



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

---

# Programmazione Concorrente e Distribuita Thread e MultiThread Parte A

Luigi Lavazza

Dipartimento di Scienze Teoriche e Applicate

[luigi.lavazza@uninsubria.it](mailto: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.

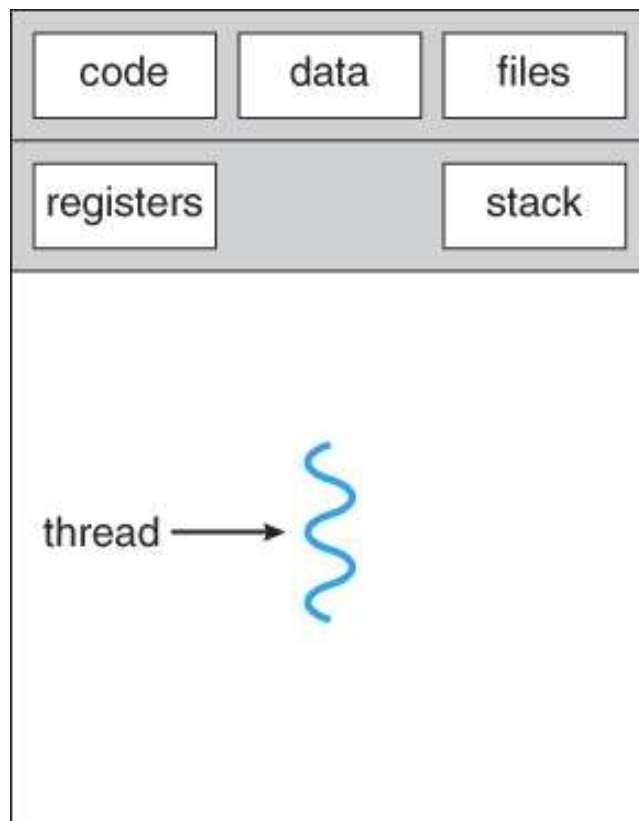
# Differenze tra processo e thread

---

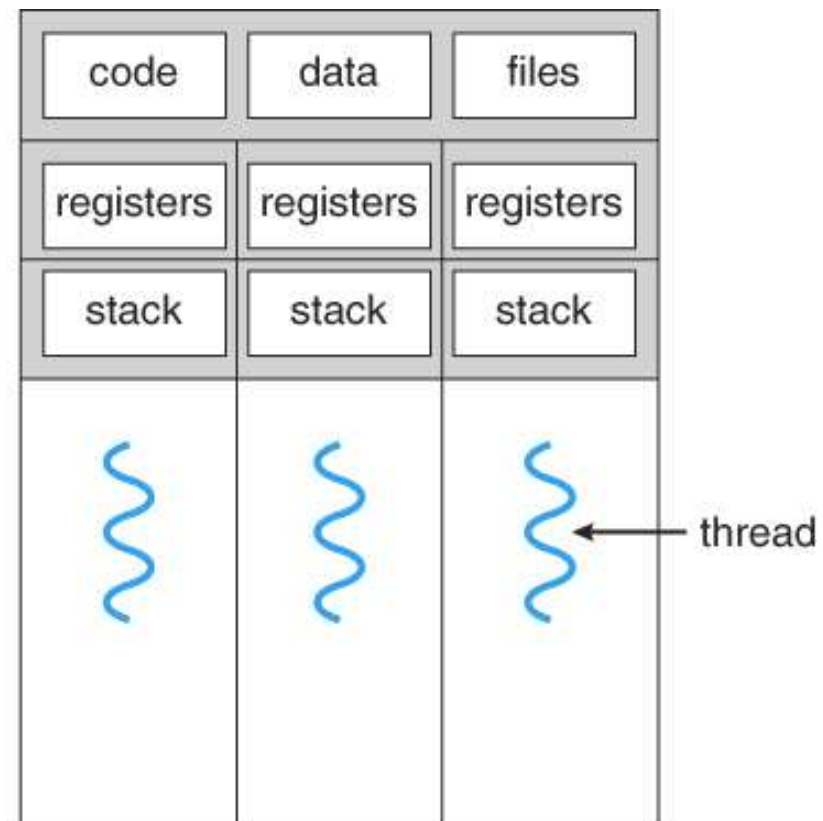
- Quando i processi hanno il proprio spazio degli indirizzi, vengono chiamati processi pesanti o semplicemente processi.
  - ▶ I processi possono comunicare attraverso meccanismi messi a disposizione dal sistema operativo
  - ▶ Le comunicazioni passano dal S.O.
- Quando i processi condividono lo stesso spazio degli indirizzi, allora vengono chiamati processi leggeri o thread.
  - ▶ I thread comunicano senza passare dal S.O. Spesso semplicemente attraverso memoria condivisa

# 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.



single-threaded process



multithreaded process

## 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]

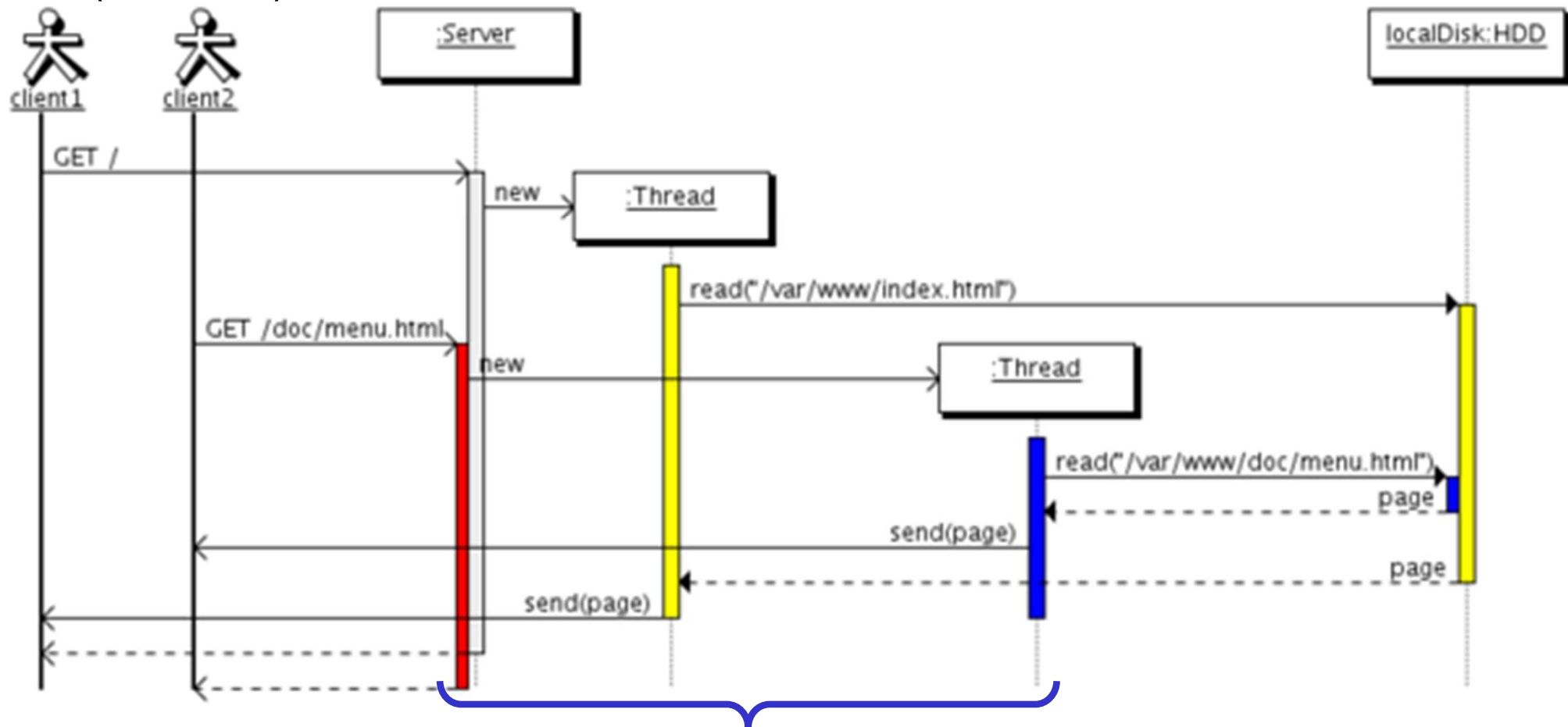
Output:

**Thread corrente: Thread[main,5,main]**

**Dopo cambio nome: Thread[Mio Thread,5,main]**

# Programmazione concorrente

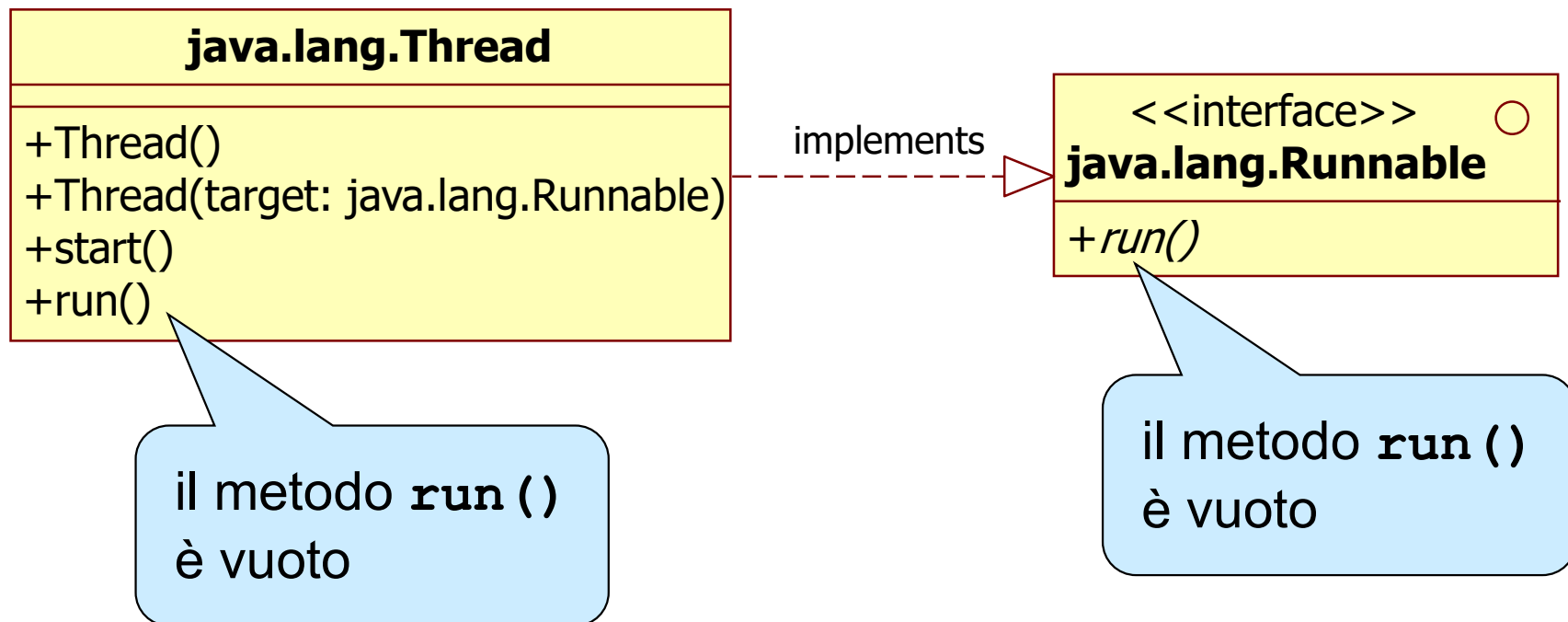
- Con il termine **programmazione concorrente** si indica la pratica di implementare dei programmi che contengano **più flussi di esecuzione** (Threads)



**concorrenti**

# La classe principale per i Thread in Java

- La classe `java.lang.Thread`





# La classe principale per i Thread in Java

---

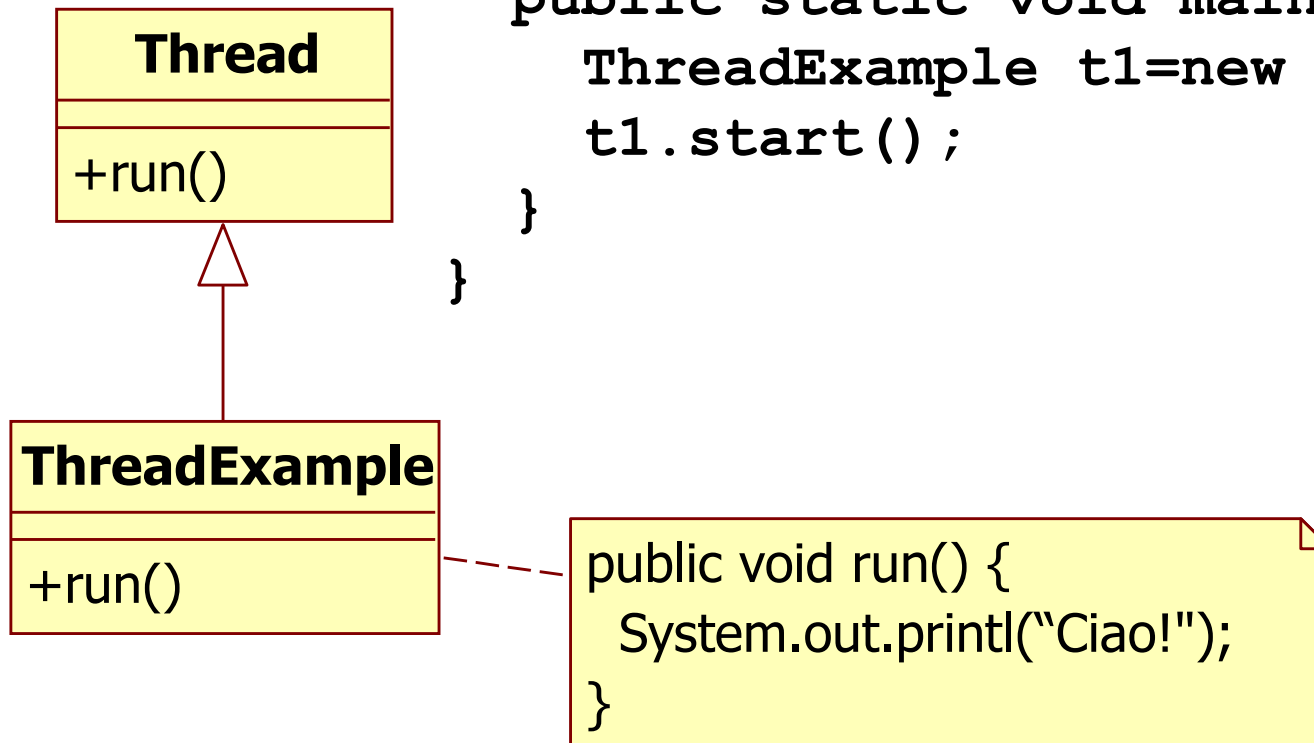
Il modo più semplice per creare ed eseguire un Thread è

1. Estendere la classe `java.lang.Thread` (che contiene un metodo `run()` vuoto)
2. Ridefinire (override) il metodo `run()` nella sottoclasse
  - ▶ Il codice eseguito dal thread è incluso nel 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. Chiamare il metodo `start()` su questa istanza
  - ▶ NB: spesso si mette `start()` nel metodo 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("Ciao!");  
    }  
    public static void main(String arg[]){  
        ThreadExample t1=new ThreadExample();  
        t1.start();  
    }  
}
```



# Estensione di Thread: start «automatica»

---

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

La creazione di una nuova istanza di **ThreadExample** implica l'esecuzione del costruttore, che contiene **start**: il thread viene immediatamente eseguito.



# Creazione di tanti thread


---

```
public class ThreadExample extends Thread {  
    ThreadExample() {  
        this.start();  
    }  
    public void run() {  
        System.out.println("Ciao!");  
    }  
    public static void main(String arg[]){  
        ThreadExample t1=new ThreadExample();  
        ThreadExample t2=new ThreadExample();  
        new ThreadExample();  
    }  
}
```



# Creazione di tanti thread

```
public class ThreadExample extends Thread {  
    ThreadExample() {  
        this.start();  
    }  
    public void run() {  
        System.out.println("Ciao!");  
    }  
    public static void main(String arg[]) {  
        ThreadExample t1=new ThreadExample();  
        ThreadExample t2=new ThreadExample();  
        new ThreadExample();  
    }  
}
```



Quando l'esecuzione arriva qui, quanti thread ci sono in esecuzione?

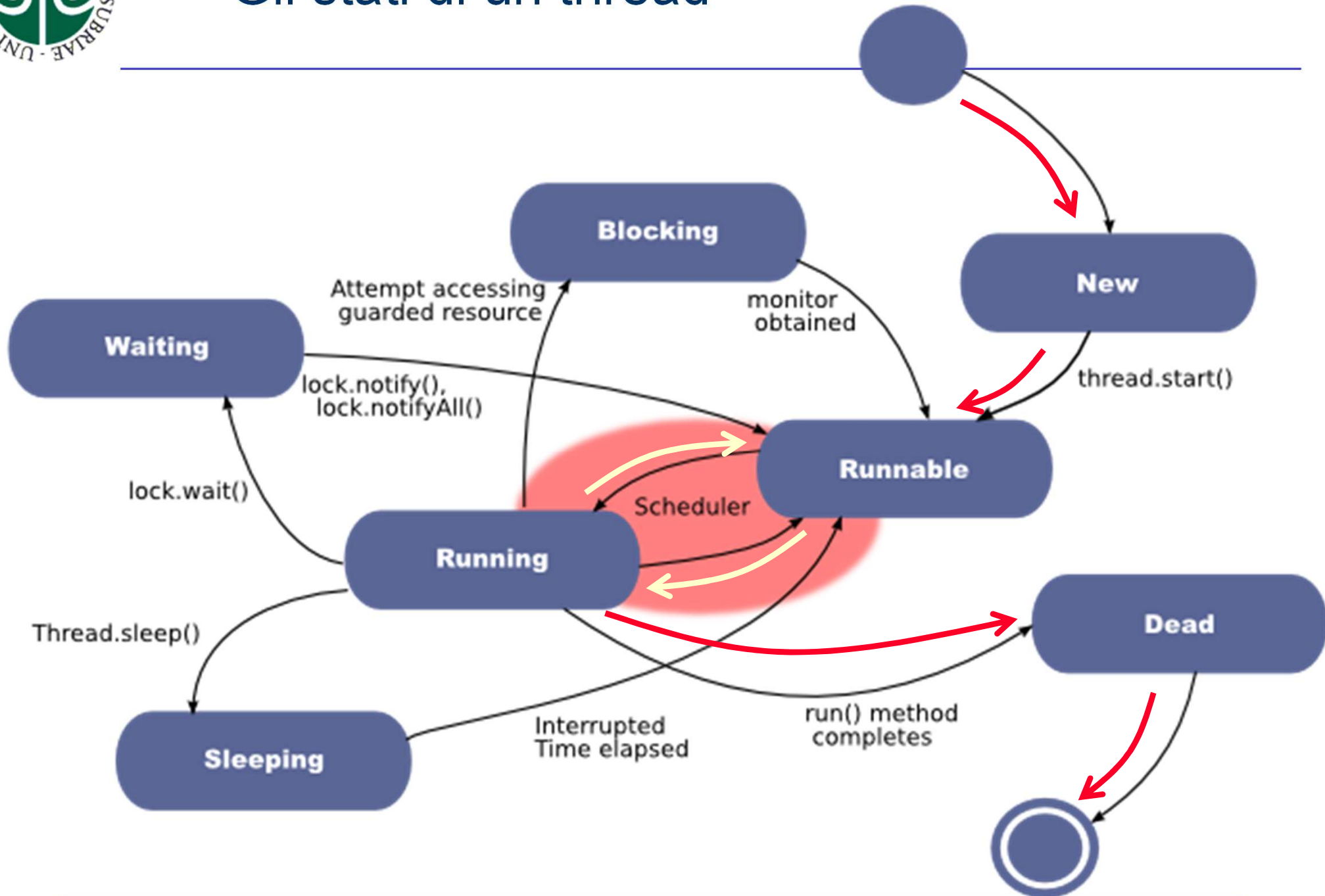


# Il metodo `run ()`

---

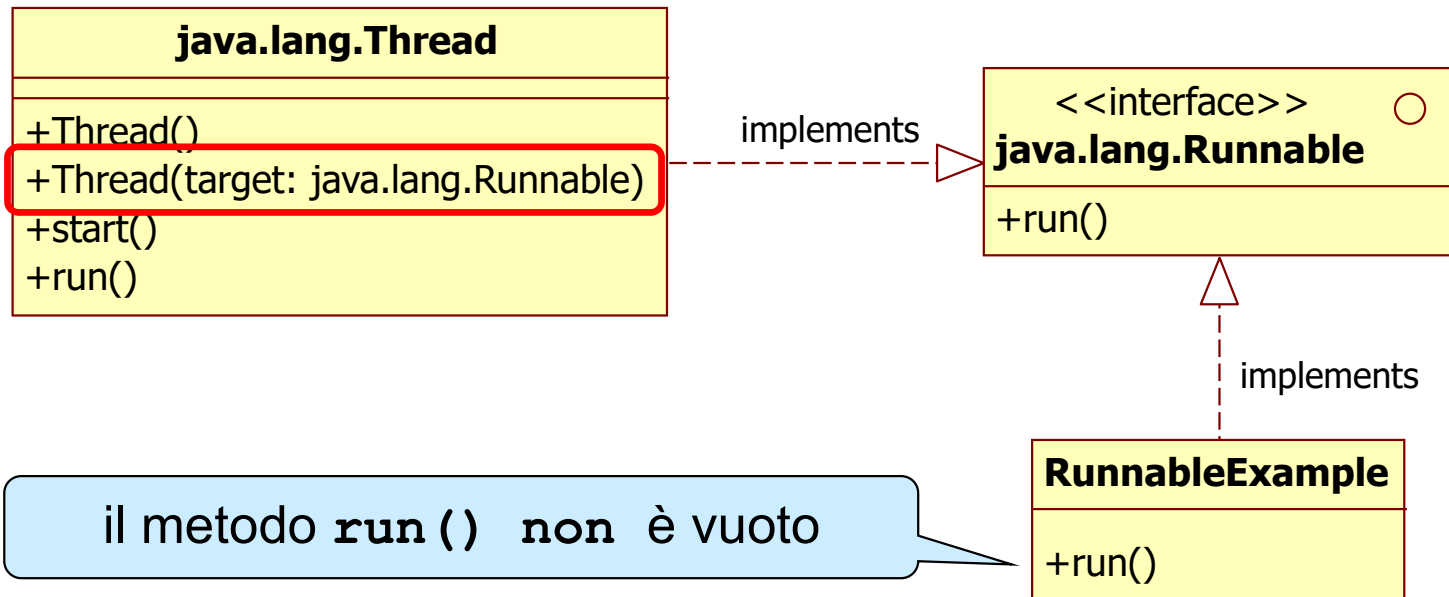
- Il metodo `run ()` costituisce l'entry point del thread:
  - ▶ Un thread è considerato **alive** finché il metodo `run ()` non ritorna
  - ▶ Quando `run ()` ritorna, il thread è considerato **dead**
- Una volta che un thread è “morto” non può essere rieseguito (pena un'eccezione `IllegalThreadStateException`)
- Se si vuole ri-eseguire il thread, 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 classe implementazione di **Runnable** dotata di un metodo **run()** significativo
  2. Creare un'istanza di questa classe
  3. Istanziare un nuovo **Thread** (cioè un'istanza della classe **Thread**) passando al costruttore l'istanza della classe che implementa **Runnable**
  4. Chiamare il metodo **start()** sull'istanza di **Thread**





# Esempio usando Runnable

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

In questo modo **t1** contiene un riferimento a **re**

Quando il metodo **t1.start()** viene chiamato, si esegue il metodo **run()** fornito da **re**





# Creazione di tanti thread usando Runnable

```
public class RunnableExample implements Runnable{
    public void run() {
        System.out.println("Ciao!");
    }
    public static void main(String arg[]){
        RunnableExample re=new RunnableExample();
        Thread t1=new Thread(re);
        t1.start();
        Thread t2=new Thread(re);
        t2.start();
    }
}
```

**re** si può usare più volte, per creare più thread che eseguono lo stesso codice.

# 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("Ciao!");
    }
    public static void main(String arg[]){
        ProceduralExample pe=new ProceduralExample();
        pe.run();
    }
}

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

# Run e start

- Nel programma sequenziale metodo `run()` può essere chiamato direttamente più volte

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


Output:  
Ciao!  
Ciao!

# Run e start

- Nel programma concorrente, 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.println("Ciao!");  
    }  
}
```

```
public static void main(String args []) {  
    RunnableExample re=new RunnableExample();  
    Thread t1=new Thread(re);  
    t1.start();  
    t1.start();  
}
```



```
Ciao!Exception in thread "main"  
java.lang.IllegalThreadStateException  
    at java.base/java.lang.Thread.start(Thread.java:794)  
    at RunnableExample.main(RunnableExample.java:10)
```



# Programma Concorrente

---

```
public class ThreadExample extends Thread{
    public void run() {
        System.out.println("Ciao! sono "+
                           Thread.currentThread() );
    }
    public static void main(String args []) {
        ThreadExample t1=new ThreadExample();
        ThreadExample t2=new ThreadExample();
        ThreadExample t3=new ThreadExample();
        t1.start();
        t2.start();
        t3.start();
        System.out.println("Main: completato");
    }
}
```

# Esecuzioni

```
Main: completato  
Ciao! sono Thread[Thread-0,5,main]  
Ciao! sono Thread[Thread-1,5,main]  
Ciao! sono Thread[Thread-2,5,main]
```

```
Ciao! sono Thread[Thread-0,5,main]  
Ciao! sono Thread[Thread-2,5,main]  
Main: completato  
Ciao! sono Thread[Thread-1,5,main]
```

Le esecuzioni  
possibili sono tante!  
**Nondeterminismo!**

```
Ciao! sono Thread[Thread-0,5,main]  
Ciao! sono Thread[Thread-2,5,main]  
Ciao! sono Thread[Thread-1,5,main]  
Main: completato
```

# Flusso di controllo

---

- Un programma termina quando tutti i suoi thread terminano.
  - ▶ Se ci sono thread in esecuzione, il programma non termina
- Eccezione: i thread «daemon»
- Un thread daemon fornisce un servizio generale e non essenziale in background mentre il programma esegue altre operazioni.
- Un programma termina quando tutti i suoi thread non daemon terminano.
  - ▶ Se ci sono thread non demoni in esecuzione, il programma non termina
  - ▶ Se sono rimasti solo thread daemon in esecuzione, il programma termina





# 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 `setDaemon(true)` nel costruttore di un thread.

```
public class DaemonExample extends Thread{  
    public DaemonExample(boolean isDaemon) {  
        setDaemon (isDaemon) ;  
    }  
    public void run() {  
        int count=0;  
        while(true) {  
            System.out.println("Ciao "+count++);  
            try {  
                Thread.sleep(2000);  
            } catch (InterruptedException e) { }  
        }  
    }  
}
```

Quando il parametro è vero il thread è un daemon thread.

Quando il parametro è falso, il thread non è daemon, è un thread normale.

NB: il run contiene un loop infinito.  
Si esegue una iterazione ogni 2 secondi.



# Thread daemon

```
public static void main(String args []) {  
    DaemonExample t1=new DaemonExample(true);  
    t1.start();  
    try {  
        Thread.sleep(7000);  
    } catch (InterruptedException e) { }  
    System.out.println("Main thread terminates.");  
}
```

Il thread è deamon

```
Output:  
Ciao 0  
Ciao 1  
Ciao 2  
Ciao 3  
Main thread terminates.  
[il programma ha terminato]
```

# Thread daemon

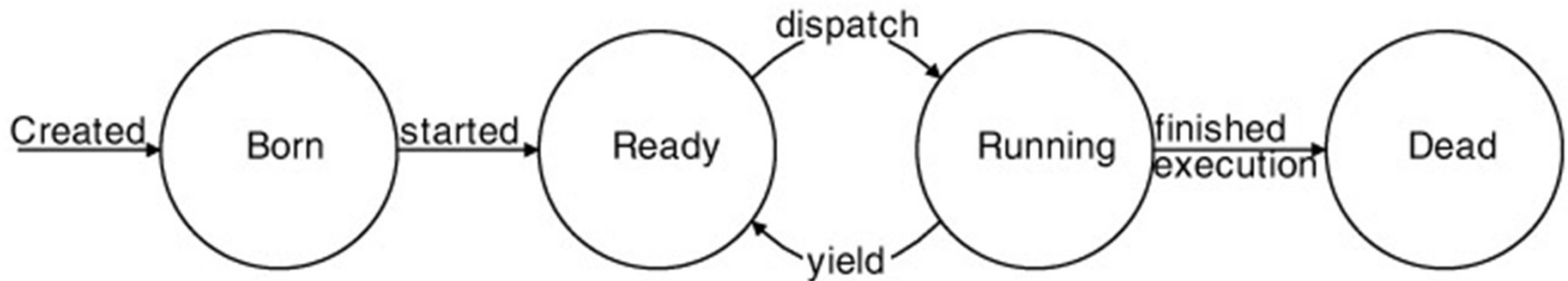
```
public static void main(String args []) {  
    DaemonExample t1=new DaemonExample(false);  
    t1.start();  
    try {  
        Thread.sleep(7000);  
    } catch (InterruptedException e) { }  
    System.out.println("Main thread terminates.");  
}
```

Il thread **non** è deamon

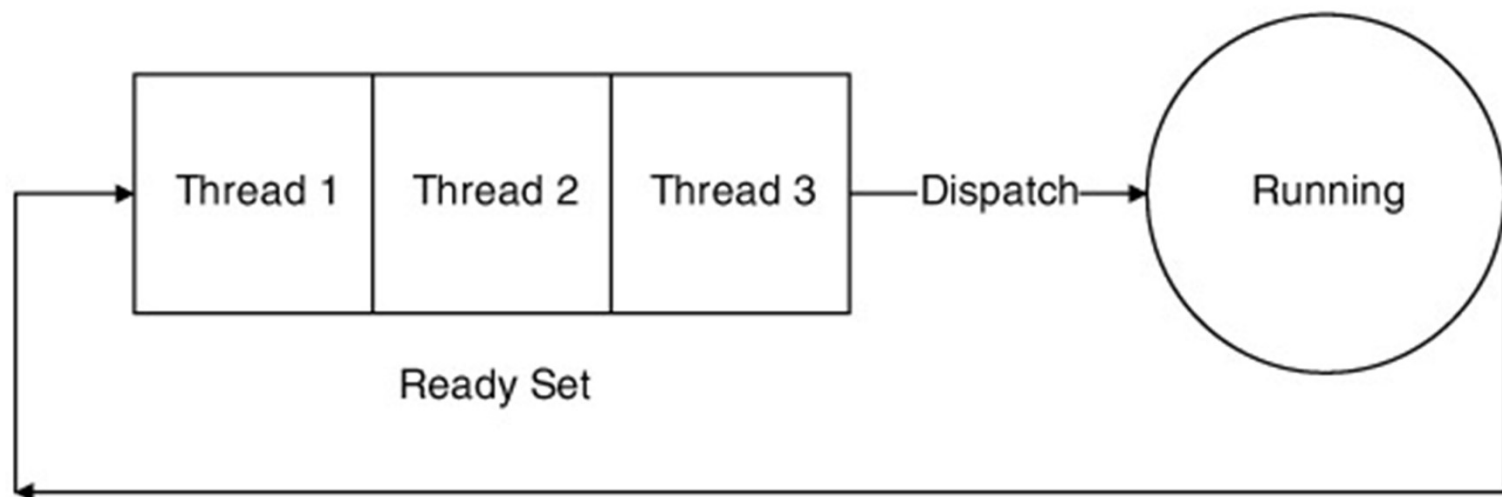
Output:

```
Ciao 0  
Ciao 1  
Ciao 2  
Ciao 3  
Main thread terminates.  
Ciao 4  
Ciao 5  
Ciao 6  
...
```

# 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 PreemptiveTest {  
    public static void main(String args []) {  
        System.out.println("Main: inizio");  
        Thread t1 = new BusyThread();  
        t1.start();  
        try {  
            Thread.sleep(100);  
        } catch (InterruptedException e) {}  
        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.

# Output

---

```
Main: inizio  
ciao  
ciao  
ciao  
ciao  
ciao
```

- Lo scheduler è preemptive!



# Programma procedurale con due chiamate a run ()

```
public class Procedural {  
    private int myNum;  
    public Procedural(int n) { myNum=n; }  
    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) { }  
    }  
    public static void main(String args []) {  
        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) { }  
    }  
}
```

Output prodotto  
(**sempre!**):

In run, myNum=1  
In run, myNum=1  
In run, myNum=2  
In run, myNum=2  
In main  
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);
        a.start();
        b.start();
        try { Thread.sleep((int) (100* Math.random()));
            System.out.println("in main ");
            Thread.sleep((int) (100* Math.random()));
            System.out.println("in main ");
        } catch (InterruptedException e) { }
    }
    public void run ( ) {
        try { Thread.sleep((int) (100* Math.random()));
            System.out.println("in run "+ myNum);
            Thread.sleep((int) (100* 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	<i>fine</i>	
dead	ready → esec	sleep		
dead	esec	sleep → ready		
dead	esec → dead	ready → esec	<i>fine</i>	
dead	dead	esec → dead	<i>fine</i>	



# 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.