Multithreading in JAVA

Advanced Computer Programming

Prof. Luigi De Simone

Sommario

- Multithreading in JAVA
- Creazione di un thread
- Scheduling dei thread
- Operazioni sui thread

Riferimenti:

- The Java Tutorials: Concurrency - http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- B. Eckel, “Thinking in Java”, capitolo 13

Multithreading in Java (1/2)

- Java supporta la programmazione multithread al **livello di linguaggio**.
- Tipicamente, i thread sono implementati a livello di sistema, richiedendo un'interfaccia di programmazione dipendente dalla piattaforma su cui girerà il programma
  - Es. C++: libreria pthread, POSIX Threads Programming, per implementare un programma multithread su piattaforme UNIX.

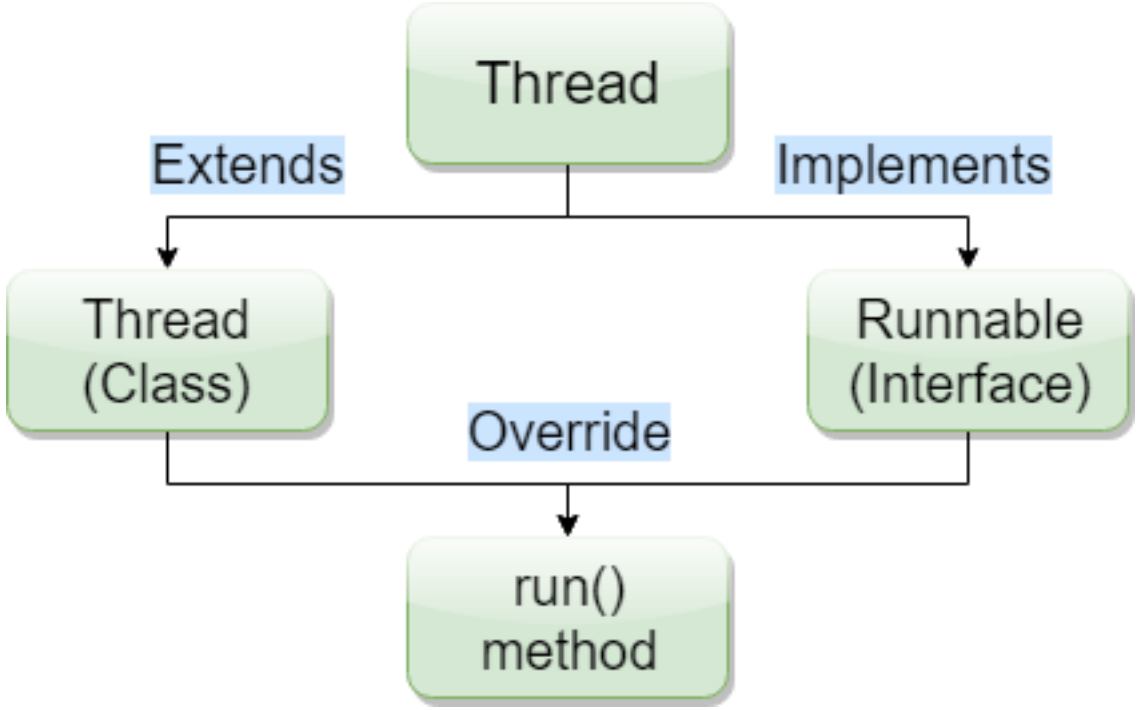
Multithreading in Java (2/2)

- **Java consente di realizzare programmi multithread in maniera standardizzata e indipendente dalla specifica piattaforma.**

- **Java fornisce:**
  - primitive per definire attività indipendenti.
  - primitive per la comunicazione e sincronizzazione tra attività eseguite in modo concorrente.

Creare un Thread

- **Java offre due possibili tecniche per poter creare un thread (java.lang):**
  - Derivazione dalla classe Thread
  - Implementazione dell'interfaccia Runnable

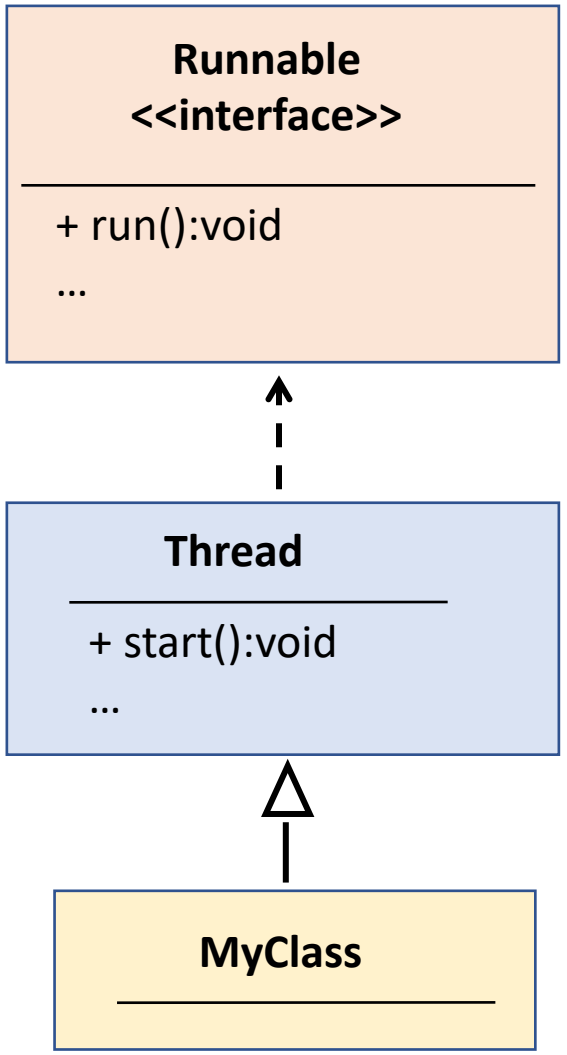


Creare un Thread

La classe utente deriva da Thread, ridefinendo il metodo *run()*.

```
public class MyThread extends Thread{
    public void run(){
      doWork();   //codice del thread
    }
}

Thread t= new MyThread();
t.start();
```



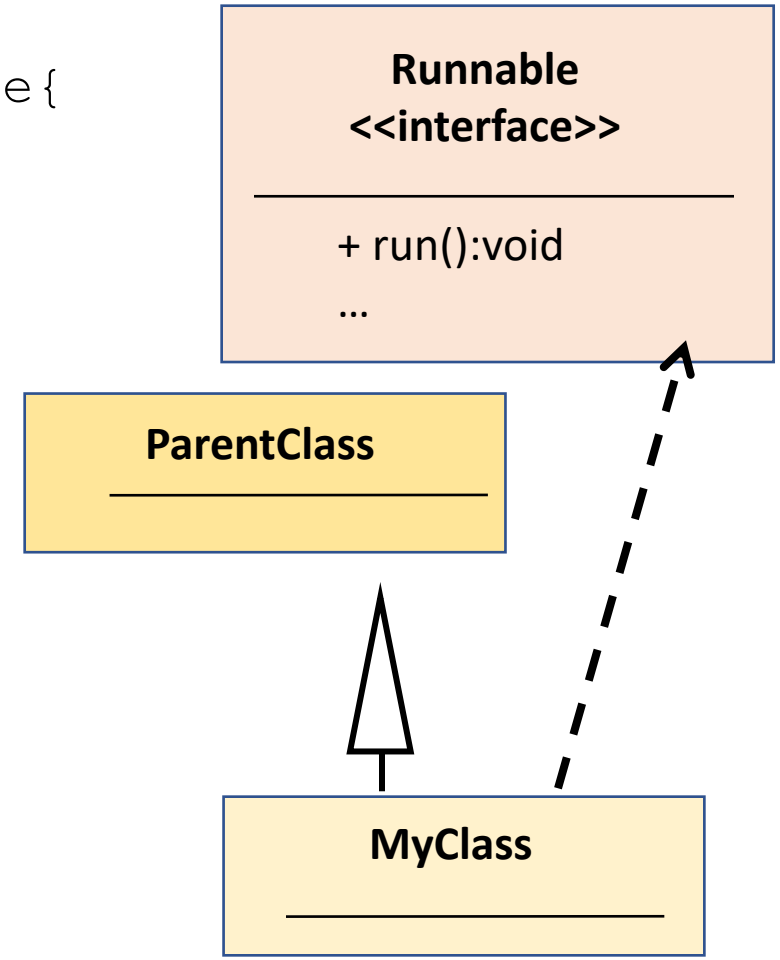
Creare un Thread

La classe utente implementa l'interfaccia *Runnable*, ridefininendo il metodo *run()*.

```
public class MyRunnable implements Runnable{

    public void run(){
      doWork();   //codice del thread
    }
}

…
Runnable r= new MyRunnable();
Thread t=new Thread(r);
t.start();
```



Creare un Thread

Perché Java offre due diverse soluzioni per la creazione di un thread?

- Java non consente la derivazione multipla.
- **Se la classe utente non è già coinvolta in un legame di derivazione**, possiamo usare la soluzione di derivare dalla classe Thread
- **Se la classe utente è già coinvolta in un legame di derivazione**, possiamo usare la soluzione di implementare l'interfaccia Runnable, ridefinendo il metodo `run()`

Costruttori della classe Thread

- Il costruttore principale di **`Thread`** è

```
Thread(ThreadGroup group, Runnable target, String name)
```

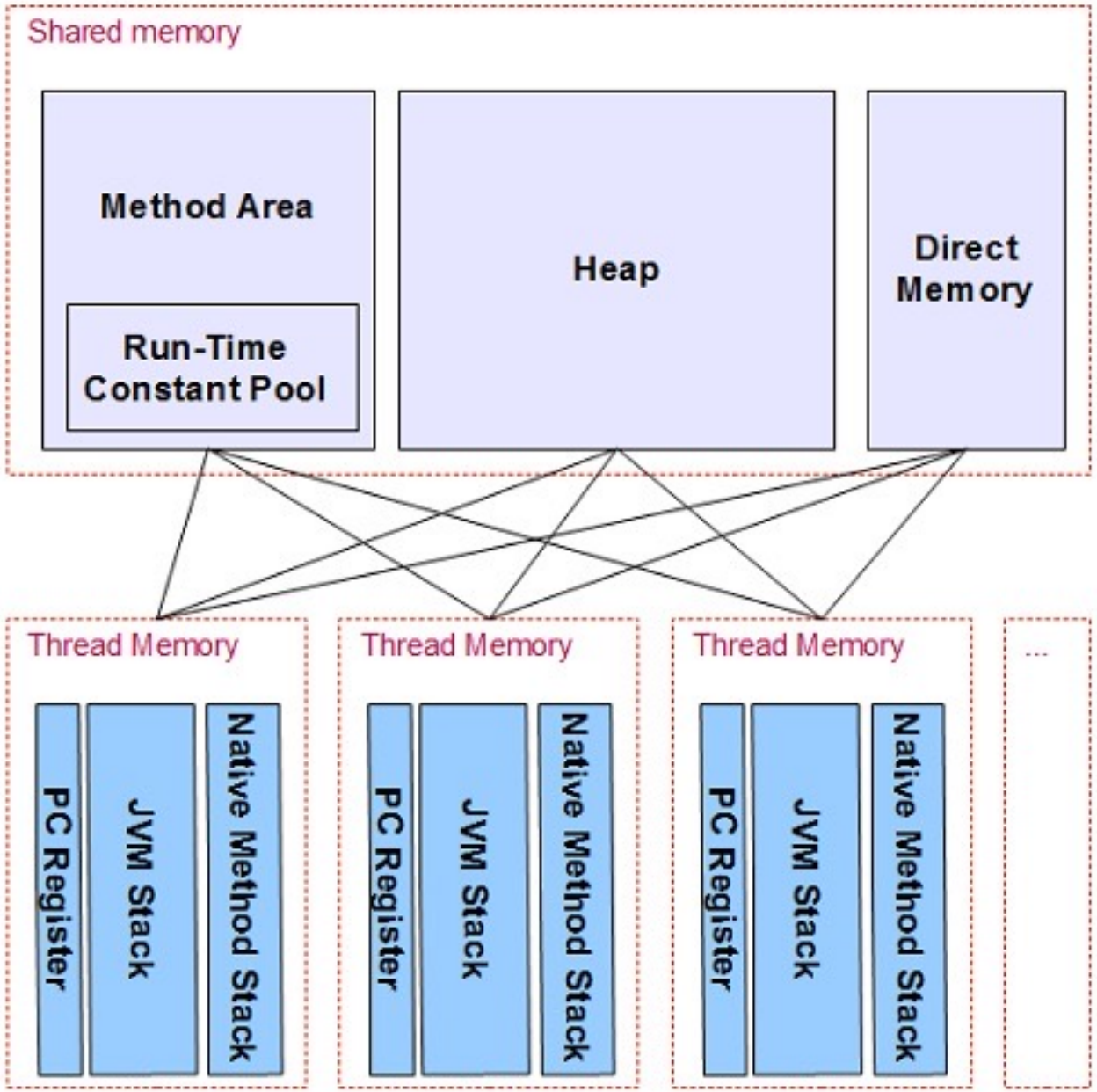
  - group: il gruppo a cui appartiene il thread
  - target: un oggetto di tipo runnable
  - name: il nome del Thread

Consultare la javadoc per le altre versioni del costruttore!

Threads e applicazioni

- Threads diversi all'interno della stessa applicazione (programma) condividono la maggior parte dello stato
  - sono condivisi **l'ambiente delle classi** e l'area **heap**
  - ogni thread ha un proprio **stack delle attivazioni**
  - per quanto riguarda le variabili
    - **sono condivise le variabili statiche** (classi) e **le variabili di istanza** (heap)
    - non sono condivise le variabili locali dei metodi (stack)

Threads e applicazioni



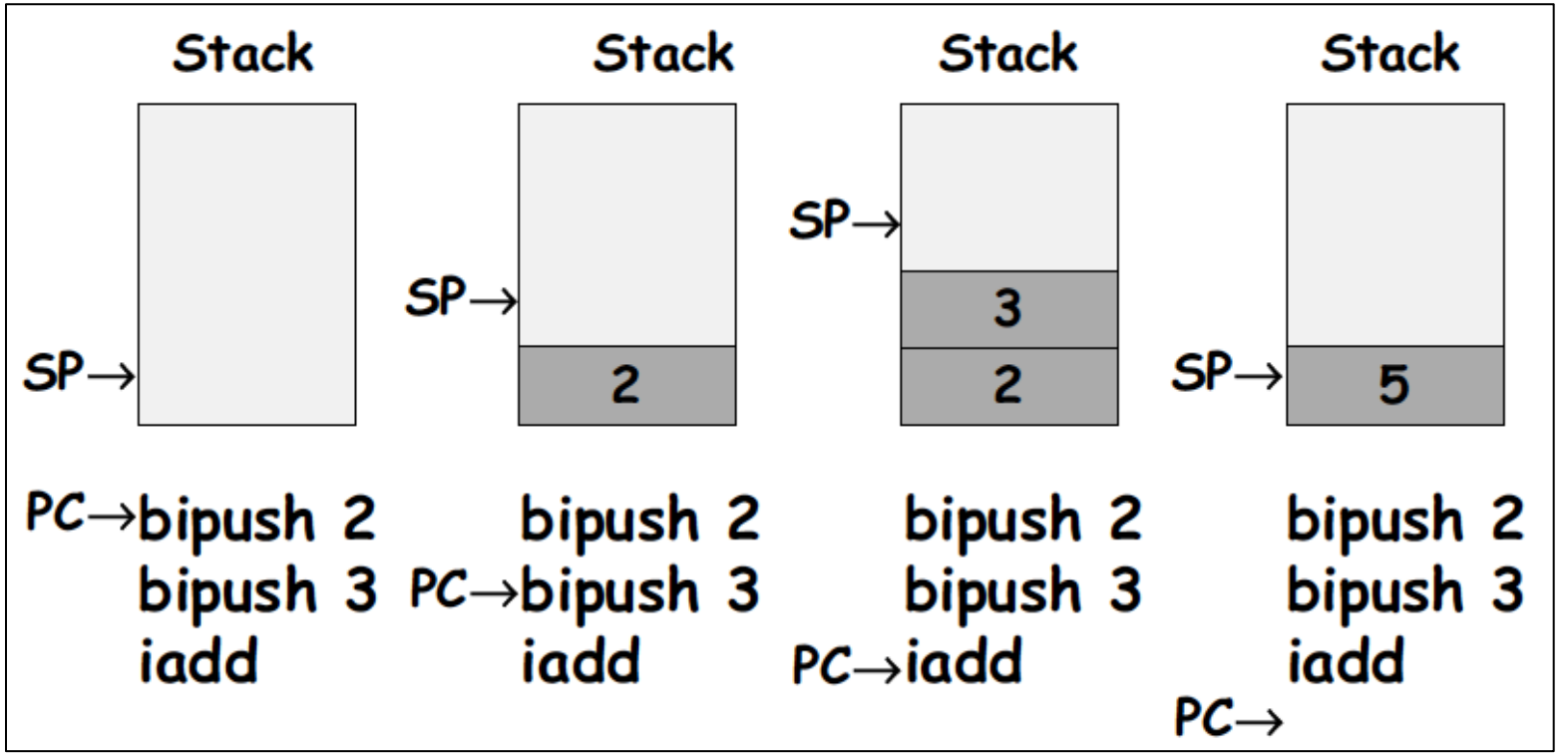
Method Area: memorizza strutture dati associate alla classe, e.g., il *run-time constant pool*, dati per attributi e metodi, e il codice per i metodi definiti e i costruttori

Direct Memory: memorizza oggetti allocati esplicitamente nella *direct memory area*, che sono automaticamente deallocati dal *JVM garbage collector*

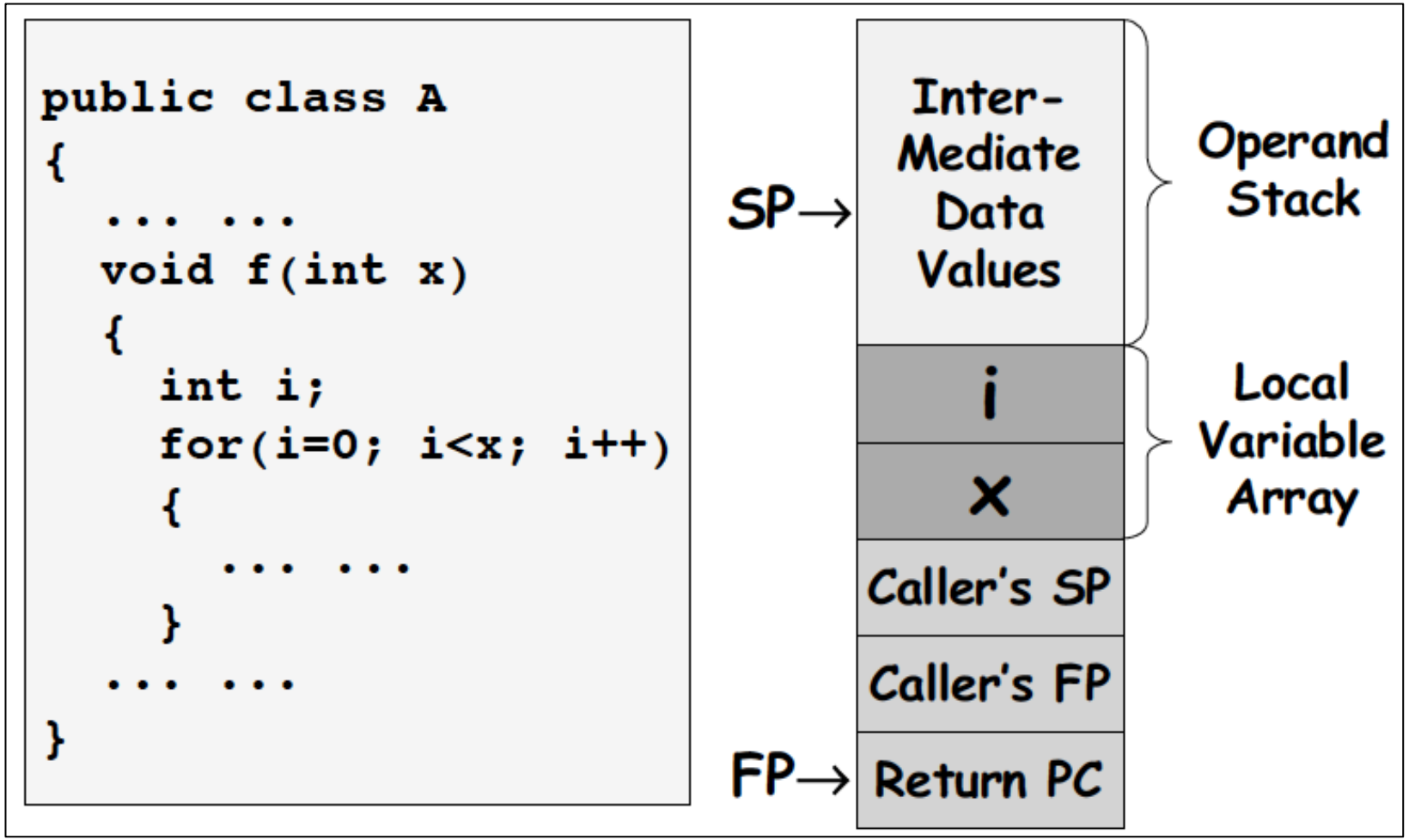
Native Method Stack: un'area data per ogni thread che memorizza informazioni per eseguire metodi nativi (non-Java)

Execution engine

- Execution engine in “azione”

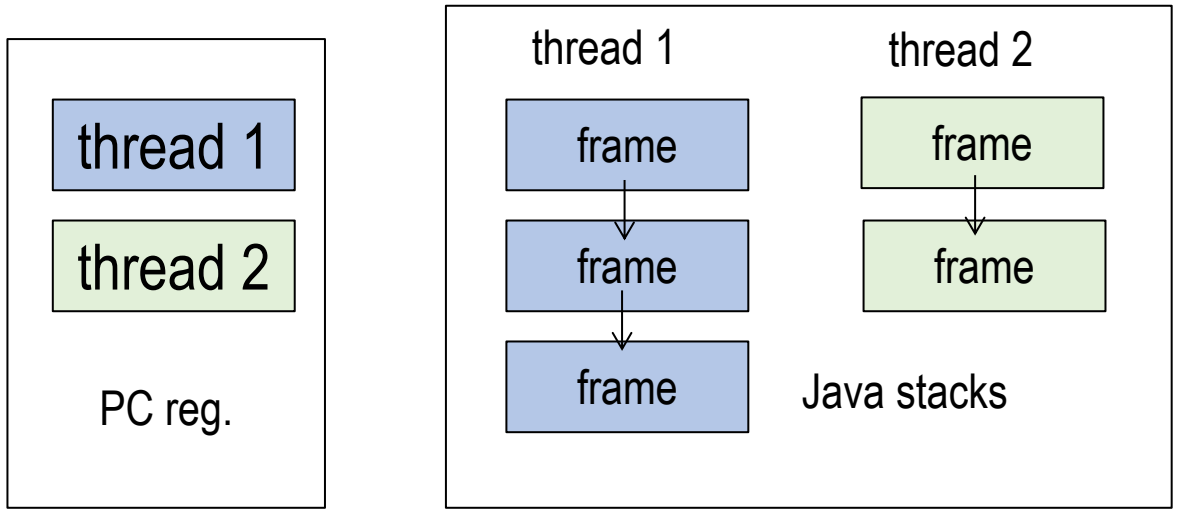


Stack frame


public class A
{
... ...
void f(int x)
{
int i;
for(i=0; i<x; i++)
{
... ...
}
... ...
}
SP→
Inter-
Mediate
Data
Values
i
x
Caller's SP
Caller's FP
FP→
Return PC
Operand
Stack
Local
Variable
Array

In definitiva

- *Method area* e *heap* sono condivisi da tutti i thread in esecuzione nella JVM
- Invece, quando un thread viene creato ottiene un proprio PC e stack
  - gli stack sono composti da frame
  - ogni frame coincide con una invocazione di un metodo



Context switch

- Il cambio contesto tra threads di un programma Java viene effettuato dalla JVM

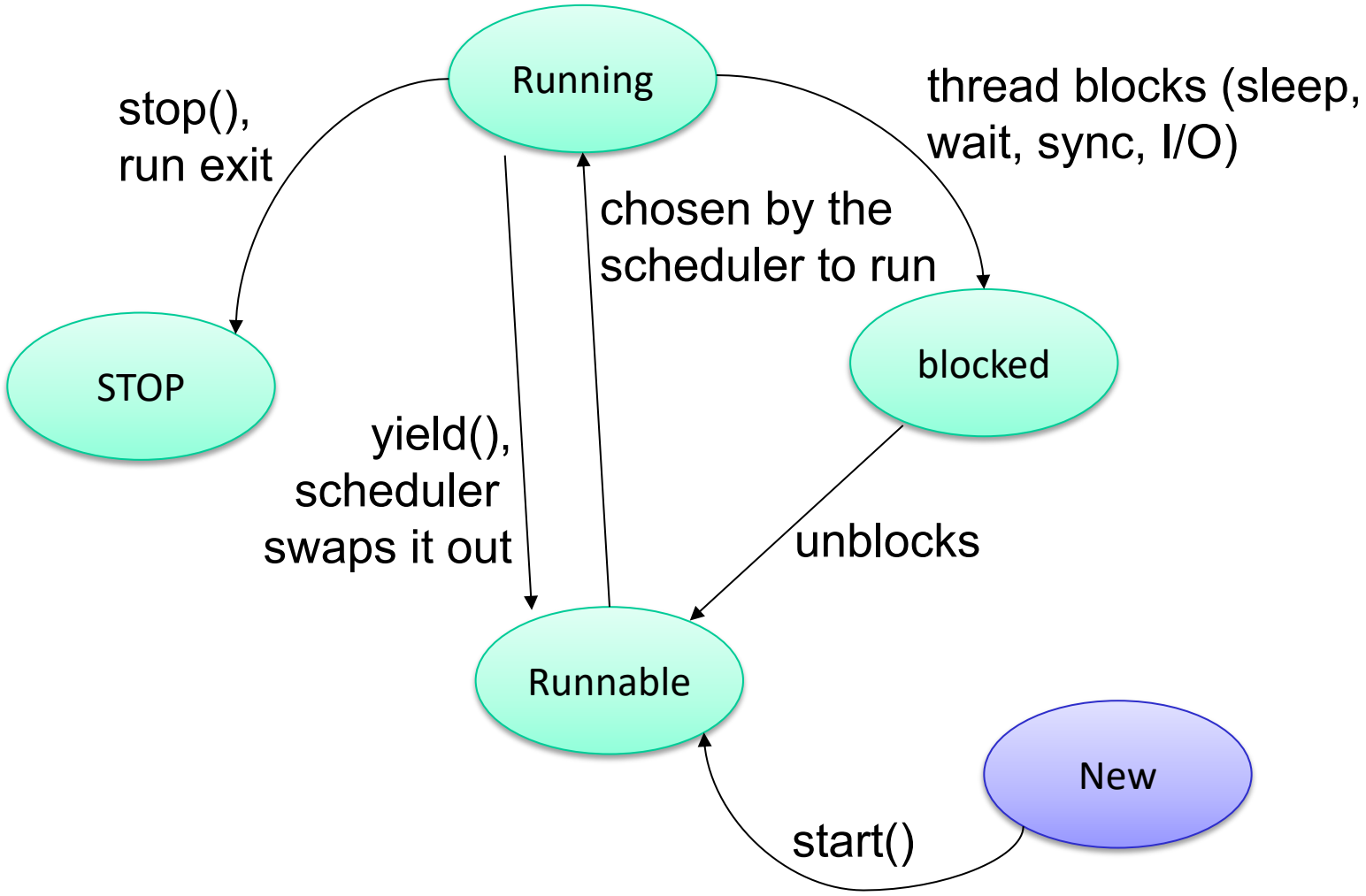
- Il **cambio di contesto** fra due threads richiede tipicamente l'esecuzione di **meno di 100 istruzioni**

Metodi di un Thread

- Il metodo **start()** ha come obiettivo quello di allocare un thread nella JVM, determinando l'invocazione di run()
- Il metodo **run()** definisce il comportamento del thread
- Il metodo **sleep(long millisec)** sospende l'esecuzione del thread per il periodo di tempo specificato
- Il metodo **yield()** causa la sospensione del thread a favore di un altro thread

NOTA: i thread in Java non hanno alcuna funzione di exit. Il thread verrà deallocato solo alla fine del metodo run()

Stati di un Thread



Stati di un Thread

Stato	Blocked	Interruptible	*Description*
Running			Il thread è *running* sul processore.
Runnable (ready-to-run)			Il thread è in attesa del proprio *turno* per il processore.
Sleeping	SI	SI	Il thread diventerà ready-to-run allo scadere del tempo di *sleep* o a seguito di una interruzione.
Waiting	SI	SI	Il thread diventerà ready-to-run allo scadere di un timeout, dopo una *notify*, o a seguito di una interruzione.
Blocked I/O	SI		Il thread diventerà ready-to-run a seguito di una variazione della condizione di I/O attesa.
Blocked synch	SI		Il thread diventerà ready-to-run quando è acquisito il lock su uno statement synchronized.

Scheduling dei thread

Nei sistemi monoprocessore un solo thread alla volta può essere eseguito

I thread in Java hanno una priorità:

- in Java esistono 10 livelli di priorità, compresi fra MIN_PRIORITY (1) e MAX_PRIORITY (10);
  - il valore intermedio è NORM_PRIORITY (5);
- un thread eredita la priorità del thread che lo crea;
- è possibile cambiare la priorità di un thread mediante il metodo **`setPriority(int)`**.

La specifica Java prevede l'algoritmo "fixed priority scheduling", basato sulle priorità dei thread:

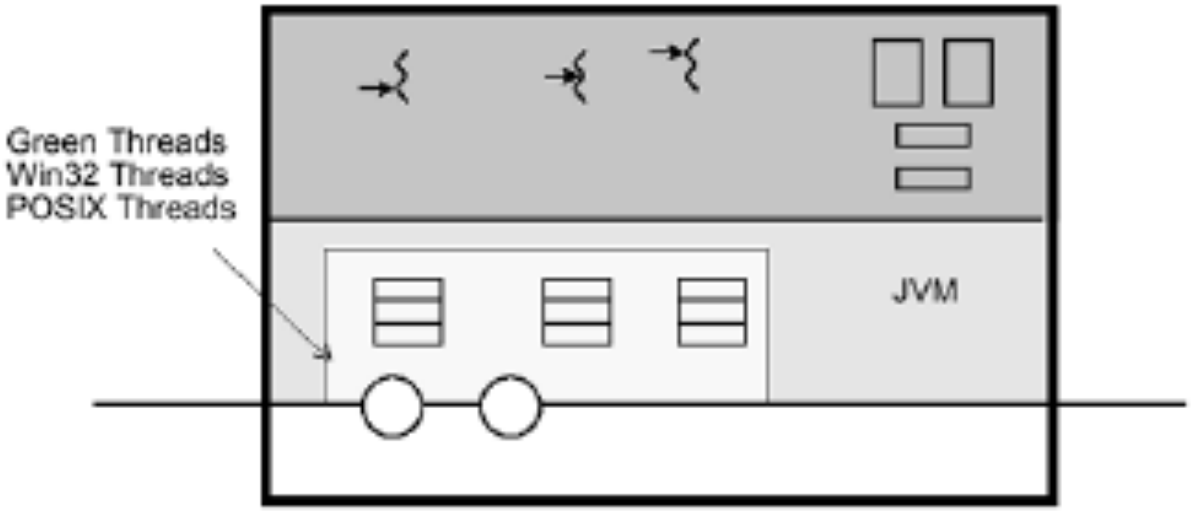
- in generale, i thread a priorità più alta ottengono più tempo di processore rispetto a quelli a priorità più bassa.

Scheduling dei thread

- In ogni momento, se c'è più di un Runnable thread in attesa, il sistema sceglie quello a più alta priorità.
- Se ci sono più Runnable threads con la stessa priorità, il sistema ne sceglie uno operando in modalità **"First Come-First Served"**
- Il thread scelto può continuare l'esecuzione fino a che:
  - Un thread a più alta priorità diviene Runnable;
  - Il thread invoca yield() o il metodo run() finisce;
  - In un **OS "time-slicing"** il suo periodo di CPU è terminato;
  - Passa dallo stato "Running" a "Blocked"
    - sleep() , sync, wait() – condizione di I/O

JVM e threads

- La JVM utilizza una libreria nativa (come la POSIX o Win32) o una propria libreria (“green” threads) per fornire l’infrastruttura su cui girano i thread
- La maggior parte delle implementazioni JVM utilizza i thread del sistema operativo sottostante:
  - Le priorità costituiscono un ‘suggerimento’ per lo scheduler e non devono servire a garantire la correttezza di un programma.



Dovrebbe interessare realmente?

- La specifica Java per lo scheduling è molto "**lasca**". Al fine di ottenere codice portabile non si dovrebbero mai fai troppe assunzioni sul sistema di scheduling sottostante!

Java Priority	Windows Priority (Java 5)	Windows Priority (Java 6)
1	THREAD_PRIORITY_LOWEST	
2		
3	THREAD_PRIORITY_BELOW_NORMAL	
4		
5	THREAD_PRIORITY_NORMAL	THREAD_PRIORITY_NORMAL
6	THREAD_PRIORITY_ABOVE_NORMAL	
7		THREAD_PRIORITY_ABOVE_NORMAL
8	THREAD_PRIORITY_HIGHEST	
9		THREAD_PRIORITY_HIGHEST
10	THREAD_PRIORITY_TIME_CRITICAL	

Good practice: scrivere programmi che funzionino bene a prescindere dal livello di priorità (o non utilizzarle affatto)

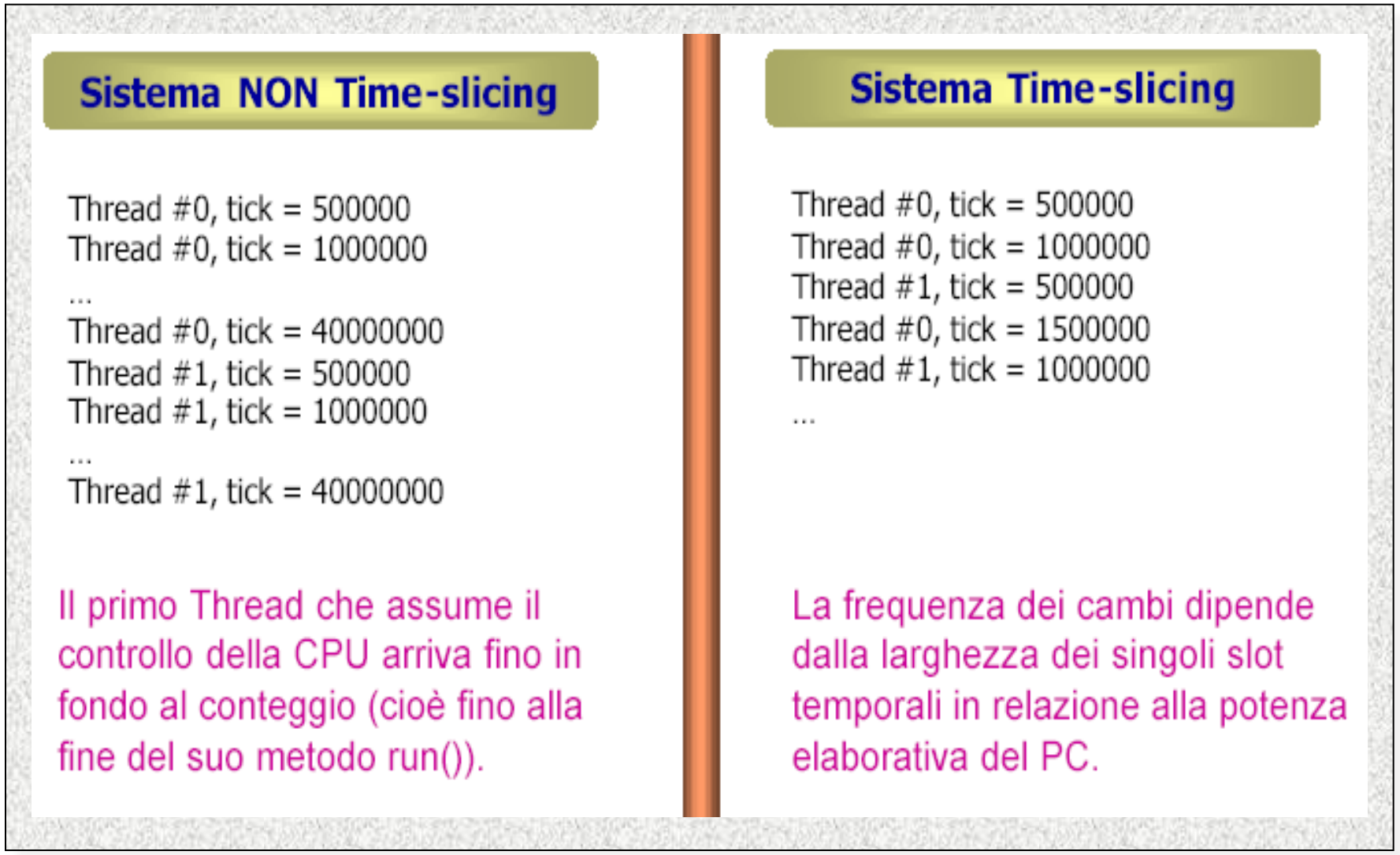
Il problema dei thread egoisti (*selfish*)

Analizziamo un esempio del metodo run di un thread:

```
public class SelfishRunner extends Thread {
  public int tick = 1;
  public void run () {
        while (tick < 4000000) {
          tick++;
        }
  }
}
```

- Una volta in esecuzione, tale thread continua fino alla terminazione del ciclo o fino all'arrivo di un altro thread a priorità maggiore.
- I thread con la stessa priorità di *SelfishRunner* potrebbero aspettare a lungo.

Selfish thread



Selfish thread

- Alcuni sistemi limitano i threads egoisti (selfish) con il time slicing: l'esecuzione di più thread, eseguiti in alternanza solo per uno specifico quanto di tempo (slice).

- Si applica quando più thread con identica priorità hanno diritto ad essere eseguiti e non ci sono altri threads a priorità più elevata.

- NB: **La specifica Java non impone il time slicing!** Infatti, non si può far affidamento sul time slicing dato che i risultati sarebbero differenti da architettura ad architettura.

Una soluzione

- Il programmatore ha a sua disposizione i metodi della classe **Thread** per forzare i threads alla collaborazione:
  - Il metodo yield(), ad esempio, permette al sistema di **cedere la CPU ad un altro thread** eseguibile con la stessa priorità.

Good practice: nel caso peggiore converrebbe assumere che il sistema sia NON time slicing. Se l'elaborazione è fortemente CPU-intesive si potrebbe *occasionalmente* lasciare il controllo (yield) o effetture una sleep

Esempio

- Il sistema alternerà i threads nello stato *Runnable* con un algoritmo round-robin
  - Nel caso di tre threads si avrà un output del tipo:

```
Thread # 0, tick 500000
Thread # 1, tick 500000
Thread # 2, tick 500000
Thread # 0, tick 1000000
Thread # 1, tick 1000000
Thread # 2, tick 1000000
Thread # 0, tick 1500000
Thread # 1, tick 1500000
Thread # 2, tick 1500000
Thread # 0, tick 2000000
Thread # 1, tick 2000000
Thread # 2, tick 2000000
```

La presenza dell'istruzione **yield()** tende a rendere l'esecuzione del programma deterministica, indipendentemente dal sistema operativo (time-slicing o meno) e dalle risorse di calcolo disponibili.

Interruzione di un thread

- Un'interruzione indica che un thread deve interrompere ciò che sta facendo e fare qualcosa altro
- È compito del programmatore indicare **cosa deve fare un thread a seguito di una notifica di interruzione**
- Un thread **solleva un'interruzione su di un altro thread**, invocando il metodo **interrupt() sull'oggetto Thread da interrompere**

Interruzione di un thread

- Alcuni metodi che un thread può invocare durante la sua esecuzione, **sollevano una eccezione di tipo** InterruptedException **quando il metodo `interrupt()` viene invocato durante la loro esecuzione**
- In questo caso, basta gestire tale eccezione opportunamente.

```
try {
    … metodo che può sollevare Interrupted Exception …
} catch (InterruptedException ex) {
    codice gestione eccezione
}
```

Interruzione di un thread

- Se invece il thread non invoca alcun metodo che potrebbe sollevare eccezioni di tipo InterruptedException, allora può periodicamente invocare il metodo Thread.interrupted(), che **ritorna true se sul thread è stato invocato il metodo interrupt()**:

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if( Thread.interrupted() ) {
            //thread interrotto…
            return;
        }
}
```

Oppure, in alcune circostanze, ha senso sollevare una eccezione se interrupted() restituisce true:

```
    if( Thread.interrupted()) {
            throw new InterruptedException();
        }
```

Interruzione di un thread

- NOTA: Thread.interrupted() verifica se il thread corrente è stato interrotto. Il flag "*interrupted status*" del thread viene resettato a seguito dell'invocazione.

```
public static boolean interrupted()
```

- isInterrupted() verifica se il thread è stato interrotto. Il flag "*interrupted status*" non viene modificato a seguito dell'invocazione.

```
public boolean isInterrupted()
```

Interruzione di un thread

- Oltre a interrupt(), la classe Thread presenta tre ulteriori metodi per l'interruzione di un thread:
  - `suspend()` – sospende un thread;
  - `resume()` – riattiva un thread precedentemente interrotto da `suspend()`;
  - `stop()` – ferma un thread e lo uccide.

Interruzione di un thread

- Questi metodi sono **deprecati**, ovvero si consiglia ai programmatori di non adoperarli.

- L'uso di tali metodi è rischioso e comporta notevoli complicazioni:
  - E.g., **un thread può essere interrotto prima di poter rilasciare una risorsa**, impedendo così agli altri di potervi accedere e generando un deadlock difficilmente risolvibile.

- Per questa ragione, si **assume che un thread si interrompe e muore solo a conclusione del suo metodo run()**

Sleep e Join

- Un metodo della classe Thread comunemente adoperato dai programmatori Java è sleep(), **che pone in attesa (o stato dormiente) un thread** per il numero di millisecondi che viene specificato dall'utente.

```
try {
    Thread.sleep(4000);
} catch (InterruptedException ex) {
    System.out.println(ex);
}
```

- Quando un thread padre genera vari thread figli, potrebbe essere necessario attendere la loro conclusione, prima di procedere. Allo scopo è possibile usare il metodo join().

Sleep e Join

```
Thread[] threads = new Thread[numOfThreads];
for (int i = 0; i < threads.length; i++) {
       threads[i] = new MyThread();
       threads[i].start();
}
for (int i = 0; i < threads.length; i++) {
       try {
          threads[i].join();
       } catch (InterruptedException ignore) {}
}
```

Sleep e Join

```
Thread[] threads = new Thread[numOfThreads];
for (int i = 0; i < threads.length; i++) {
       threads[i] = new MyThread();
       threads[i].start();
}
for (int i = 0; i < threads.length; i++) {
       try {
          threads[i].join();
       } catch (InterruptedException ignore) {}
}
```

Il processo padre attende fino alla conclusione da parte di threads[i] della sua esecuzione.

ThreadGroup

- All'atto della loro creazione, i thread posso essere raggruppati per mezzo di ThreadGroup, così da poterli controllare congiuntamente come se fossero **una singola entità**.
- Ogni thread appartiene sempre a un gruppo; **se non specificato, un thread appartiene a un gruppo di default chiamato main**.