

# **Programmazione Orientata agli Oggetti**

---

Introduzione Java Thread

# Sommario

- Introduzione
  - L'era multi-core
- Concetto di *Thread*
- `java.lang.Thread`
- Creazione di thread
  - `java.lang.Thread.start()`
- Specifica del codice da eseguire
  - Interface *Runnable*
- Terminazione
- Join tra Thread
  - `java.lang.Thread.join()`
- Decomposizione parallela
- Speed-up

# L'Era *Multi-Core*

- Siamo, oramai conclamatamente, nell'era della diffusione di massa di CPU ad architettura multi-core

```
public class EraMultiCoreTest {  
    @Test  
    public void testPossiedoUnMultiCoreEmagariNonLoSapevo() {  
        assertTrue(Runtime.getRuntime().availableProcessors() > 0);  
    }  
}
```

- Non sono solo più macchine server, ma anche
  - laptop
  - desktop
  - smartphone
  - smartTVtutti dotate di CPU multi-core

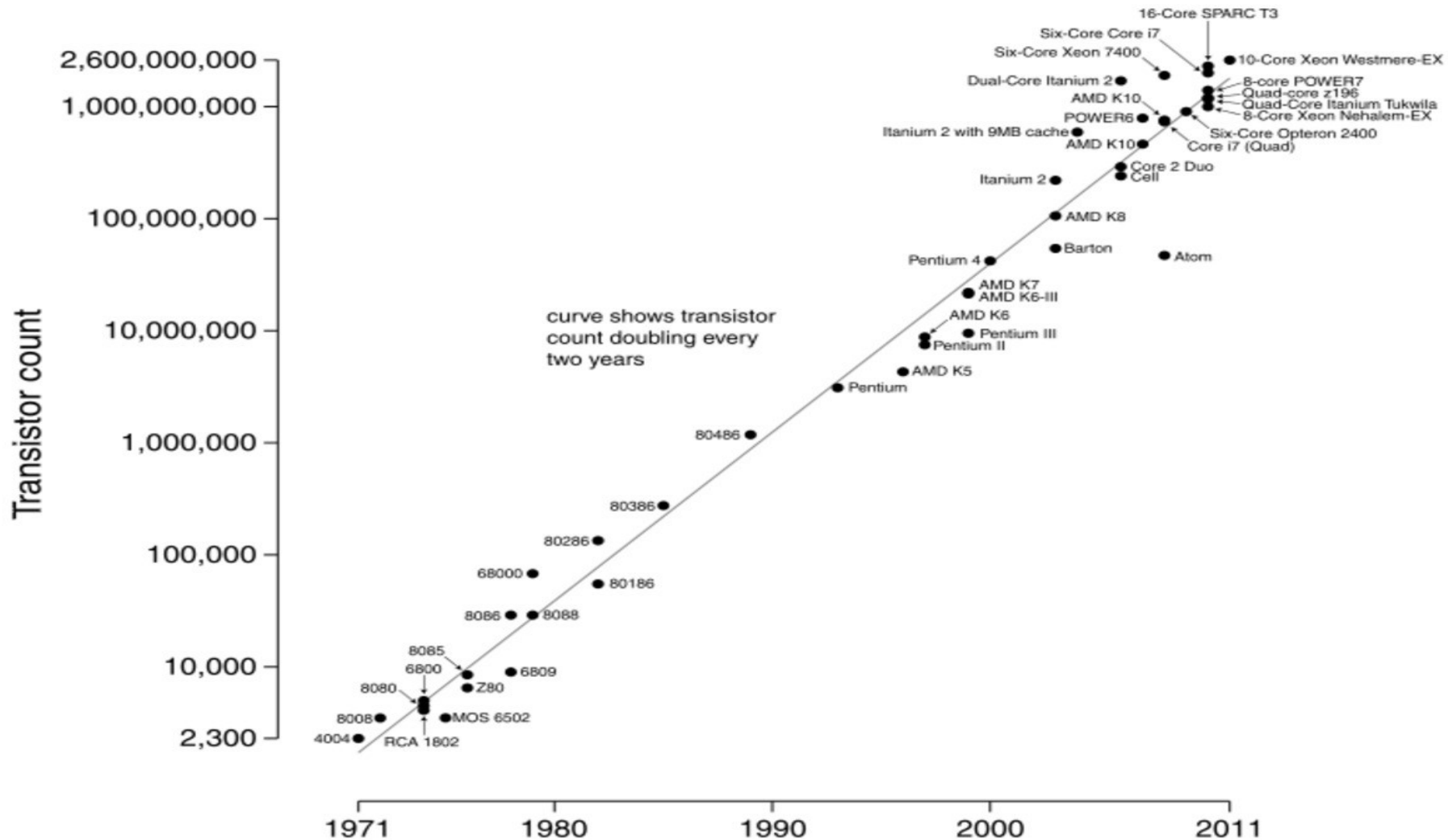
- Perché?

# L'Insaziabile Fame di Prestazioni

- Software ed hardware si rincorrono da sempre
  - Hardware sempre più potente...
  - ...che permette di sviluppare software sempre più esigente...
  - ...che richiede hardware sempre più potente per funzionare↑
- Continuo, progressivo e quasi costante sinora, aumento delle prestazioni
- C'è stata un'era, ormai esaurita, in cui un ruolo chiave per l'aumento delle prestazioni veniva ricoperto dagli incrementi della cosiddetta *frequenza di clock*
  - banalizzando, il ritmo a cui funziona una macchina discreta come la CPU
- Ad un aumento lineare della frequenza di funzionamento di una CPU si poteva avere, in prima approssimazione, un aumento lineare delle sue prestazioni

# Legge di Moore

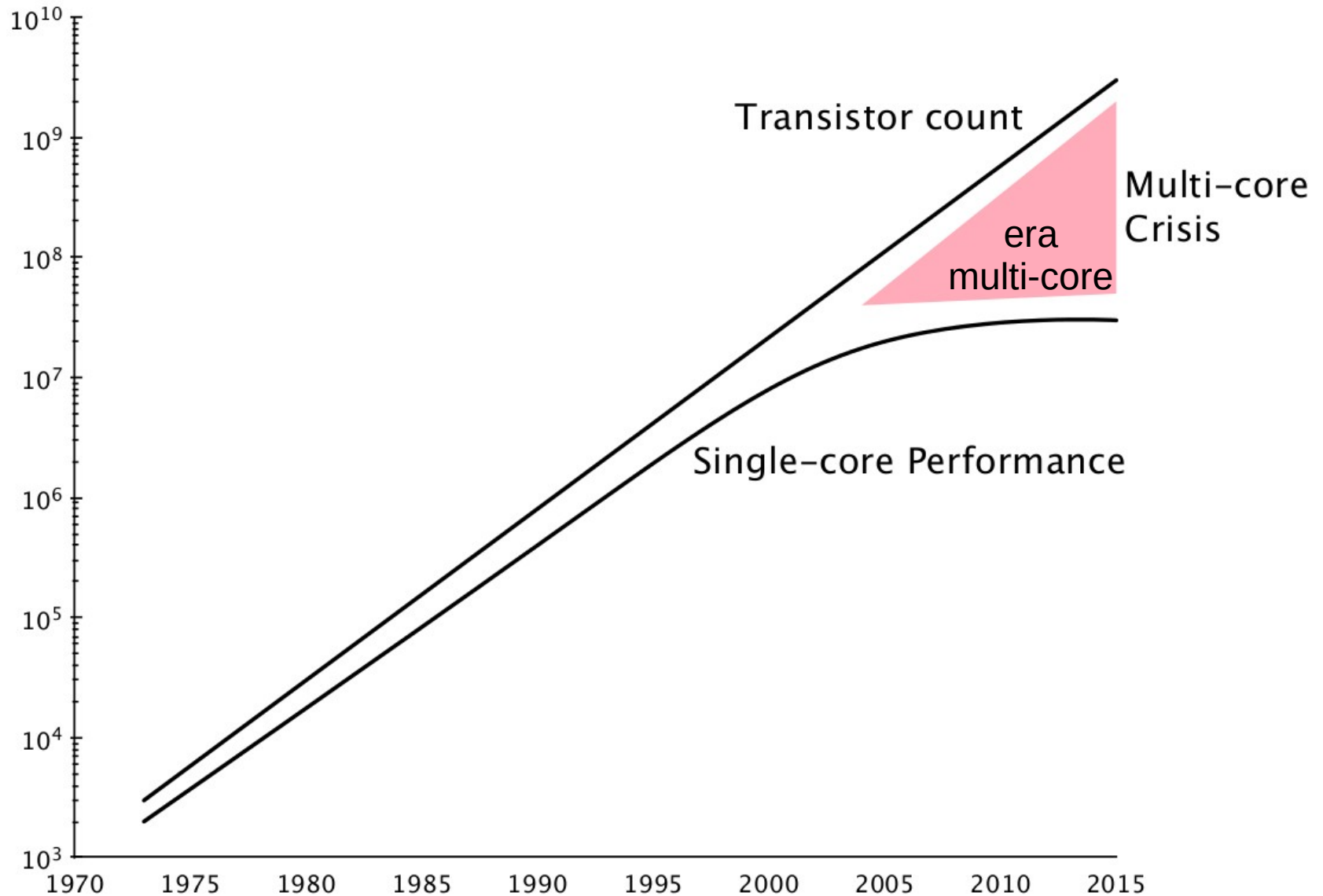
Microprocessor Transistor Counts 1971-2011 & Moore's Law



Il numero di transistor e le prestazioni di una CPU raddoppia[vano] approssimativamente ogni due anni

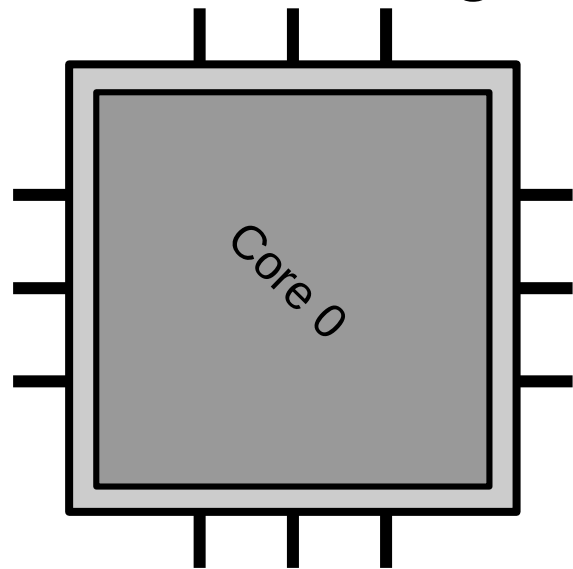
# Fine della Legge di Moore

- ✓ All'aumento della frequenza purtroppo corrisponde anche un aumento *più che lineare* della quantità di calore generato

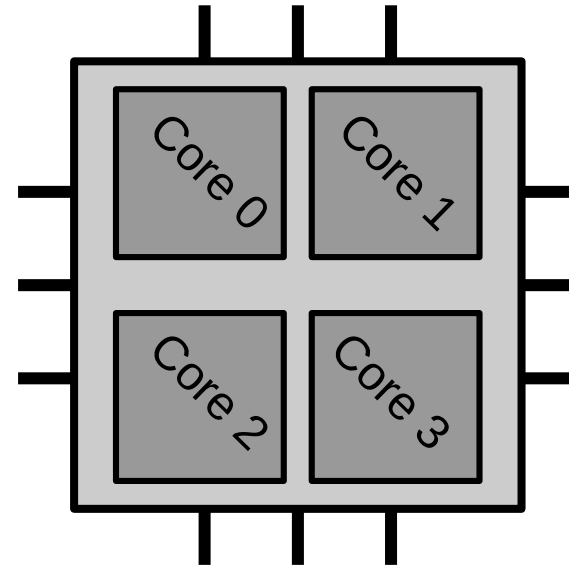


# Multi-Core

Architettura single-core



Architettura multi-core



- Piuttosto che aumentare la frequenza di clock, conviene oramai creare nuove unità di processamento
  - problemi legati alla dissipazione del calore sempre più onerosi
- Tali unità di processamento:
  - sono dette *core* e risiedono sullo stesso circuito integrato
  - possono eseguire flussi di esecuzione indipendenti
    - ✓ come un'intera CPU mono-core di vecchia generazione

# Programmazione di Architetture Multi-Core

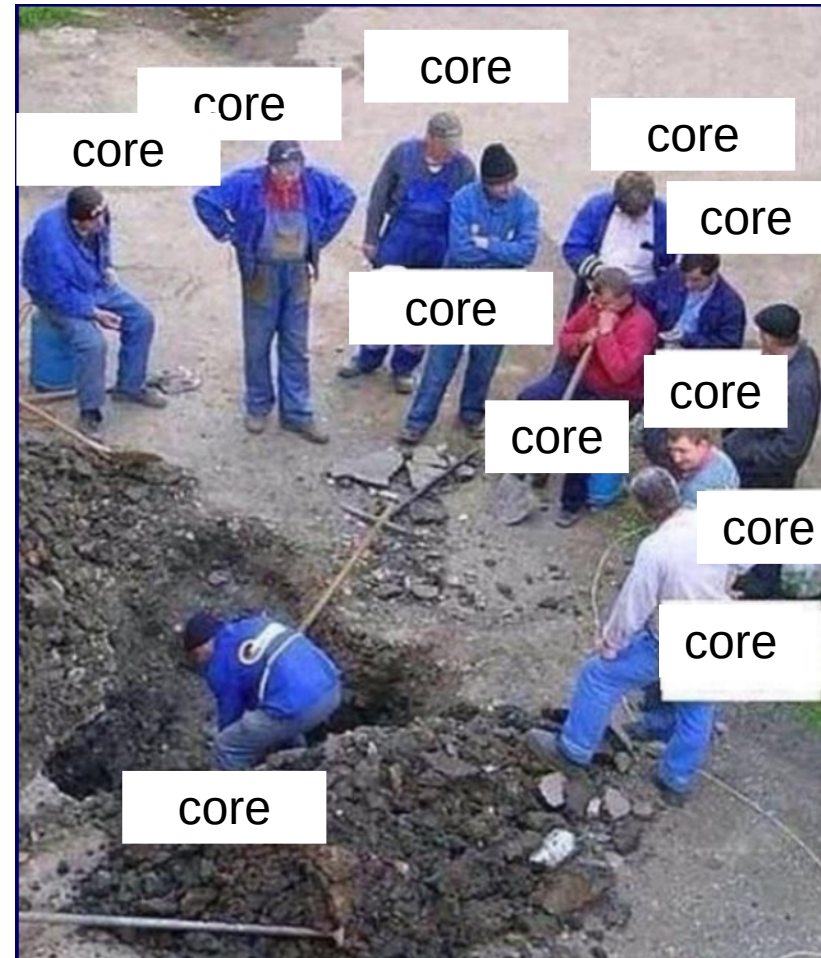
- Non è però così semplice ed immediato aumentare le prestazioni di un *programma* con le architetture multi-core
  - Perché bisogna dire ad ogni core come impegnare il proprio tempo per metterla in condizione di contribuire
  - In presenza di un solo core, questo problema, semplicemente, non si poneva
- Difatti: cosa accade se eseguiamo un programma di elaborazione intensiva, pensato per un'architettura tradizionale a singolo core su un'architettura multi-core e di nuova generazione?



# *Uno Lavora e 10 Guardano!*

- Il core che esegue il nostro programma risulta completamente utilizzato
- ✓ Ma tutti gli altri non sanno assolutamente cosa fare!
- ✓ Per beneficiare delle nuove architetture, il programmatore è chiamato a specificare cosa devono fare tutti i core, non solo uno come avveniva già in precedenza
- ✓ Radicale cambio di mentalità

*Programmazione Concorrente*  
(>>>)



# Decomposizione Parallela

- Per potere sfruttare le nuove architetture multi-core, il programmatore, ad esempio, è chiamato a specificare:
  - come suddividere il lavoro da compiere
  - come assegnare la porzione di lavoro a ciascuno dei core disponibili
  - come risolvere ciascuna parte del lavoro per calcolare il corrispondente risultato parziale
  - come ricomporre tutti i risultati parziali per ottenere quello complessivo
- La *Decomposizione Parallela* di un algoritmo non è sempre semplice e nemmeno sempre possibile
- Alcuni problemi si prestano bene ad essere decomposti parallelamente; altri assolutamente no

# Programmazione Concorrente

- La programmazione concorrente è un argomento vasto, che merita interi corsi
  - Come “Programmazione Concorrente” (PC>>)
- Lo scopo dichiarato di questa lezione è solo una prima introduzione alle problematiche di base e soprattutto alle opportunità della programmazione concorrente
- Per far acquisire consapevolezza dell’esistenza di questo importante ed attualissimo tema
- Per semplicità, ci concentriamo solo su alcuni problemi piuttosto particolari con un ottimo rapporto costi/benefici
- Ovvero, i *Problemi Decomponibili Parallelamente*
  - *facilmente* decomponibili
  - aumento di prestazioni *significativo*

# Programmazione Concorrente in Java

- La piattaforma Java, proprio perché nata originariamente per supporto lo sviluppo di applicazioni *embedded* da sempre ha un supporto nativo alla programmazione concorrente
  - In generale la programmazione concorrente è *molto più difficile* di quella tradizionale
    - anche detta *seriale, sequenziale* (o *mono-thread*)
  - Invece in alcuni casi particolari, appunto in presenza di molteplici attività concorrenti (sistemi embedded, GUI), può risultare invece un modello di programmazione addirittura più naturale
- Per queste esigenze la piattaforma Java si basa sul concetto di *Thread*

# Java Thread

- *Thread*: flusso di esecuzione (f.d.e.) indipendente che condivide la memoria con altri thread
  - Spesso usato in contrapposizione con il termine *Processo*
  - *Processo*: f.d.e. che non condivide la memoria con altri
- Tutti i nostri programmi fanno già uso di thread
- Per eseguire il nostro codice, la JVM ci assegna un thread che si chiama *Thread Main*

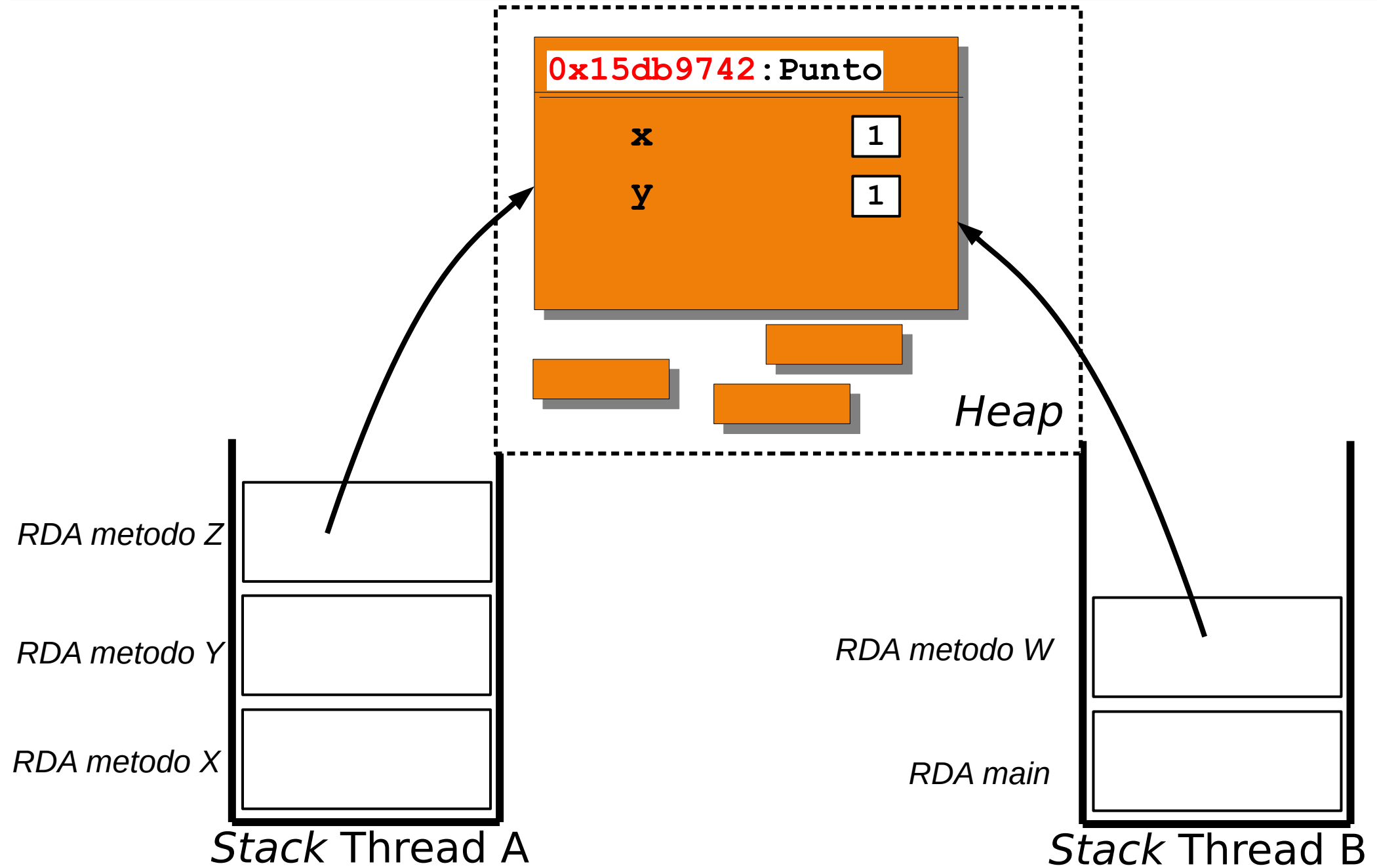
# Java *Thread Main*

- Per eseguire il nostro codice, la JVM ci assegna un thread che si chiama *Thread Main*
  - Nome non causale: si occupa di eseguire il metodo `main()` da cui parte l'esecuzione
- In realtà il numero di thread creati allo start-up di una JVM è via via aumentato all'aumentare della versione Java
  - Tra quelli nelle versioni più moderne segnaliamo:
    - il *Garbage Collector* che impegna uno o più thread
    - le GUI (*JavaFx*, *Swing*) usano un thread separato per gestire gli eventi grafici (>>)
    - Un pool di thread (>>) a supporto di lavoro intensivo (Java 8+)

# Thread e Memoria Condivisa

- I thread sono flussi di esecuzione indipendenti
  - ciascuno necessita di un proprio *Stack* autonomo a supporto del mantenimento e della gestione del proprio stato dell'esecuzione
- Due o più thread possono comunicare semplicemente condividendo memoria con altri thread, ovvero “vedendo” gli stessi oggetti
  - basta che tutti possiedano un riferimento comune
  - ciò che un thread scrive, può essere letto dagli altri

# Thread e Gestione della Memoria: Stack Autonomi / Heap Condiviso



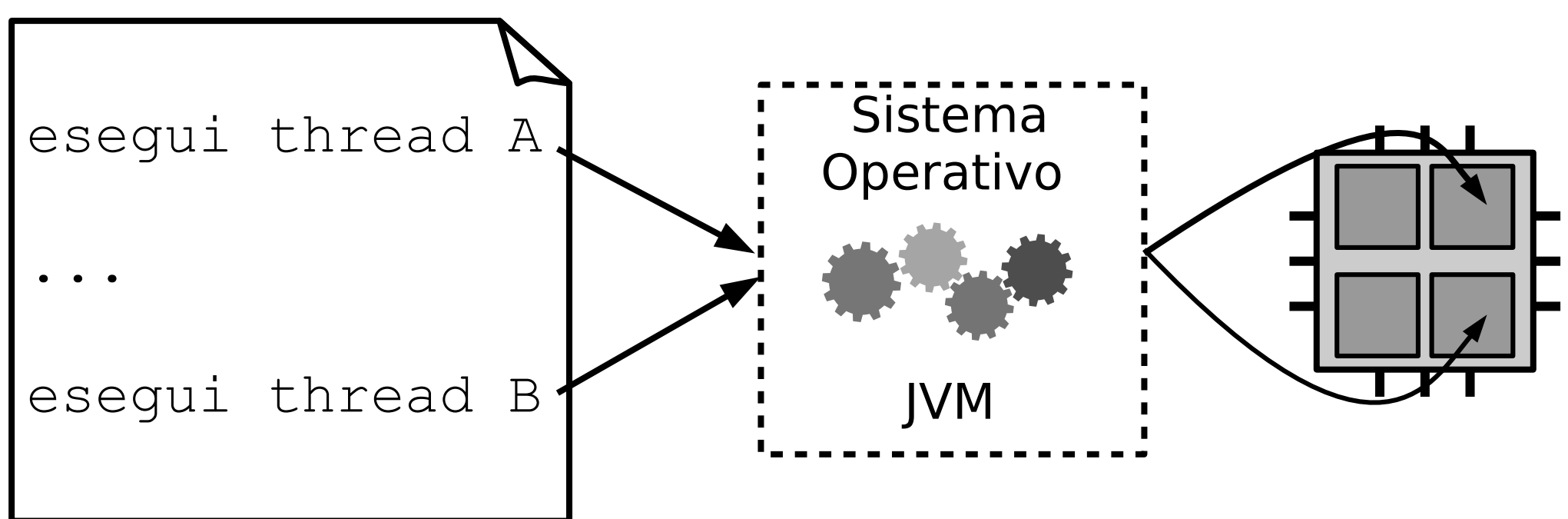


# Piattaforma Java: Supporto alla Programmazione Multi-Thread

- I thread in java sono stati supportati sin dalle primissime versioni della piattaforma in maniera nativa
- Meglio chiarire da subito che *solo* la JVM può offrire il servizio di creare nuovi flussi di esecuzione
- Dobbiamo limitarci a richiederlo passando per oggetti istanza di una classe appositamente pensata allo scopo  
**java.lang.Thread**
  - Oggetti istanza di questa classe NON sono flussi di esecuzione, ma il nostro mezzo di comunicazione con la JVM
- Di nuovo: *solo* la JVM può creare nuovi flussi di esecuzione
- Tale classe permette (tra l'altro) di:
  - creare nuovi flussi di esecuzione (metodo **start()**)
  - specificare il codice da eseguire (interface **Runnable**)
  - aspettare che un flussi termini (metodo **join()**)

# Esecuzione Concorrente

- Un thread viene eseguito dalla JVM che a sua volta fa perno sui servizi offerti dal sistema operativo sottostante
  - ✓ Il programmatore non ha *nessun* controllo sulla scelta operata dalla JVM e dal S.O. sottostante
  - ✓ Non è possibile prevedere quale core verrà usato!



# Creazione di Thread in Java (1)

- Per creare un nuovo thread in Java è possibile istanziare la classe `java.lang.Thread`
- Mediante un costruttore che riceve un parametro di tipo `java.lang.Runnable` per specificare il codice da far eseguire al nuovo thread
- Interfaccia a singolo metodo: `public void run();`
- Bisogna sovrascrivere il suo metodo `run()`

```
public class PrintA implements Runnable {  
    @Override  
    public void run() {  
        for (int j=0; j<100; j++)  
            System.out.println("Sono A");  
    }  
}
```

# Creazione di Thread in Java (2)

- Creare un oggetto istanza della classe `java.lang.Thread` non significa aver creato un nuovo flusso di esecuzione dotato di *Stack*
- Significa aver creato un oggetto che *successivamente* permette di richiedere alla JVM la creazione di un flusso di esecuzione
- La richiesta viene inoltrata alla JVM invocando il metodo `start()`
- Il nuovo flusso esegue il metodo `run()` dell'oggetto sottotipo concreto di *Runnable* ricevuto nel costruttore

# Creazione di Thread in Java (3)

- Attenzione!

Chiamare direttamente il metodo `run()` non è la stessa cosa!

- Significa fare eseguire il codice *NON* ad un nuovo flusso di esecuzione ma direttamente a quello già in corso di esecuzione, come per tutte le usuali chiamate di metodo
- ✓ Non si creano affatto nuovi *Stack...*

# Esecuzione Codice Multi-Thread (1)

```
public class MainThread {  
    public static void main(String[] args) {  
        Thread t = new Thread(new PrintA());  
        t.start();  
        for (int j=0; j<100; j++)  
            System.out.println("Sono Main");  
    }  
}
```

- Cosa stampa?
- Chi lo stampa?

# Esecuzione Codice Multi-Thread (2)

Possibile output:

Sono A  
Sono A  
Sono A  
Sono A  
Sono Main  
Sono Main  
Sono A  
Sono A  
Sono Main  
Sono A  
Sono A  
Sono A  
Sono Main  
Sono A  
Sono A

...

# Esecuzione Codice Multi-Thread (3)

- L'esecuzione del precedente programma in realtà coinvolge *due* thread:
  - Appunto il cosiddetto thread *Main* che viene creato direttamente dalla JVM per conto nostro quando chiediamo di eseguire il metodo `main()`
  - uno per l'esecuzione del codice specificato da `PrintA (<<)` e da noi *esplicitamente* creato invocando il metodo `Thread.start()`
- Le stampe dei due thread sono intervallate:
  - il metodo `main()` e il metodo `run()` sono eseguiti concorrentemente da due thread distinti



# Imprevedibilità delle Esecuzioni

- Esecuzioni distinte del precedente programma possono produrre output diversi
  - non è possibile prevedere l'ordine con cui le operazioni di due thread diversi verranno eseguite
  - Alcune decisioni tese a stabilire l'ordine non sono sotto il controllo diretto del programmatore
- Meglio non cadere nella tentazione di fare assunzioni sull'ordine con cui le istruzioni di due thread verranno eseguite
  - ✓ è *sia* concettualmente *sia* praticamente errato...  
(>>PC)

# Terminazione di un Thread

- Un **Thread** termina normalmente quando:
  - Termina l'esecuzione del metodo `run()`
- Un thread termina anche quando il metodo `run()` viene forzatamente interrotto, ovvero:
  - vengono generati errori e/o sollevate eccezioni durante l'esecuzione del metodo `run()`
  - ad es. perché viene “invitato” a terminare (`>> PC`)

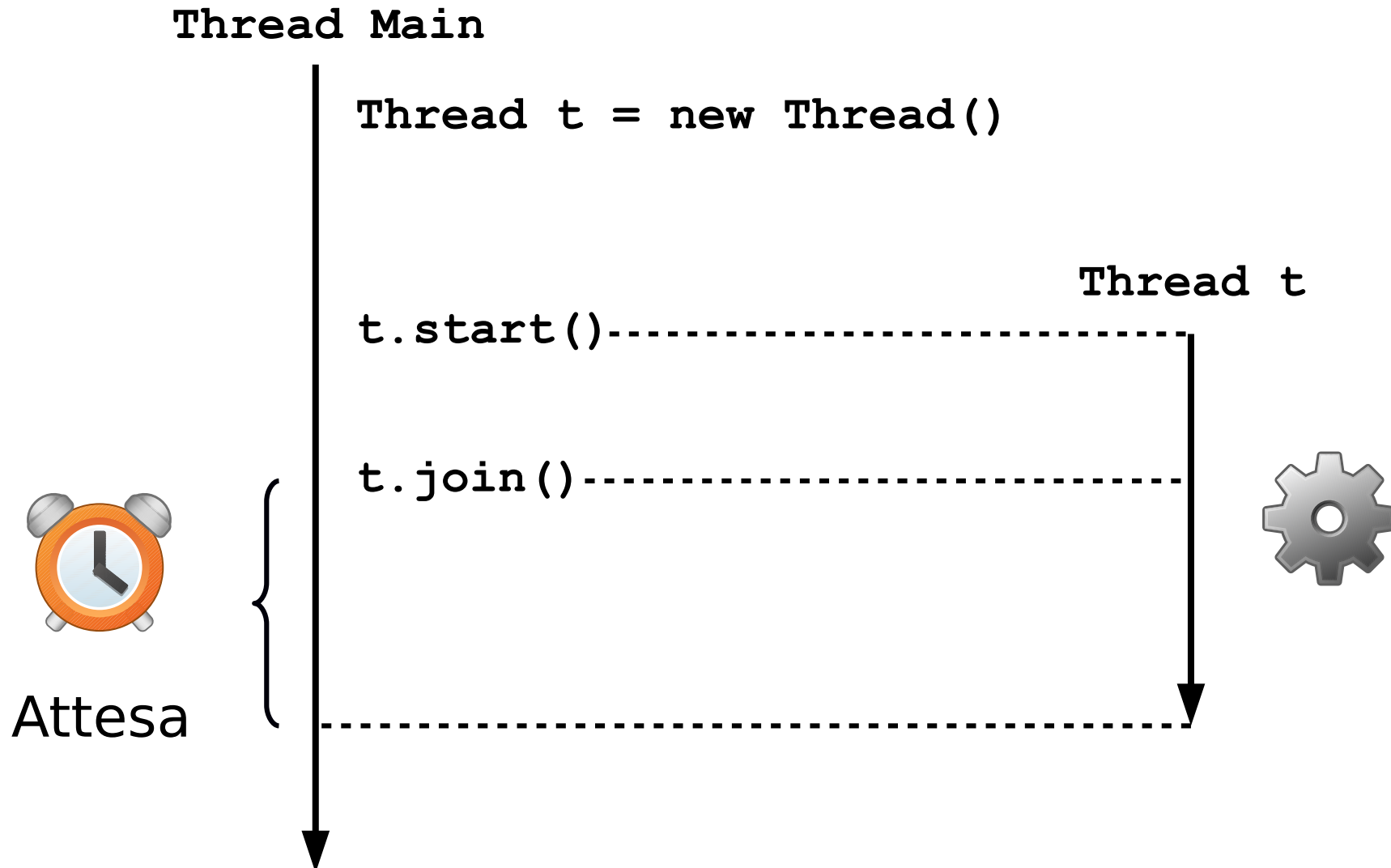
# Decomposizione Parallela

- Un determinato problema può essere diviso in sotto-problemi
  - Ad es.: Calcolo del max assoluto di un array enorme
- La *decomposizione parallela* prevede questi passi:
  - si crei un thread per ogni core disponibile
  - ogni sotto-problema sia assegnato ad un thread distinto che calcoli il “suo” risultato *parziale*
  - i parziali siano collezionati per formare il risultato *globale*
- Molti problemi permettono il calcolo del risultato globale è possibile solo *dopo* che *tutti* i risultati parziali siano stati collezionati
- Per disporre di *tutti* i risultati parziali bisogna aspettare che ciascuno dei thread abbia finito di calcolare il proprio risultato

# Attesa della Terminazione di un Thread: il metodo `java.lang.Thread.join()`

- La classe `java.lang.Thread` offre il metodo `join()`
  - invocando tale metodo su un'istanza di un'oggetto di tipo `Thread`, il thread *corrente* (ovvero quello che sta eseguendo il codice ed invocando il metodo `join()`) viene posto in attesa della terminazione del thread associato all'oggetto su cui è stata chiamata la `join()`
  - Fondamentale per gestire semplici punti di *sincronizzazione* tra thread basati sulla terminazione
  - Unico strumento di sincronizzazione esplicita che vedremo
  - Esistono molte altre tecniche di sincronizzazione
    - Ma ben oltre gli obiettivi formativi di questo corso (vedi corso di PC>>)

# java.lang.Thread.join() : Semantica



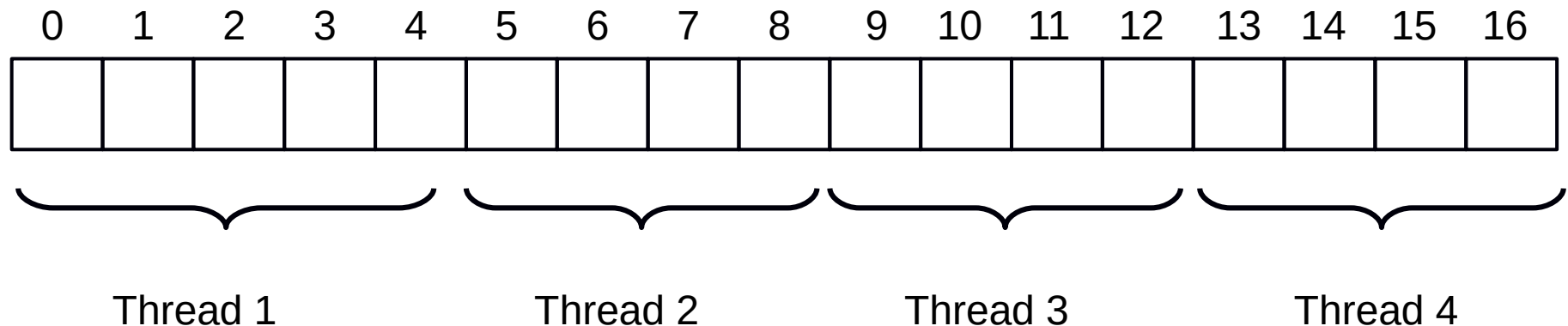
# Esempio: Max di un Array (1)

- L'algoritmo per calcolare il max di un array di interi prevede di scansionare l'intero array con un'unica iterazione completa

```
static interface CalcolatoreMax {  
    public int calcolaIlMaxDi(int[] array) ;  
}  
  
public class CalcolatoreMaxSeriale implements CalcolatoreMax {  
    @Override  
    public int calcolaIlMaxDi(int[] array) {  
        return calcolaIlMaxDi(array, 0, array.length);  
    }  
  
    public int calcolaIlMaxDi(int[] array, int inizio, int fine) {  
        if (fine-inizio==0) throw new NoSuchElementException();  
        int max = array[inizio];  
        for(int i=inizio+1; i<fine; i++) {  
            if (array[i]>max)  
                max = array[i];  
        }  
        return max;  
    }  
}
```

# Esempio: Max di un Array (2)

- Quante parti?
- Per impegnare tutti i core disponibili conviene creare tanti thread quanti sono i core disponibili



- In questo caso, usando 4 thread e un array di 17 elementi, ogni thread elabora 4 elementi; il primo, che gestisce lo sfrido, ne elabora 5
  - Ogni thread calcolerà il max *relativo* alla propria “fetta” di array
  - Il max *assoluto* è il massimo dei max relativi

# Esempio: Max di un Array (3)

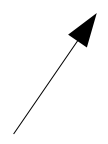
```
public class CalcolatoreMaxParallelo implements CalcolatoreMax {
    final private CalcolatoreMaxSeriale calcolatoreSeriale = new CalcolatoreMaxSeriale();
    @Override
    public int calcolaIlMaxDi(final int[] array) {
        int _N_CORE = Runtime.getRuntime().availableProcessors();
        final int[] maxRelativi = new int[_N_CORE];
        final Thread[] lavoratori = new Thread[_N_CORE];
        int dimensioneFetta = ( array.length ) / _N_CORE;
        int resto = ( array.length ) % _N_CORE;
        for(int i=0; i<_N_CORE; i++) {
            /* fai calcolare il massimo relativo per
               questa fetta d'array ad un thread dedicato */
            lavoratori[i] = new Thread(new Runnable() {
                ...// vedi slide che segue
            });
            lavoratori[i].start(); // comincia subito
        }
        for(int i=0; i<_N_CORE; i++) {
            /* aspetta abbiano tutti terminato */
        }
        return calcolatoreSeriale.calcolaIlMaxDi(maxRelativi);
    }
}
```



# Esempio: Max di un Array (4)

```
... ..  
    int inizio_fetta = 0;  
    for(int i=0; i<_N_CORE; i++) {  
        int fine_fetta = inizio_fetta + dimensioneFetta;  
        if (i<resto) fine_fetta++; // sfrido distribuito ai primi thread  
        final int indiceFetta = i;  
        lavoratori[indiceFetta] = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                maxRelativi[indiceFetta] =  
                    calcolatoreSeriale.calcolaIlMaxDi(array, inizio_fetta, fine_fetta);  
            }  
        });  
        lavoratori[i].start();  
        inizio_fetta = fine_fetta;  
    }  
    for(int i=0; i<_N_CORE; i++) {  
        /* aspetta abbiano tutti terminato */  
    }  
    return calcolatoreSeriale.calcolaIlMaxDi(maxRelativi);  
}
```

?



# Esempio: Max di un Array (5)

... ..

```
/* aspetta abbiano tutti i worker thread abbiano terminato */
for(int i=0; i<_N_CORE; i++) {
    try {
        lavoratori[i].join();
    } catch (InterruptedException e) { // N.B. Checked Exception!
        throw new RuntimeException(e);
    }
}
return calcolatoreSeriale.calcolaIlMaxDi(maxRelativi);
}
```

# Esempio: la Classe `CalcolatoreMaxParallelo`

- Tutti i thread possiedono un riferimento allo stesso array (passato come parametro `final` del metodo `CalcolatoreMaxParallelo.calcolaIlMaxDi()`)
  - Ogni thread però elabora solo gli elementi della propria porzione di array
    - Da: `inizioFetta` (incluso)
    - A: `fineFetta` (escluso)
  - Il risultato viene memorizzato in `maxRelativi`
    - Su questo array si invoca nuovamente `CalcolatoreMaxSeriale.calcolaIlMaxDi()`
    - Tale metodo *deve* essere chiamato solo dopo aver invocato su tutti i thread creati la `join()`
    - Perchè???

# Speed-Up Test (1)

```
@Test
```

```
public void testMaDiQuantiElementiRiescoAcalcolare-  
    IlMaxInOgniMillisecondo_seriale() {  
  
    System.out.println("_____SERIALE_____");  
  
    this.testMaDiQuantiElementiRiescoAcalcolareIlMaxInOgniMs(  
        new CalcolatoreMaxSeriale()  
    );  
}
```

```
@Test
```

```
public void testMaDiQuantiElementiRiescoAcalcolare-  
    IlMaxInOgniMillisecondo_parallelo() {  
  
    System.out.println("_____PARALLELO_____");  
  
    this.testMaDiQuantiElementiRiescoAcalcolareIlMaxInOgniMs(  
        new CalcolatoreMaxParallelo()  
    );  
}
```

# Speed-Up Test (2)

```
private void testMaDiQuantiElementiRiescoAcalcolareIlMaxInOgniMs (
    CalcolatoreMax calcolatore) {
    int volte=31; // 31 ; 14 ... N.B. questi n. dipendono dalla piattaforma...
    for(int v=14; v<volte; v++) {
        int n = (int) Math.pow(2,v); // 2^v
        System.out.println(v+"");
        final int[] array = creaUnArrayDiInteriEriempiloCasualmente(n);
        long prima = System.currentTimeMillis();
        int max = calcolatore.calcolaIlMaxDi(array);
        long dopo = System.currentTimeMillis();
        long ms = dopo - prima ;
        if (ms==0) continue; // troppo veloce, così non riesco a misurare!
        long elXms = n / ( ms ) ;
        System.out.println(
            "Max= "+max+"; Ogni ms ho calcolato il max di " + elXms + " elementi"
        );
    }
}
```

# Speed-Up Test: Risultati (1)

SERIALE

14)

15)

16)

17)

18)

19)

20)

Max= 2147483255; Ogni ms ho calcolato il max di 1048576 elementi

21)

22)

Max= 2147481742; Ogni ms ho calcolato il max di 4194304 elementi

23)

Max= 2147483324; Ogni ms ho calcolato il max di 2796202 elementi

24)

Max= 2147483373; Ogni ms ho calcolato il max di 4194304 elementi

25)

Max= 2147483627; Ogni ms ho calcolato il max di 3728270 elementi

26)

Max= 2147483641; Ogni ms ho calcolato il max di 3195660 elementi

27)

Max= 2147483604; Ogni ms ho calcolato il max di 3355443 elementi

28)

Max= 2147483595; Ogni ms ho calcolato il max di 3677198 elementi

29)

Max= 2147483642; Ogni ms ho calcolato il max di 2618882 elementi

30)

Max= 2147483647; Ogni ms ho calcolato il max di 2810842 elementi

# Speed-Up Test: Risultati (2)

## PARALLELO

14)  
Max= 2147058140; Ogni ms ho calcolato il max di 1365 elementi

15)  
Max= 2147381149; Ogni ms ho calcolato il max di 5461 elementi

16)  
Max= 2147423574; Ogni ms ho calcolato il max di 6553 elementi

17)  
Max= 2147479308; Ogni ms ho calcolato il max di 18724 elementi

18)  
Max= 2147478291; Ogni ms ho calcolato il max di 262144 elementi

19)  
Max= 2147482968; Ogni ms ho calcolato il max di 174762 elementi

20)  
Max= 2147478347; Ogni ms ho calcolato il max di 524288 elementi

21)  
Max= 2147477630; Ogni ms ho calcolato il max di 2097152 elementi

22)  
Max= 2147483116; Ogni ms ho calcolato il max di 2097152 elementi

23)  
Max= 2147481732; Ogni ms ho calcolato il max di 4194304 elementi

24)  
Max= 2147482590; Ogni ms ho calcolato il max di 2396745 elementi

25)  
Max= 2147483332; Ogni ms ho calcolato il max di 5592405 elementi

26)  
Max= 2147483589; Ogni ms ho calcolato il max di 6100805 elementi

27)  
Max= 2147483629; Ogni ms ho calcolato il max di 6391320 elementi

28)  
Max= 2147483646; Ogni ms ho calcolato il max di 3195660 elementi

29)  
Max= 2147483620; Ogni ms ho calcolato il max di 7158278 elementi

30)  
Max= 2147483640; Ogni ms ho calcolato il max di 7017920 elementi

# Speed-Up Test: Commenti

- Confrontiamo uno degli ultimi risultati (tra i migliori):
  - Seriale
    - 29) Max= 2147483642; Ogni ms ... max di 2618882 elementi
  - Parallelo
    - 29) Max= 2147483620; Ogni ms ... max di 7158278 elementi
- Versione parallela circa 2.73 volte più veloce...
  - non male...
  - Ma deludente se consideriamo che l'esecuzione è stata fatta su un macchina con ben 8 core
- Due principali motivi:
  - La parallelizzazione ha introdotto un sovracosto per la decomposizione
  - Una parte del lavoro è rimasta completamente seriale. Quale?



# Esempio: Max Quinte Potenze di un Array

- Proviamo a cambiare leggermente il *problema* affrontato per renderlo ancora più pesante dal punto di vista computazionale
- Calcoliamo il max delle quinte potenze degli elementi dell'array
- Cambiamo l'implementazione di **CalcolatoreMaxSeriale**

```
public int calcolaIlMaxDi(int[] array, int inizio, int fine) {  
    if (fine-inizio==0)  
        throw new NoSuchElementException();  
    int max = array[inizio];  
    for(int i=inizio+1; i<fine; i++) {  
        if (Math.pow(array[i],5) > max)  
            max = array[i];  
    }  
    return max;  
}
```

# Speed-Up Test

## PARALLELO

...

26)

Max= 939950888; Ogni ms ho calcolato il max di 95460 elementi

27)

Max= 187273864; Ogni ms ho calcolato il max di 93336 elementi

## SERIALE

...

26)

Max= 1217087784; Ogni ms ho calcolato il max di 14397 elementi

27)

Max= 1724773000; Ogni ms ho calcolato il max di 14281 elementi

- Versione parallela circa 6.63 volte più veloce...
- Perché ora lo *speed-up* è più elevato?
- ✓ Perché il problema è diventato così pesante che i sovracosti della decomposizione parallela sono divenuti trascurabili rispetto al costo complessivo
- Raggiungeremo mai 8?  
Purtroppo NO! (PC>>)

# Creazione di un Thread: Estendere `java.lang.Thread`

- Per evitare di definire una nuova classe **Runnable** è possibile anche estendere direttamente **Thread** che in effetti è già un suo sottotipo
- A quel punto invocando **Thread.start()** si chiede la creazione di un thread che esegue il metodo **Thread.run()** sovrascritto nella sottoclasse
- Due principali problemi di questo approccio:
  - Non si è più liberi di estendere un'altra classe
  - *Si confondono due concetti intimamente correlati e tuttavia ben distinti:*
    - La creazione di un thread
    - La specifica di ciò che deve fare il thread appena creato

# Esempio: Estensione della Classe Thread

```
public class GreeterA extends Thread implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("Hello! I'm A");  
    }  
}
```

```
public class GreeterB extends Thread implements Runnable {  
  
    public void run() {  
        System.out.println("Hello! I'm B");  
    }  
}
```

```
public class MultiGreeter {  
    public static void main (String[] args) {  
        Thread greeterA = new GreeterA();  
        Thread greeterB = new GreeterB();  
  
        Thread threadA = new Thread(greeterA);  
        Thread threadB = new Thread(greeterB);  
  
        threadA.start();  
        threadB.start();  
    }  
}
```

# Conclusioni

- Si è vista una breve introduzione alla *Programmazione Concorrente*
- Con la diffusione di massa delle architetture multi-core e l'adozione di massa di soluzioni distribuite risulta un argomento ancora più ineludibile
- Vedremo che risulta argomento necessario anche per la comprensione del modello di computazione sottostante le applicazioni dotate di GUI
  - Ad es. praticamente *tutte quelle mobile*