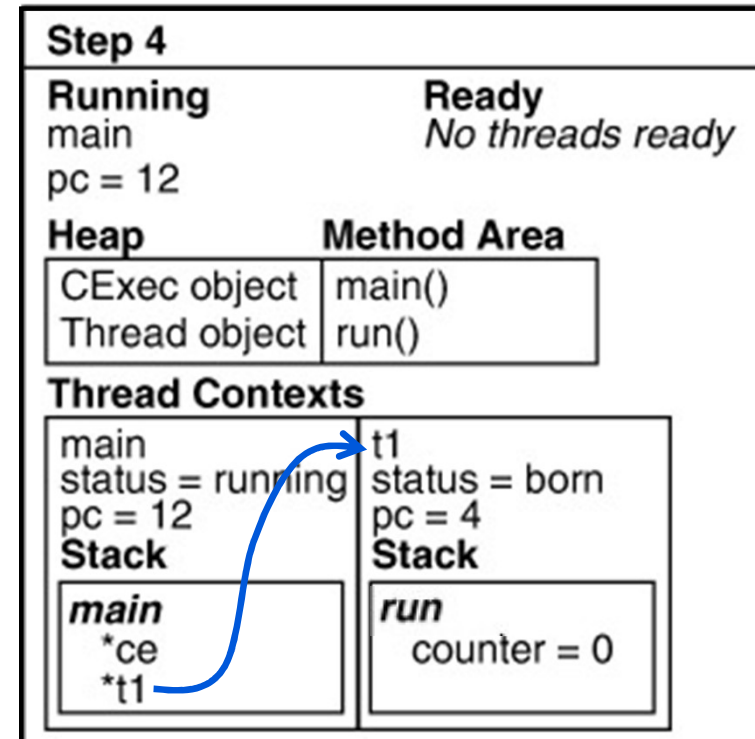
SMM durante l'esecuzione di un programma multithread

```

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**.
- Il PC viene incrementato a 12

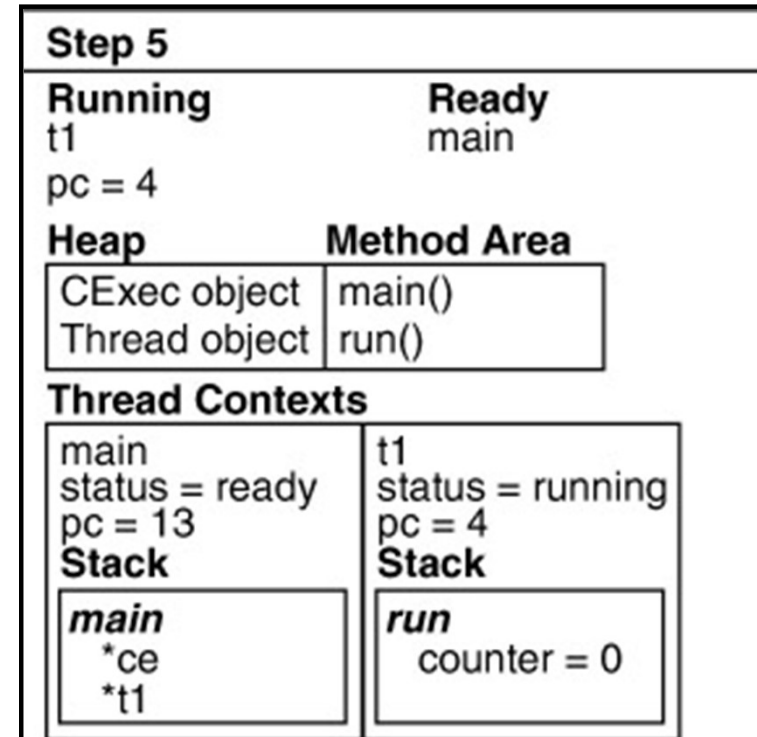




SMM durante l'esecuzione di un programma multithread

```
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 }
```

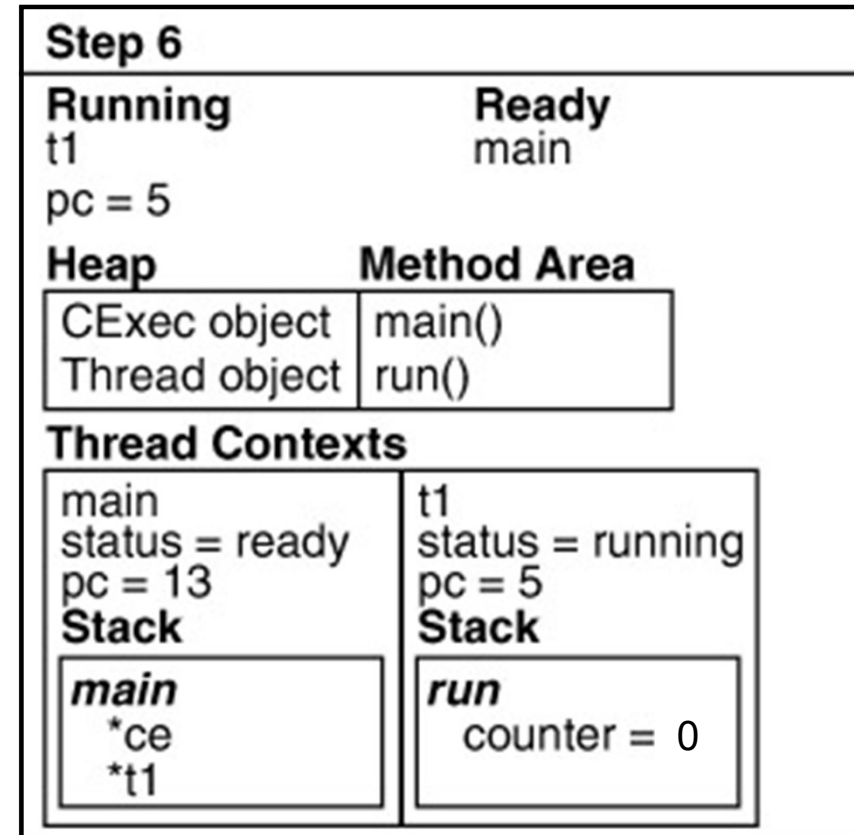
- La SVM esegue la linea 12: il thread **t1** diventa ready.
 - ▶ Quindi si possono eseguire sia **main** sia **t1**.
- Lo scheduler sceglie il thread **t1**, con conseguente context switch dal thread **main** a **t1**.
 - ▶ Il PC di processo assume il valore del PC del thread **t1**





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 }
```

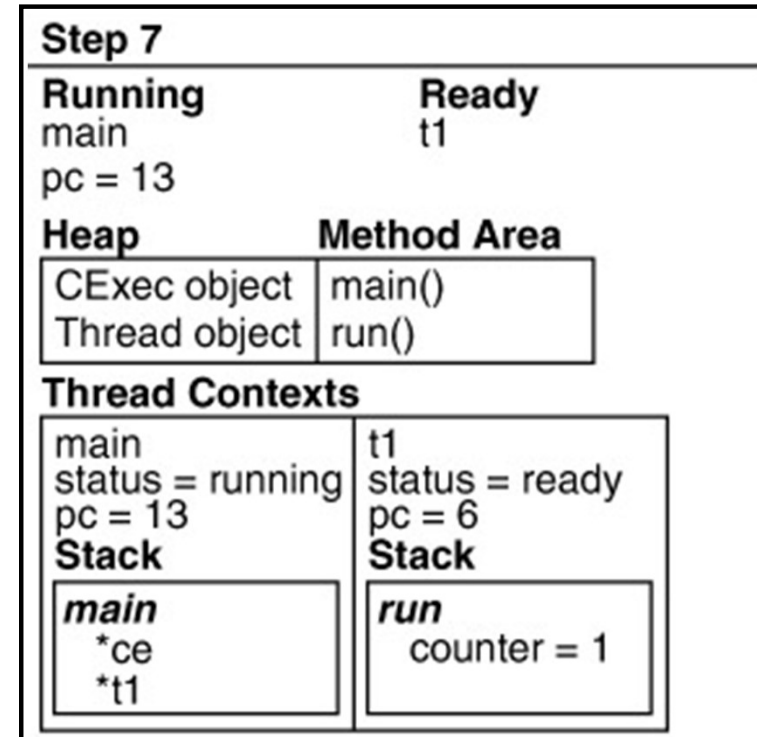


- Viene eseguita l'istruzione 4 del thread **t1**.
- L'SVM deve nuovamente scegliere quale thread eseguire e sceglie ancora **t1**



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 }
```



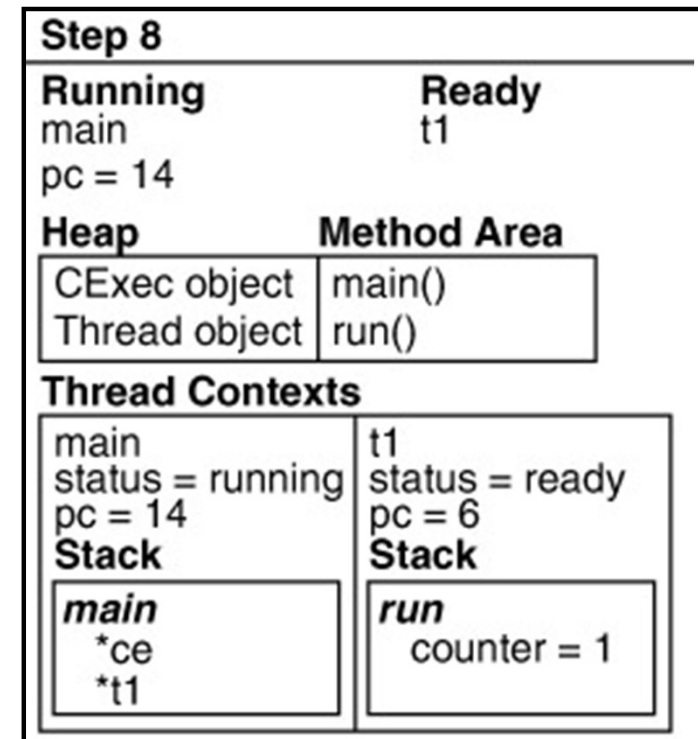
- Viene eseguita la linea 5: **counter** viene incrementato nel record di attivazione di **run**.
- La SVM adesso sceglie di eseguire il thread **main** ed esegue quindi un nuovo context switch.
 - Il PC di processo diventa = al PC di main, che vale 13



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 }
```

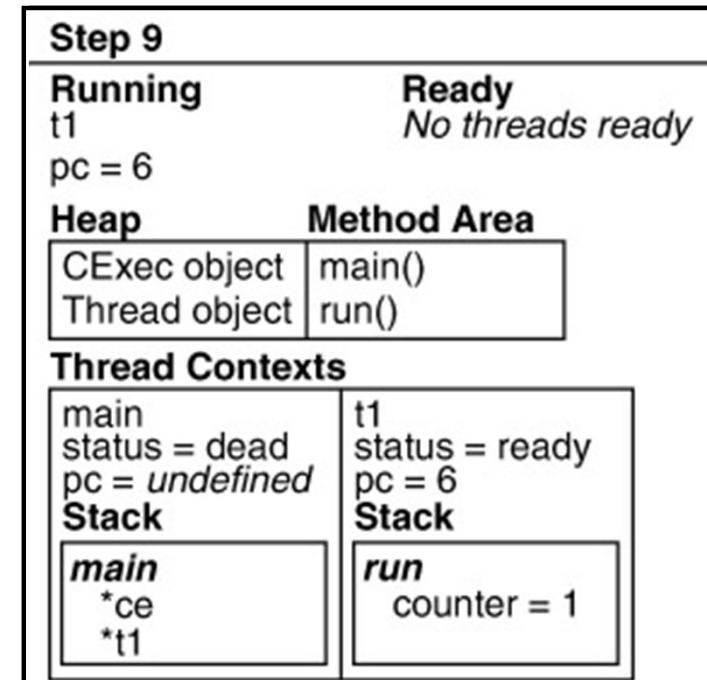
- Viene eseguita la linea 13.
- L'SVM sceglie di continuare con il thread **main**, aggiornando il PC alla linea 14.





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 thread figli non daemon).

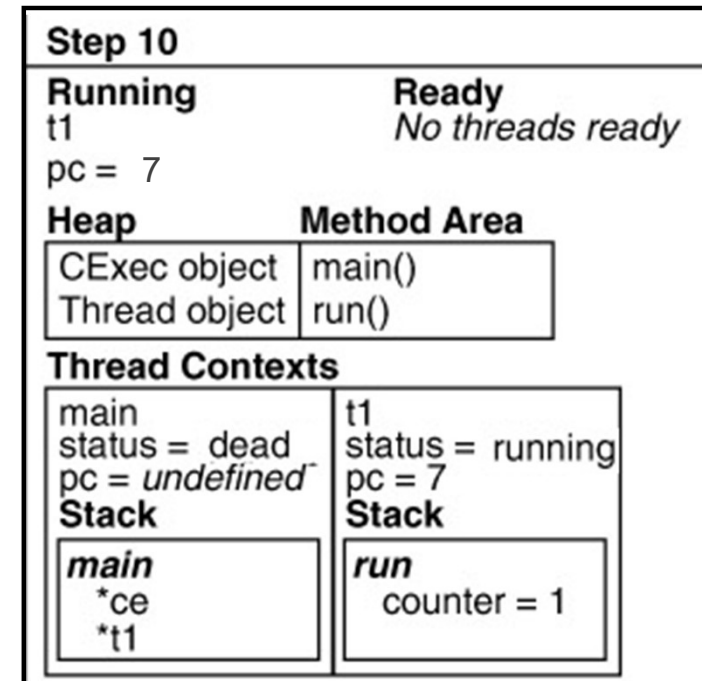
● Si ritorna ad eseguire **t1**, che è l'unico thread ready.

▶ 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

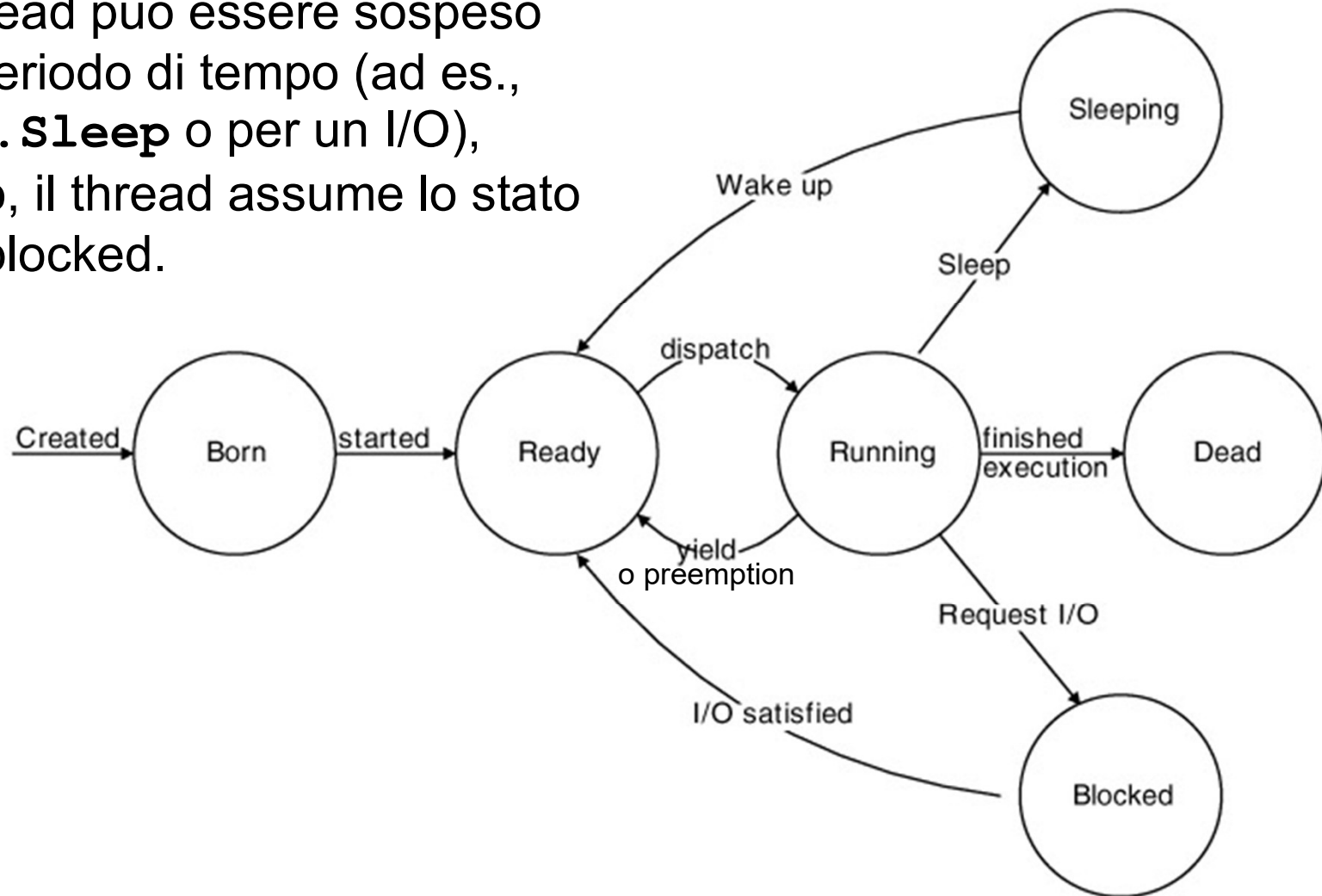
```
1 public class CExec implements Runnable {
2     public void run ( ) {
3         int counter = 0;
4         System.out.println("In run, counter= " + counter);
5         counter++;
6         System.out.println("In run, counter= " + counter);
7         return;
8     }
9     public static void main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

- 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.

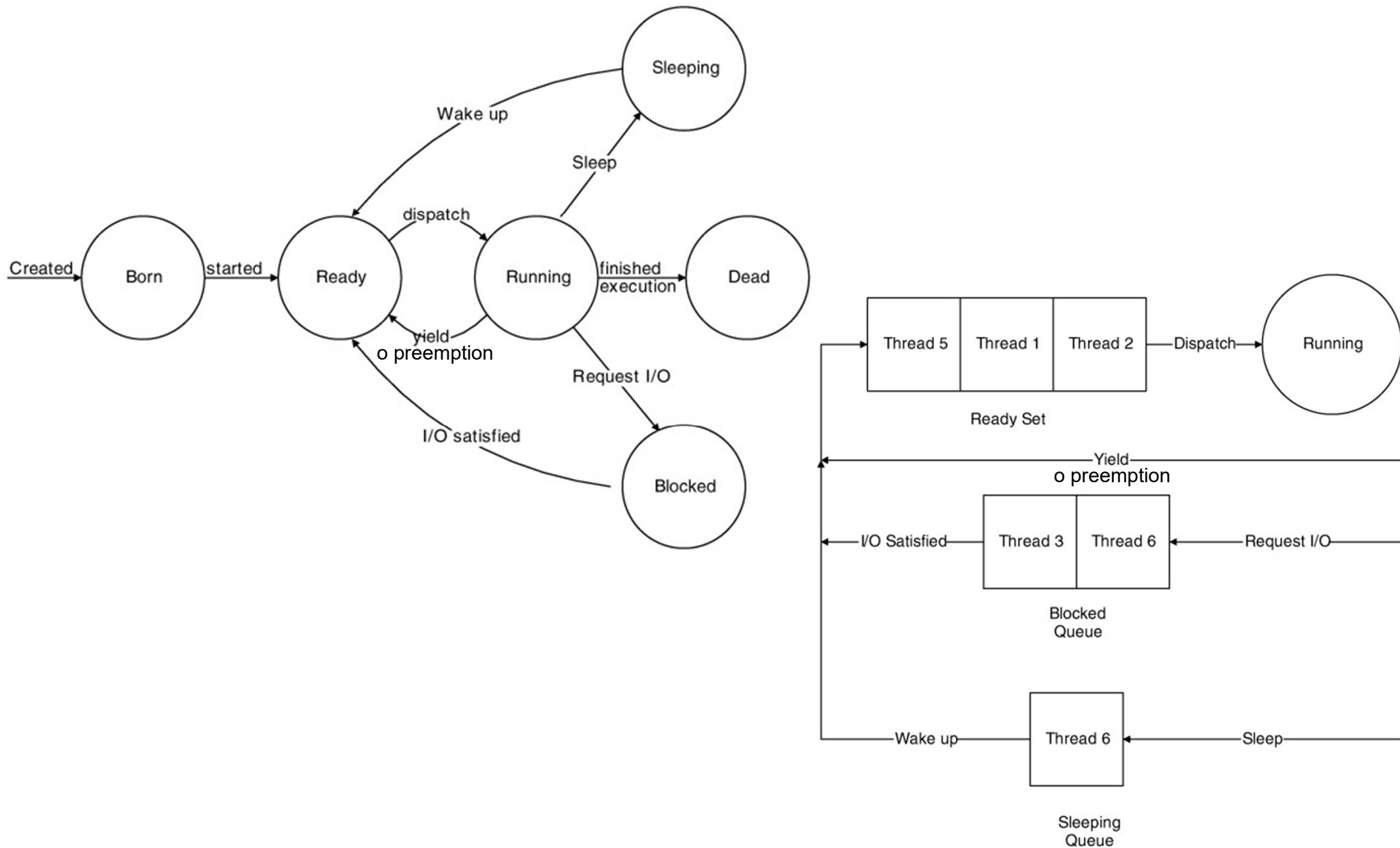
Step 10	
Running <i>No threads running</i> <i>pc = undefined</i>	Ready <i>No threads ready</i>
Heap	Method Area
CExec object Thread object	main() run()
Thread Contexts	

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 (ad es., dopo **Thread.Sleep** o per un I/O), in questo caso, il thread assume lo stato di sleeping o blocked.



Lo scheduler gestisce tante code





Far partire i thread: `start()`

- Una chiamata `t.start()` rende il thread `t` pronto all'esecuzione.
- 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 NondeterminismExample extends Thread{
    final static int numIterations=10;
    public void run() {
        for(int i=0; i<numIterations; i++) {
            System.out.println("Nuovo thread");
        }
    }
    public static void main(String args []) {
        NondeterminismExample t=new NondeterminismExample();
        t.start();
        for(int i=0; i<numIterations; 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

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 NondeterminismExample extends Thread{
    final static int numIterations=5;
    public void run() {
        for(int i=0; i<numIterations; i++) {
            System.out.println("Nuovo thread");
        }
    }
    public static void main(String args []) {
        NondeterminismExample t=new NondeterminismExample();
        t.start();
        try { Thread.sleep(1); }
        catch (InterruptedException e) { }
        for(int i=0; i<numIterations; i++) {
            System.out.println("Main");
        }
    }
}
```

Dopo la sleep, solo il thread `t` è ready

Output

- Naturalmente, nulla vieta che succeda questo:

```
Nuovo thread  
Main  
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
```



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
 - ▶ In pratica, serve ai thread per «autosospendersi» per un certo tempo
 - NB: scaduto il tempo, il thread torna ready, ma non è detto che vada subito in esecuzione. Quindi `sleep(100)` sospende il task per **almeno** 100 ms.



`sleep()` e interruzioni

- 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)
- In generale, quando non ci sono altri thread che possono mandarci interruzioni possiamo tranquillamente ignorare l'eccezione



Un esempio di programma concorrente con `sleep()`

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



Alternativa a `sleep`: 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() {  
    long startTime=System.currentTimeMillis();  
    long stopTime=startTime+60000; // 60 seconds in the future  
    while(System.currentTimeMillis()<stopTime) {  
        // do nothing  
    }  
}
```

- È una buona soluzione?
 - ▶ Generalmente no: si spreca 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” finché il metodo `run()` non ritorna
- Usare `isAlive()` per attendere la terminazione 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 la terminazione di `t1`:

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



Esempio: Attendere la terminazione di altri Thread

```
public class ExampleWithSleeps extends Thread{
    final static int numIterations=10;
    public ExampleWithSleeps(String s) {
        super(s) ;
    }
    public void run() {
        for(int i=0; i<numIterations; i++) {
            System.out.println("thread "+getName()+" in esec.");
            try {
                Thread.sleep(200) ;
            } catch (InterruptedException e) { }
        }
        System.out.println("thread "+getName()+" : finito.");
    }
}
```



Esempio: Attendere la terminazione di altri Thread

```
public class AliveTestExit {  
    public static void main(String args[]) throws Exception {  
        System.err.println("I thread stanno per partire");  
        Thread t1 = new ExampleWithSleeps("t1"); t1.start();  
        Thread t2 = new ExampleWithSleeps("t2"); t2.start();  
        Thread t3 = new ExampleWithSleeps("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  
thread t1 in esec. I thread sono partiti  
  
thread t3 in esec.  
  
chiude l'applicazione
```

Attenzione

L'applicazione esce prima della terminazione dei suoi task



Esempio: Attendere la terminazione di altri Thread

```
public class AliveTestExit {
    public static void main(String args[]) throws Exception
    {
        System.err.println("I thread stanno per partire");
        Thread t1 = new ExampleWithSleeps("t1"); t1.start();
        Thread t2 = new ExampleWithSleeps("t2"); t2.start();
        Thread t3 = new ExampleWithSleeps("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 JoinWaitExit {  
    public static void main(String args[]) {  
        System.err.println("I thread stanno per partire");  
        Thread t1 = new ExampleWithSleeps("t1"); t1.start();  
        Thread t2 = new ExampleWithSleeps("t2"); t2.start();  
        Thread t3 = new ExampleWithSleeps("t3"); t3.start();  
        System.err.println("I thread sono partiti\n");  
        try {  
            t1.join();  
            t2.join();  
            t3.join();  
        } catch (InterruptedException e) { }  
        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.

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...

Thread (Java Platform SE 7) - Google Chrome

docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

boolean	<code>isInterrupted()</code> Tests whether this thread has been interrupted.
void	<code>join()</code> Waits for this thread to die.
void	<code>join(long millis)</code> Waits at most <code>millis</code> milliseconds for this thread to die.
void	<code>join(long millis, int nanos)</code> Waits at most <code>millis</code> milliseconds plus <code>nanos</code> nanoseconds for this thread to die.
void	<code>resume()</code> Deprecated. <i>This method exists solely for use with <code>suspend()</code>, which has been deprecated because it is deadlock-prone. For more information, see Why are Thread.stop, Thread.suspend and Thread.resume Deprecated?.</i>
void	<code>run()</code> If this thread was constructed using a separate <code>Runnable</code> run object, then that <code>Runnable</code> object's <code>run</code> method is called; otherwise, this method does nothing and returns.



Join con timeout

```
public class ExampleWithSleeps extends Thread{
    final static int numIterations=10;
    public ExampleWithSleeps(String s) {
        super(s);
    }
    public void run() {
        for(int i=0; i<numIterations; i++) {
            System.out.println("thread "+getName()+" in esec.");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {}
        }
        System.out.println("thread "+getName()+" : finito.");
    }
}
```



Join con timeout

```
public class JoinWaitExit {
    public static void main(String args[]) {
        System.err.println("I thread stanno per partire");
        Thread t1 = new ExampleWithSleeps("t1"); t1.start();
        System.err.println("Fine creazione thread\n");
        try {
            t1.join(1000);
            if(t1.isAlive()) {
                System.out.println("t1 non terminato dopo timeout");
            }
        } catch (InterruptedException e) { }
        System.err.println("chiude l'applicazione");
        System.exit(0); //chiude l'applicazione in ogni caso
    }
}
```



Esecuzione

I thread stanno per partire

Fine creazione thread

thread t1 in esec.

thread t1 in esec.

thread t1 in esec.

thread t1 in esec.

thread t1 in esec.

Timeout scaduto e t1 non terminato

chiude l'applicazione

Come fermare un 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 mantenuto.
- Tutti gli altri sono stati deprecati.
- 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 JoinWaitExit {  
    public static void main(String args[]) {  
        System.err.println("I thread stanno per partire");  
        Thread t1 = new ExampleWithSleeps("t1"); t1.start();  
        Thread t2 = new ExampleWithSleeps("t2"); t2.start();  
        Thread t3 = new ExampleWithSleeps("t3"); t3.start();  
        System.err.println("I thread sono partiti\n");  
        t1.interrupt();  
        t2.interrupt();  
        t3.interrupt();  
        try {  
            t1.join();  
            t2.join();  
            t3.join();  
        } catch (InterruptedException e) { }  
        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 nel thread **ExampleWithSleeps**



Esecuzione

- Se non modifichiamo **ExampleWithSleeps**, l'esecuzione non cambia!
- Perché l'eccezione **InterruptedException** non è gestita nel thread **ExampleWithSleeps**
- Se vogliamo che **ExampleWithSleeps** si accorga dell'eccezione, dobbiamo modificarne il codice.



Esempio: terminazione in seguito a interrupt

```
public class ExampleWithSleeps extends Thread{
    final static int numIterations=10;
    public ExampleWithSleeps(String s) {
        super(s);
    }
    public void run() {
        for(int i=0; i<numIterations; i++) {
            System.out.println("thread "+getName()+" in esec.");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                System.err.println(getName() + ": interrotto");
                break;
            }
        }
        System.out.println("thread "+getName()+" : finito.");
    }
}
```

Gestisce l'interruzione uscendo dal loop e quindi terminando



Esempio: Interrompere i Thread con `interrupt()`

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

Cosa succede, adesso che gli interrupt sono gestiti dal thread che li riceve?



Esempio: Interrompere i Thread con `interrupt()`

Nuovo output

```
I thread stanno per partire
I thread sono partiti
thread t1 in esec.

thread t3 in esec.
thread t2 in esec.
t1: interrotto
t3: interrottothread t1: finito.

t2: interrottothread t3: finito.

thread t2: finito.
chiude l'applicazione
```



Problemi con `interrupt()`

- Il metodo `interrupt()` non genera eccezioni se il thread “interrotto” non esegue mai metodi di attesa (`sleep`, `join`, ...)
- I thread che si aspettano degli interrupt devono cooperare, ad esempio controllando periodicamente il loro stato di “interruzione”:
 - ▶ `Thread.interrupted()` controlla il flag di interruzione del thread corrente e se settato lo resetta
 - Metodo statico
 - ▶ `isInterrupted()` controlla il flag di interruzione senza resettarlo



Thread che controlla la ricezione di interrupt

```
public void run(){
    boolean interrupted=false;
    while(condizione){
        // esegue un'operazione complessa
        if (Thread.interrupted()){ // o this.isInterrupted
            interrupted=true;
            break;
        }
    }
    if(interrupted) {
        // gestisce l'interruzione
    }
}
```




Collaborazione: `Thread.yield()`

- La chiamata al metodo statico `Thread.yield()` permette ad un thread di lasciare volontariamente il processore ad un altro thread
Ho fatto una parte importante del mio lavoro e questo è un buon momento per passare a un altro thread per un po'
- Utile soprattutto nel caso di un thread che non esegue spesso operazioni (ad es., di I/O) che quando lo scheduler non è preemptive 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, quando tutto il programma è concepito in modo da rendere naturale l'uso di `yield()`



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();
        }
    }
}
```



Esempio completo con `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

1(4)

4(4)

3(4)

0(4)

2(4)

0(3)

3(3)

4(3)

1(3)

4(2)

3(2)

0(2)

2(3)

0(1)

3(1)

4(1)

1(2)

4(Decollo !)

3(Decollo !)

0(Decollo !)

2(2)

1(1)

2(1)

1(Decollo !)

2(Decollo !)

Nessun thread scrive mai due volte di fila!

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 Distributed Computing in Java by Charles W. Kann
 - ▶ nel capitolo 'Concurrency' di: Thinking in Java by Bruce Eckel et. Al.