UNIVERSITÀ DI PISA

Programmazione di Reti
Corso B

20 Settembre 2016

Lezione 1

Contatti

- Alina Sîrbu
  - alina.sirbu@unipi.it
  - Stanza 331 DO – su appuntamento
- Andrea De Salve
  - desalve@di.unipi.it
- Lezioni - Modulo Laboratorio
  - 11:15 - 13:00 – teoria – si spiegano i concetti
  - 14:00 - 15:45 – pratica – in laboratorio – si lavora individualmente ai compiti

Corso

- elearning.di.unipi.it - corso “Laboratorio di reti B - AA 2016-2017”
- iscrizione obbligatoria - chiave “LabRetiB1617”
- annunci
- slide
- compiti
  - 1 per settimana – sottomissione entro 2 settimane (Lunedì sera)
  - valutati – unico voto alla fine
  - contano come bonus al voto finale (0-3 punti di bonus)

Contenuti

- 2 settimane – Programmazione multithreaded
- 1 settimana - Java IO
- 5 settimane – Programmazione di rete a basso livello, HTTP e servizi web
- 2 settimane – Programmazione di rete ad alto livello (RMI)
- 2 settimane - NIO

Bibliografia

- Documentazione Java 8
- *Multithreading*:
  - *Java concurrency tutorial*: https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
  - Oaks & Wong, “Java Threads”, O’Reilly
  - Lewis & Berg, “Multithreaded programming with java technology”, Sun Microsystems
- *Socket*:
  - *Java networking tutorial*: https://docs.oracle.com/javase/tutorial/networking/index.html
  - Harold, “Java Network Programming”, O’Reilly
- RMI:
  - *Java RMI tutorial*: https://docs.oracle.com/javase/tutorial/rmi/
  - Architettura RMI : https://www.cis.upenn.edu/~bcpierce/courses/629/jdkdocs/guide/rmi/spec/rmi-arch.doc.html

Valutazione

- Un voto unico annuale per il corso “Reti di calcolatori e laboratorio”:
  - **progetto** - Alina Sîrbu
    - sottomissione 7 giorni prima della prova scritta
    - discussione progetto (se sottomissione sufficiente)
    - bonus per i compiti
  - scritto - Prof Bonuccelli (solo se superata la discussione)
  - orale - Prof Bonuccelli (solo se superato lo scritto)

Esempi ed esercizi

- Eclipse IDE (ambiente di sviluppo) o editore di testo + *command line*

- Obbligatorio fare delle domande

Perché

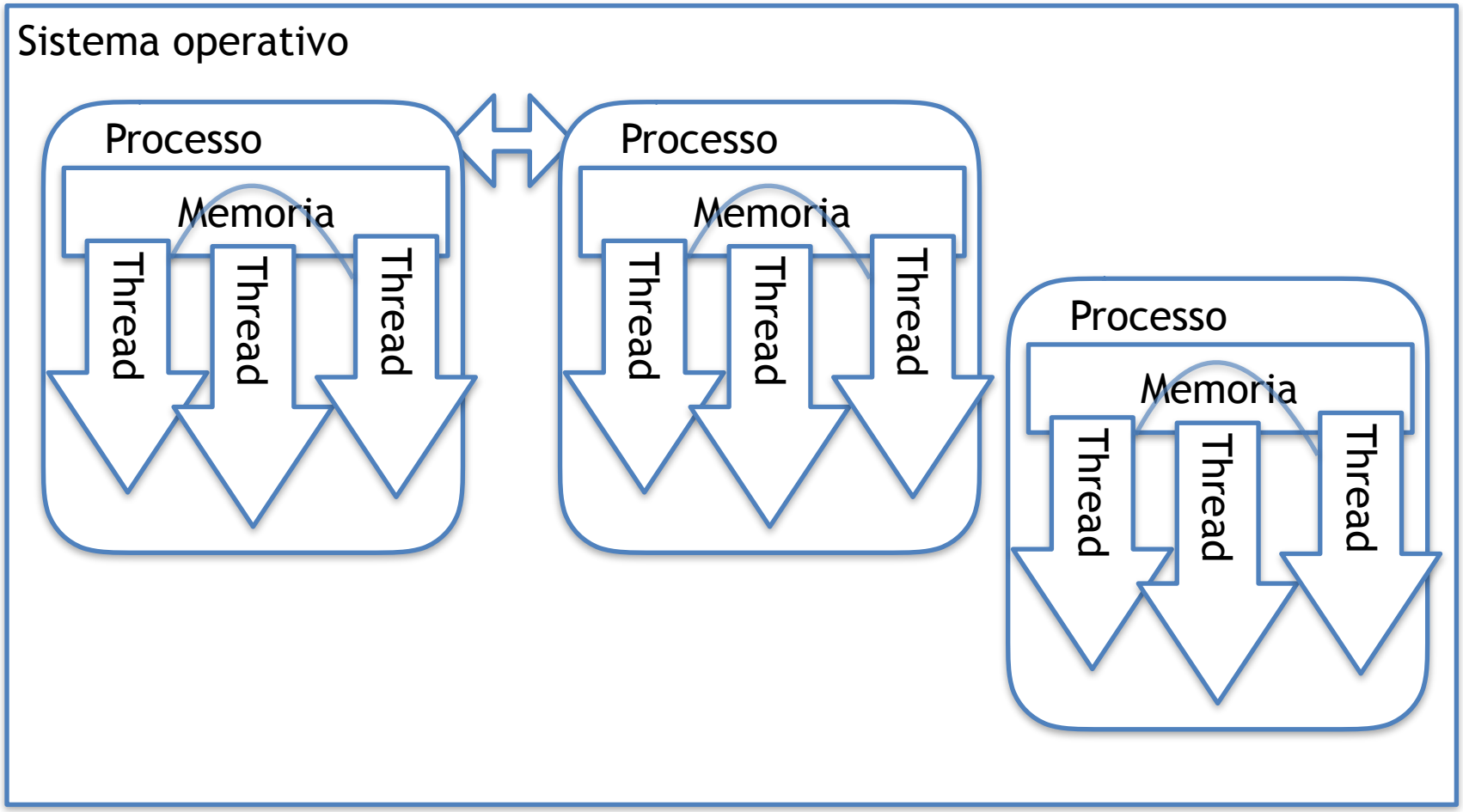
- **Tutte** le applicazioni moderne usano multithreading
  - Basi di dati
  - *Software design*
  - *Interface design*

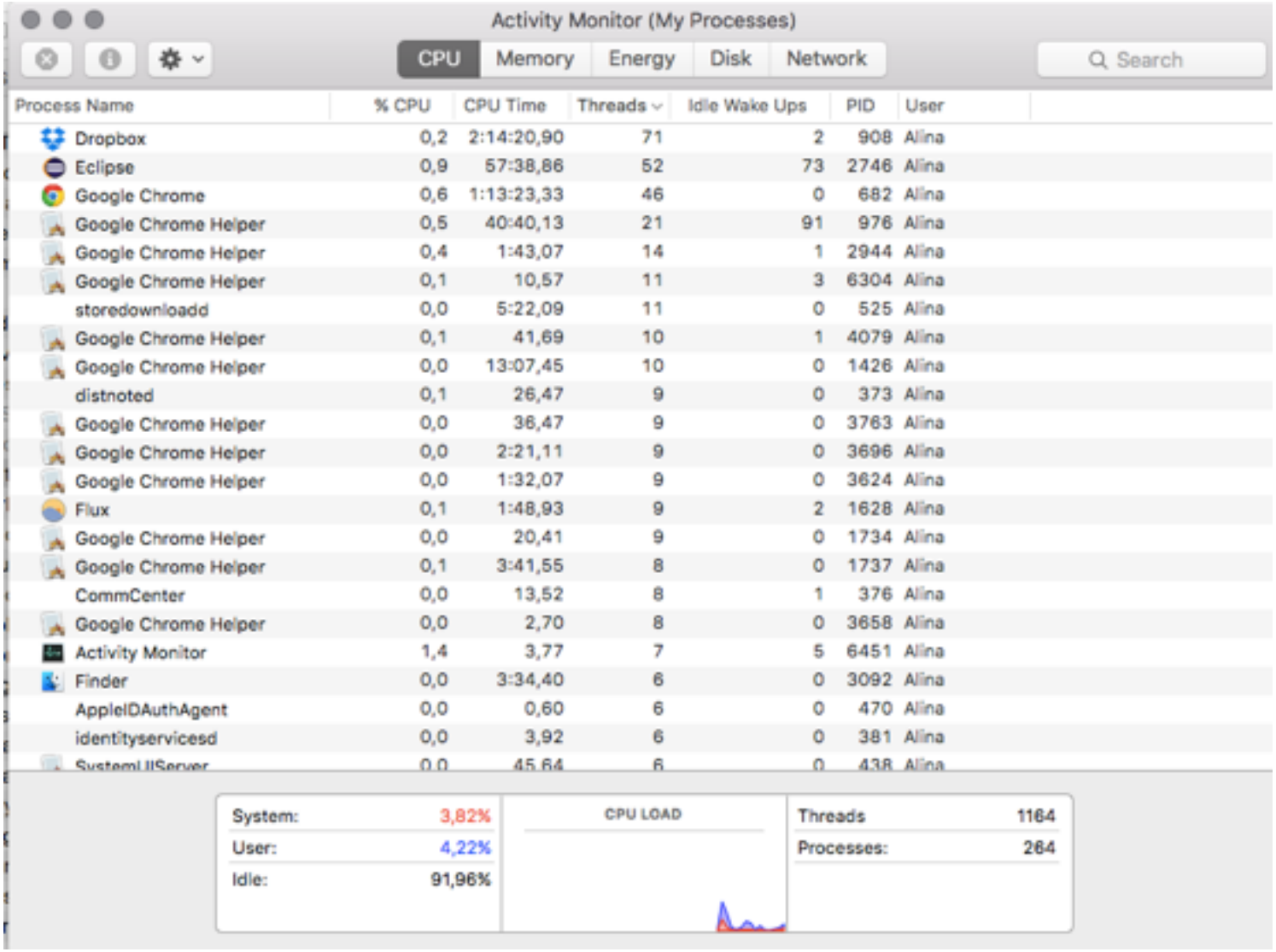
- **La maggiorità** delle applicazioni moderne usano programmazione di reti - cliente e server, servizi web
  - *Gaming industry*
  - *Chat* e *social network*
  - *Cloud computing*

Multithreading

- **processo**
  - programma (insieme di istruzioni) eseguito indipendentemente nel sistema operativa, con un spazio di memoria privata, file handler privati, e impostazioni di sicurezza
  - comunicano tramite *socket*, *pipe* o *file* - anche a distanza
- **thread**
  - insieme di istruzioni consecutive eseguite dentro un processo - filo di esecuzione, processo *lightweight*
  - condivide lo spazio di memoria e *file handler* con gli altri *thread* dello stesso processo
  - comunicano facilmente usando *shared memory*
- nel sistema operativo: ogni applicazione ha almeno un processo
- ogni processo ha almeno un *thread* - inizia e finisce con la funzione `main()`

Processi e *thread*

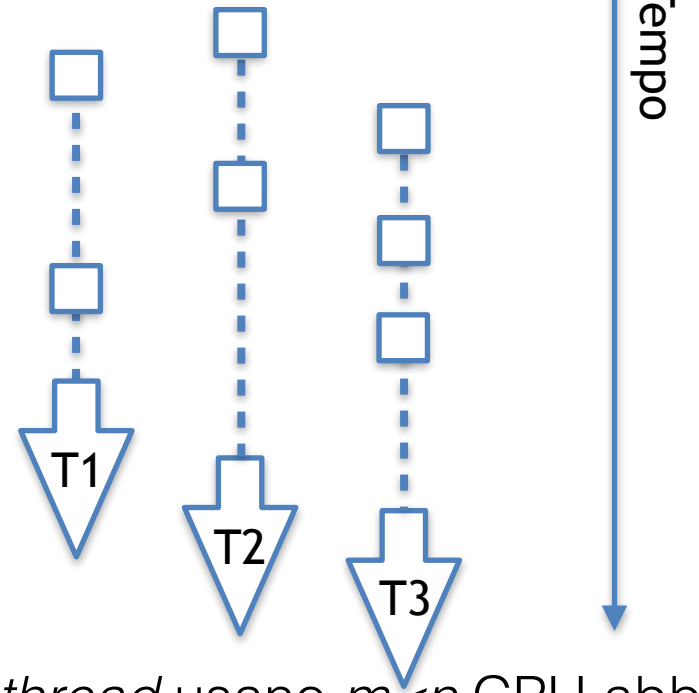




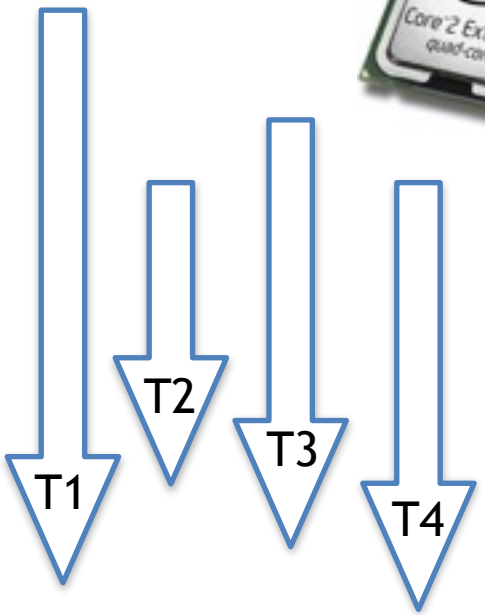
Quasi tutti i processi usano più di un *thread* (un'eccezione e la *command line - bash)*

Motivazione

- **Concorrenza**
  - n *thread* condividono lo stesso CPU



- **Parallelismo**
  - *n thread* usano *n* CPU (core)



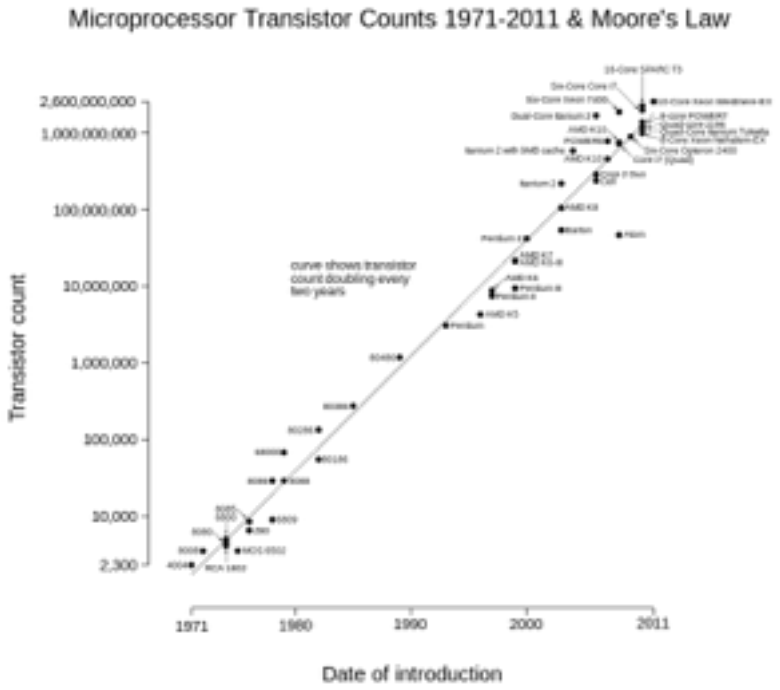
- Se *n thread* usano *m*<*n* CPU abbiamo concorrenza **e** parallelismo allo stesso tempo
- per il programmatore il numero di CPU disponibili non fa differenza

Motivazione

- **Concorrenza**
  - **Reattività** migliore - GUI
  - Uso migliore delle risorse - quando un *thread* si blocca gli altri continuano
  - *Fairness* - tutti gli utenti/ *thread* hanno accesso alle risorse senza aspettare (concorrentemente)
  - Semplifica programmi complessi - ogni *thread* responsabile di un *task* semplice
- **Parallelismo**
  - Uso migliore delle risorse - *core*
  - Computazioni più veloci
  - Possibile solo per piattaforme *multiprocessor* (virtualmente ogni processore moderno)

La legge di Moore

- La velocità dei CPU si raddoppia ogni due anni
  - Non più valida - i CPU hanno raggiunto i loro limiti fisici
  - Adesso la potenza dei computer cresce istallando più CPU (core) in una sola machina
  - Parallelizzazione diventa onnipresente



La legge di Amdahl

- Un programma ha un tempo di esecuzione su 1 CPU uguale a T
- Quale sarà il tempo di esecuzione su N CPU
- F - frazione del programma che deve essere eseguito serialmente
- T'>Ts+Tp/N =TxF+ Tx(1-F)/N=

= Tx(F+(1-F)/N)

- Il limite sta nel nostro programma

Context switching

- Il scheduler (sistema operativo) decide quale *thread* mettere in esecuzione
- Strategia: *time slicing* - ogni *thread* ha una “fetta” di tempo a disposizione, poi viene sospeso
- *Context switching*: la sospensione di un thread e la ripresa di un altro *thread*
  - Un’operazione relativamente costosa

Synchronous vs asynchronous

- 2 modi di effettuare un'operazione
- **synchronous**
  - avvio e eseguo l'operazione fino ad arrivare al risultato
  - applicazioni con un solo thread
- **asynchronous**
  - avvio l'operazione da eseguire da un altro, e comincio a fare un'altra cosa
  - devo controllare più tardi il risultato
  - applicazioni con thread multipli

Blocking vs non-blocking

- Qualche operazioni possono aver bisogno di tempo per concludere
  - Leggere un file già aperto da un'altra applicazione
  - Leggere dati dalla rete
  - etc.
- Due modi di eseguire questi operazioni
  - **blocking**: avvio, mi fermo e non faccio niente fino che l'operazione è avvenuta con successo.
  - **non-blocking:** provo di avviare, se funziona eseguo l'operazione, altrimenti faccio altro e ritorno più tardi

Interfaccia Java

- Classe: definizione di un tipo di oggetti: attributi (proprietà) e metodi (azioni disponibili)
- Oggetti comunicano tra di loro tramite i metodi - interfacciano
- Interfaccia*:* insieme di metodi che un oggetto deve avere per poter essere usato in un modo predefinito
- Interfaccia in Java - definizione dei metodi, senza l'implementazione
- Una Classe può implementare l'interfaccia per definire un tipo di oggetto

```
public interface GeometricShape {
    public double getSurface();
    public double getXCoordinate();
    public double getYCoordinate();

}

public class ShapeHandler {

    public void handle(GeometricShape s){
        System.out.format("I am working with "
                + "a shape located at (%f,%f), "
                + "with a surface of %f",
                s.getXCoordinate(),
                s.getYCoordinate(),
                s.getSurface());
    }
}
```

```
public class Circle implements GeometricShape{
    private double radius;
    private double x,y;
    public Circle(double radius, double x,double y){
        this.radius=radius;
        this.x=x;
        this.y=y;
    }
    @Override
    public double getSurface() {
        return this.radius*this.radius*Math.PI;
    }
    @Override
    public double getXCoordinate() {
        return this.x;
    }
    @Override
    public double getYCoordinate() {
        return this.y;
    }
}
```

```
public static void main(String[] args){
        GeometricShape shape= new Circle(13,0,0);
        ShapeHandler h = new ShapeHandler();
        h.handle(shape);
}
```

Creazione *thread* in Java

- Definizione del *task* in due modi:
  - Estendere la classe j`ava.lang.Thread`
  - Implementare l'interfaccia `java.lang.Runnable`
- Creazione oggetto `Thread`
- Richiamare metodo `start`()

Classe `java.lang.Thread`

```
public class MyFirstThread extends Thread {

    public void threadPrint(String message){
        System.out.println("Thread “
          +Thread.currentThread().getId()+": “+message);
  }

    public void printInfo(){
        this.threadPrint("I am a new thread.");
        this.threadPrint(this.toString());
        this.threadPrint("My name is "+ this.getName());
        this.threadPrint("My priority is "+ this.getPriority());
    }

    public void run(){
        this.printInfo();
    }
}
```

```
public static void main(String[] args) {

    for (int i=0;i<10;i++){
        MyFirstThread thread=new MyFirstThread();
        thread.start(); //not thread.run()
    }
}
```

```
Thread 9: I am a new thread.
Thread 12: I am a new thread.
Thread 11: I am a new thread.
Thread 11: Thread[Thread-2,5,main]
Thread 10: I am a new thread.
Thread 13: I am a new thread.
Thread 11: My name is Thread-2
Thread 12: Thread[Thread-3,5,main]
Thread 12: My name is Thread-3
Thread 12: My priority is 5
Thread 9: Thread[Thread-0,5,main]
Thread 9: My name is Thread-0
Thread 9: My priority is 5
Thread 15: I am a new thread.
Thread 15: Thread[Thread-6,5,main]
Thread 15: My name is Thread-6
Thread 15: My priority is 5
Thread 14: I am a new thread.
Thread 17: I am a new thread.
Thread 17: Thread[Thread-8,5,main]
Thread 17: My name is Thread-8
Thread 11: My priority is 5
Thread 13: Thread[Thread-4,5,main]
Thread 13: My name is Thread-4
Thread 10: Thread[Thread-1,5,main]
Thread 13: My priority is 5
Thread 18: I am a new thread.
Thread 17: My priority is 5
Thread 14: Thread[Thread-5,5,main]
Thread 16: I am a new thread.
Thread 14: My name is Thread-5
Thread 14: My priority is 5
Thread 18: Thread[Thread-9,5,main]
Thread 18: My name is Thread-9
Thread 10: My name is Thread-1
Thread 18: My priority is 5
Thread 16: Thread[Thread-7,5,main]
Thread 10: My priority is 5
Thread 16: My name is Thread-7
Thread 16: My priority is 5
```

`start()` e `run()`

- `start()` crea un nuovo *thread (processo lightweight)* che esegue le istruzioni del metodo `run()`
- `run()` esegue le istruzioni del metodo `run()` nel thread già esistente - **NO MULTITHREADING**
- **non usare mai** `run()` **direttamente!!!!**

```
Thread 1: I am a new thread.
Thread 1: Thread[Thread-0,5,main]
Thread 1: My name is Thread-0
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-1,5,main]
Thread 1: My name is Thread-1
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-2,5,main]
Thread 1: My name is Thread-2
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-3,5,main]
Thread 1: My name is Thread-3
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-4,5,main]
Thread 1: My name is Thread-4
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-5,5,main]
Thread 1: My name is Thread-5
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-6,5,main]
Thread 1: My name is Thread-6
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-7,5,main]
Thread 1: My name is Thread-7
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-8,5,main]
Thread 1: My name is Thread-8
Thread 1: My priority is 5
Thread 1: I am a new thread.
Thread 1: Thread[Thread-9,5,main]
Thread 1: My name is Thread-9
Thread 1: My priority is 5
```

Interfaccia `java.lang.Runnable`

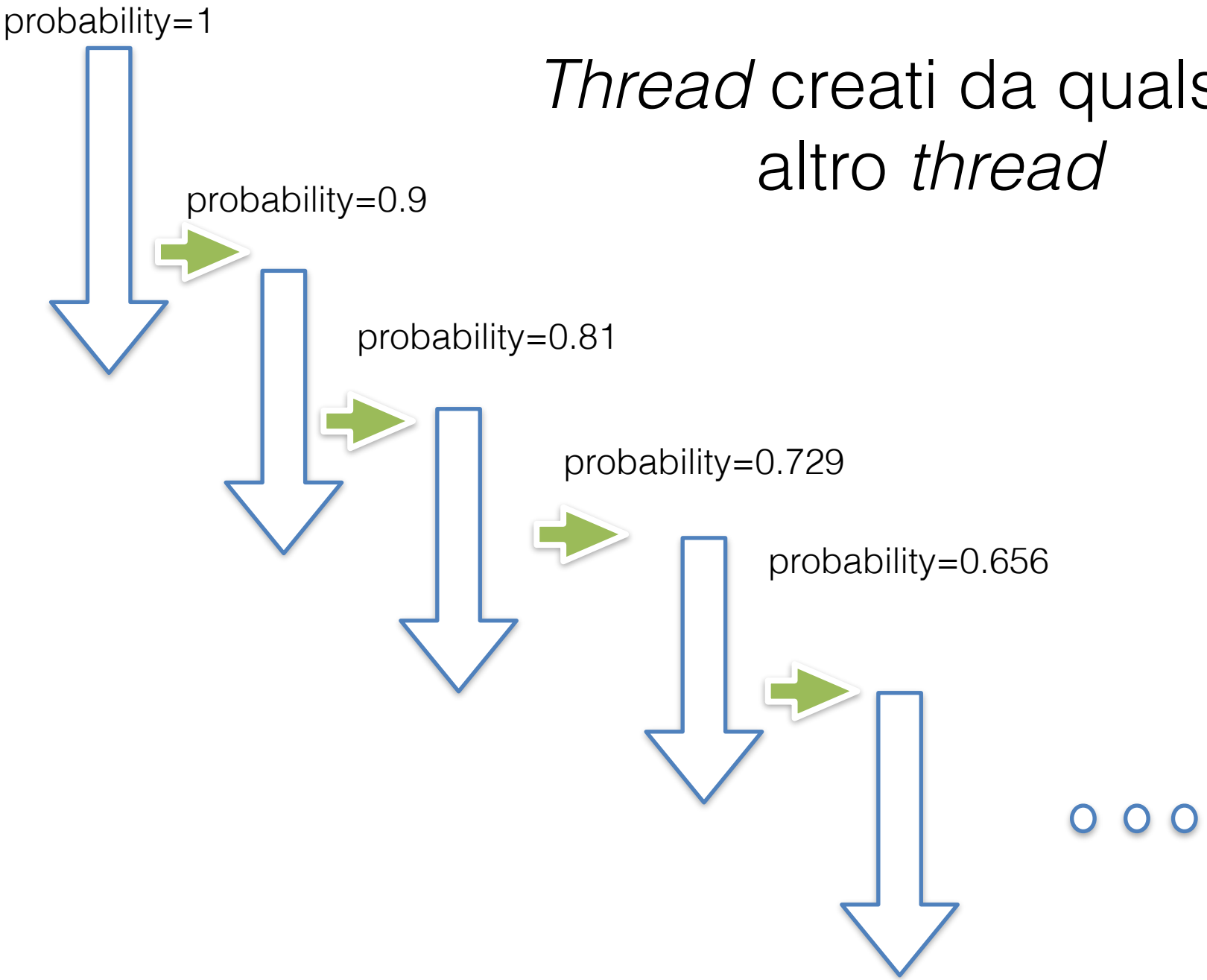
```
public class MyFirstRunnable implements Runnable{

    public void threadPrint(String message){
        System.out.println("Thread “+
            Thread.currentThread().getId()+": "+message);
    }

    public void printInfo(){
        this.threadPrint("I am a new thread.");
        this.threadPrint(Thread.currentThread().toString());
        this.threadPrint("My name is "+
            Thread.currentThread().getName());
        this.threadPrint("My priority is "+
            Thread.currentThread().getPriority());

    }
    public void run(){
        this.printInfo();
    }
}
```

```
public static void main(String[] args) {
    for (int i=0;i<10;i++){
        MyFirstRunnable runnable= new MyFirstRunnable();
        Thread thread=new Thread(runnable);
        thread.start();
    }
}
```

`Thread` e `Runnable`

- `Thread` più semplice per applicazioni piccole
- `Runnable` più flessibile ed elegante
  - si può estendere un'atra classe di base
  - la logica del *thread* (il *task*) è separata dall'oggetto `Thread`
  - usabile con meccanismi nuovi di multithreading (Java > 5)

Thread creati da qualsiasi altro *thread*



Thread creati da qualsiasi altro *thread*

```
import java.util.Random;
//creating a chain of threads
public class ThreadInRunnable implements Runnable {

    private double probability;

    public ThreadInRunnable(double probability){
        this.probability=probability;
    }
    public static void main(String[] args) {
        //start the chain with probability 1
        ThreadInRunnable runnable= new ThreadInRunnable(1.0);
        Thread t = new Thread(runnable);
        t.start();
    }
    //method run goes here
}
```

```
public void run(){
    String me = Thread.currentThread().getId();
    System.out.println(me+": Created");

    //with probability probability create a new thread

    Random r = new Random(System.currentTimeMillis()*2000);
    double rn=r.nextDouble();

    if (rn<this.probability){

        System.out.println(me + ": Creating a new thread");
        ThreadInRunnable runnable=
                new ThreadInRunnable(this.probability*0.9);
        Thread t = new Thread(runnable);
        t.start();
        System.out.println(me + ": created thread “+t.getId());

    }
    else{
        System.out.println(me + ": Stoping the chain");
    }
}
```

Fare una pausa

- `Thread.sleep(2000)`
- Causa il *thread* attivo di fermarsi per 2 secondi
- Il tempo in cui il *thread* ridiventa attivo non e fisso a 2 secondi - dipende dal sistema operativo
- Usare `sleep` per dare spazio ad altri *thread* quando necessario

Interruzione thread

- Richiamare `t.interrupt()` da un altro thread
- Alcuni metodi si fermano e lanciano una `InterruptedException`: `sleep()`, `join()`, `wait()`
- Se non usiamo questi metodi: dobbiamo controllare periodicamente se `Thread.interrupted()` e `true` nel metodo `run()`
  - Meglio trattare l'eccezione una volta sola (quindi `throw new InterruptedException()`)
- Quando il *thread* e interrotto, per fermarlo basta fare `return` nel metodo `run()`

```
public class InterruptedThread implements Runnable{
    public void run(){
        public void run(){
        try {
            //generate numbers & write to screen until interrupted
            Random r = new Random(System.currentTimeMillis()*1000);
            while(true){
                if (Thread.interrupted()){
                    throw new InterruptedException("Interrupted
                                                    outside sleep");
                }
                System.out.print(r.nextDouble()+", ");
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e) {
            System.out.println("\nI was interrupted, I am stopping
                                (“+e.getMessage()+")");
        }
    }//main goes here
}
```

```
public static void main(String args[]){
    InterruptedThread runnable= new InterruptedThread();
    Thread t = new Thread(runnable);
    t.start();

    boolean interrupted=false;
    Random r = new Random(System.currentTimeMillis()*1000);
    while (!interrupted){
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
        if (r.nextDouble()<0.2){
            t.interrupt();
            interrupted=true;
        }
    }
}
```

Meccanismo interruzioni

- La classe `Thread` contiene `true/false` flag: "*interrupt status*"
- `threadObject.interrupt()` mette il flag True per il thread `threadObject`
- Thread.interrupted() se `true`, mette il flag `false` per thread attuale
- `threadObject.isInterrupted()` non cambia il flag

Join

- Quando un *thread* deve aspettare che un'altro finisca, usare `t.join();`
- Versione con timeout `t.join(1000);`
- Il *thread* attuale si blocca fino alla fine del *thread* `t`
- Lancia `InterruptedException`
- Si usa quando un thread deve aspettare i risultati di un altro thread per continuare

```
public class JoinThread implements Runnable{

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread()
                                    +": Just started”);

            Thread.sleep((System.currentTimeMillis()%10)*1000);

            System.out.println(Thread.currentThread()+": Done”);

        } catch (InterruptedException e) {}
    }

}
```

```
public static void main(String[] str) throws InterruptedException{
        System.out.println(Thread.currentThread()
                                                +": Main thread started");

        ArrayList<Thread> threads= new ArrayList<Thread>();
        for (int i=0;i<10;i++){
            Thread t= new Thread(new JoinThread());
            threads.add(t);
            t.start();
        }

        System.out.println(Thread.currentThread()
                                            +": Waiting on other threads");
        for (Thread t : threads){
            t.join();
        }

        System.out.println(Thread.currentThread()
                                            +": All threads finished");
    }
```

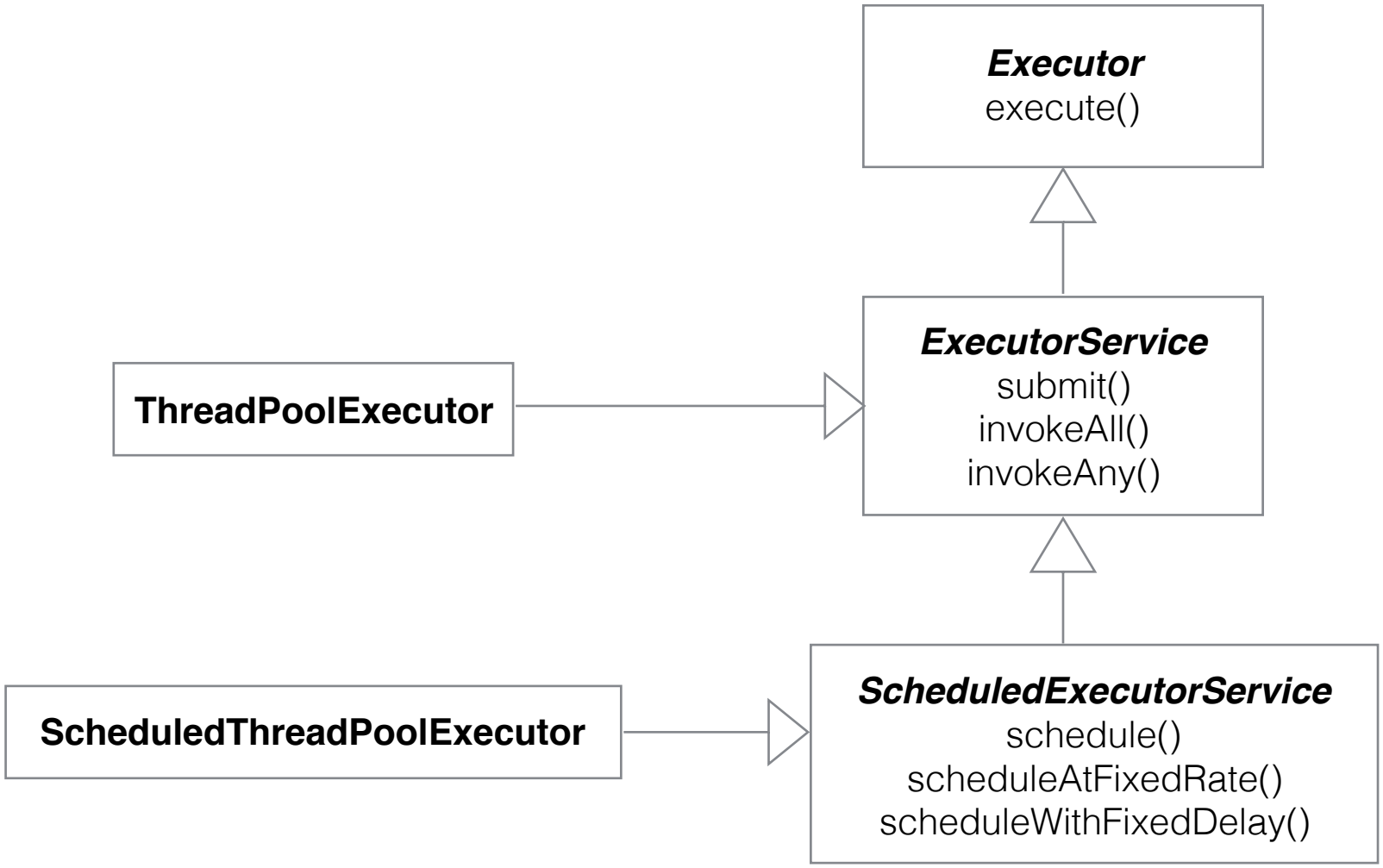
Output:

```
Thread[main,5,main]: Main thread started
Thread[Thread-0,5,main]: Just started
Thread[Thread-1,5,main]: Just started
Thread[Thread-2,5,main]: Just started
Thread[Thread-3,5,main]: Just started
Thread[Thread-4,5,main]: Just started
Thread[Thread-5,5,main]: Just started
Thread[Thread-6,5,main]: Just started
Thread[Thread-7,5,main]: Just started
Thread[main,5,main]: Waiting on other threads
Thread[Thread-9,5,main]: Just started
Thread[Thread-8,5,main]: Just started
Thread[Thread-0,5,main]: Done
Thread[Thread-2,5,main]: Done
Thread[Thread-7,5,main]: Done
Thread[Thread-8,5,main]: Done
Thread[Thread-9,5,main]: Done
Thread[Thread-6,5,main]: Done
Thread[Thread-5,5,main]: Done
Thread[Thread-4,5,main]: Done
Thread[Thread-3,5,main]: Done
Thread[Thread-1,5,main]: Done
Thread[main,5,main]: All threads finished
```

I *thread pool*

- Creare un *thread* nuovo introduce un overhead
- Avere un numero di *thread* più grande di una soglia può danneggiare la *performance*
- Può essere utile limitare il numero massimo dei *thread* del mio programma
- In caso i *task* sono molti e corti, può essere utile riusare dei *thread* già creati
- La scelta dipende dal sistema usato (quanti CPU, memoria, etc) e dalla applicazione (e.g. quanti utenti sono previsti)
- In **Java 5** sono introdotti gli *executor*, che ci aiutano a creare dei *thread pool*

Gerarchia di *executor*



Interfaccia `Executor`

- Ambiente controllato di esecuzione dei *task*
- Alternativa a creare oggetti **`Thread`** in tutti i programmi già presentati
- Dei *task* vengono inviati agli *executor* che li gestiscono automaticamente - metodo **`void execute(Runnable task)`**
- Per ogni *executor* attivo, ci possono essere *task* sottomessi in attesa (una coda), e *task* in esecuzione

```
Runnable task;
….
(new Thread(task)).start()
```

```
Runnable task;
Executor e;
….
e.execute(task);
```

Interfaccia `ExecutorService`

- Vari altri metodi, oltre `execute()`
- Implementata da `ThreadPoolExecutor`
- Gli *executor* devono essere chiusi all fine, altrimenti il programma non si ferma
- metodo `void shutdown()`
  - *L'executor* non accetta *task* nuovi. I *task* gia sottomessi finiscono (sia quelli in coda che quelli in esecuzione).

Interfaccia `ExecutorService`

- metodo `List<Runnable> shutdownNow()`
  - L'*executor* non accetta *task* nuovi e cancella i *task* in coda. Invia un'interruzione ai *task* in esecuzione (non è detto che questi si spengono, devono essere in grado di gestire l'interruzione).
  - restituisce tutti i `Runnable` che non hanno iniziato la loro esecuzione (quelli in coda, cancellati)
- metodo `boolean awaitTermination(long timeout, TimeUnit unit)`
  - si blocca fin che tutti i *task* finiscono dopo aver richiamato `shutdown` o `shutdownNow`, o fin che il *timeout* passa

Classe `Executors`

- *Factory* per vari tipi di *executor*
- `ExecutorService Executors.newFixedThreadPool(int n)` - crea un *pool* con numero di *thread* fisso
- `ExecutorService Executors.newCachedThreadPool()` - crea un *pool* con numero di *thread* variabile - possibilmente molto grande - riusa i *thread* esistenti quando possibile
- `ExecutorService Executors.newSingleThreadExecutor()` - *executor* con un solo *thread* - può sostituire il semplice `Thread`. Può eseguire più *task*.

Esempio

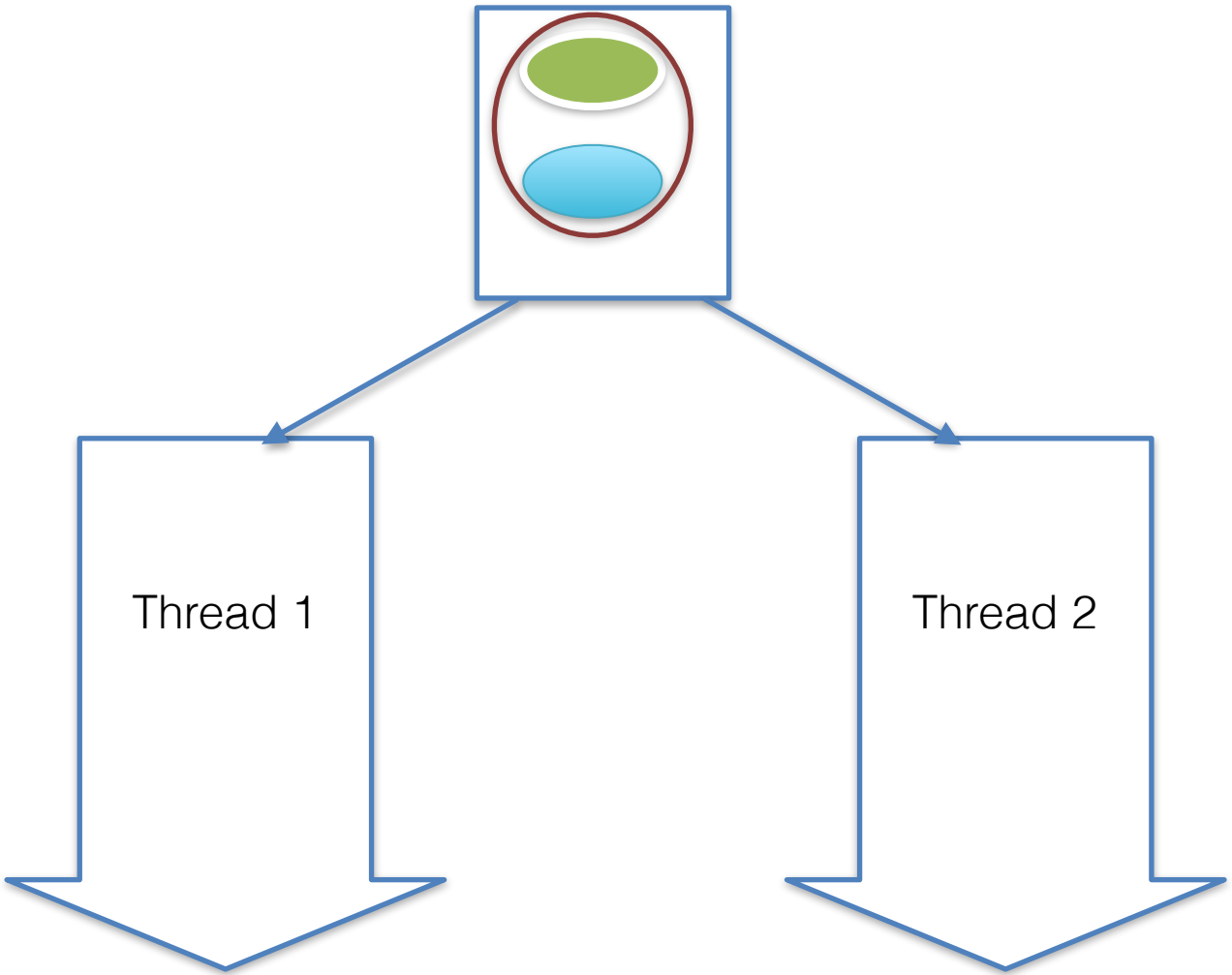
- Implementiamo l'esempio 'join thread' con Executor

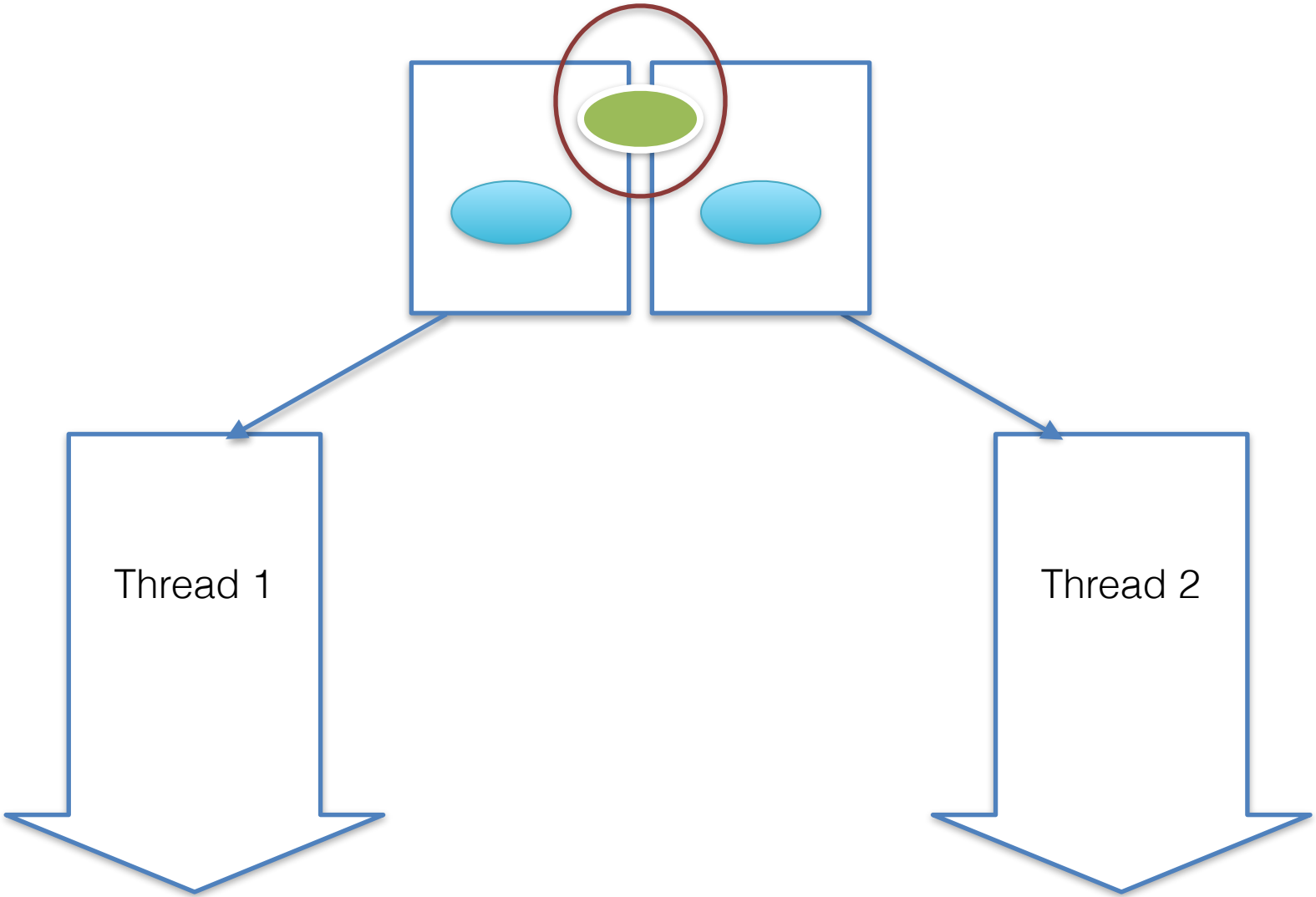
Shared memory

- Le classi che implementano `Runnable` o che estendono `Thread` sono classi normali
- Oltre al metodo `run()` ci sono attributi (altri oggetti) e metodi normali
- Non statici
  - Accessibili solo a un oggetto della classe
- Statici
  - Comuni a tutti gli oggetti della classe

Shared memory

- Un'oggetto usato in due thread diversi (regola 1):
  - tutti gli attributi sono condivisi dai thread
- Due oggetti della stessa Classe usati in due thread diversi (regola 2):
  - gli attributi statici sono condivisi
  - gli attributi non-statici sono privati ad ogni thread
- Attenzione agli oggetti non primitivi (le variabili sono referenze):
  - Classe C1 ha un attributo di tipo C2 (non primitivo) - vedi esempio sotto
- Tutte le variabili locali dei metodi sono privati ai *thread*


Thread 1
Thread 2


Thread 1
Thread 2

Caso `Runnable`

- Un oggetto `Runnable` definisce un *task* - ha bisogno di dati
- Si devono definire i dati condivisi e quei privati a ogni *thread* - fase di *design* del programma
- Se si usa un solo `Runnable` per più `Thread`
  - Tutti gli attributi sono shared
- Se si usa un `Runnable` per ogni `Thread`
  - Usare `static` per avere attributi *shared*
  - Non usare `static` per attributi primitivi locali ai *thread*
    - Usare un solo oggetto per attributi non-primitivi shared
    - Usare copie dei oggetti per attributi non-primitivi locali ai *thread*

```
public class DataThread implements Runnable{

  private String nonSharedString;

  public void run() {
    try{
        String me=Thread.currentThread().getName();
        System.out.println(me+": Setting my string
                                  to be equal to my name.");
        this.nonSharedString=me;
        System.out.println(me+": The string is "
               + this.nonSharedString;
        Thread.sleep(2000);
        System.out.println(me+": After sleeping the string is "
               + this.nonSharedString;
    }catch(InterruptedException e){}
  }
  //main goes here
}
```

Questo programma non è ancora corretto

```
public static void main(String[] args){

    DataThread d1=new DataThread();
    DataThread d2=new DataThread();
    Thread t1= new Thread(d1);
    Thread t2= new Thread(d2);
    t1.start();
    t2.start();
}
```

Output:

```
Thread-0: Setting my string to be equal to my name.
Thread-1: Setting my string to be equal to my name.

Thread-0: The string is Thread-0
Thread-1: The string is Thread-1

Thread-1: After sleeping the string is Thread-1
Thread-0: After sleeping the string is Thread-0
```

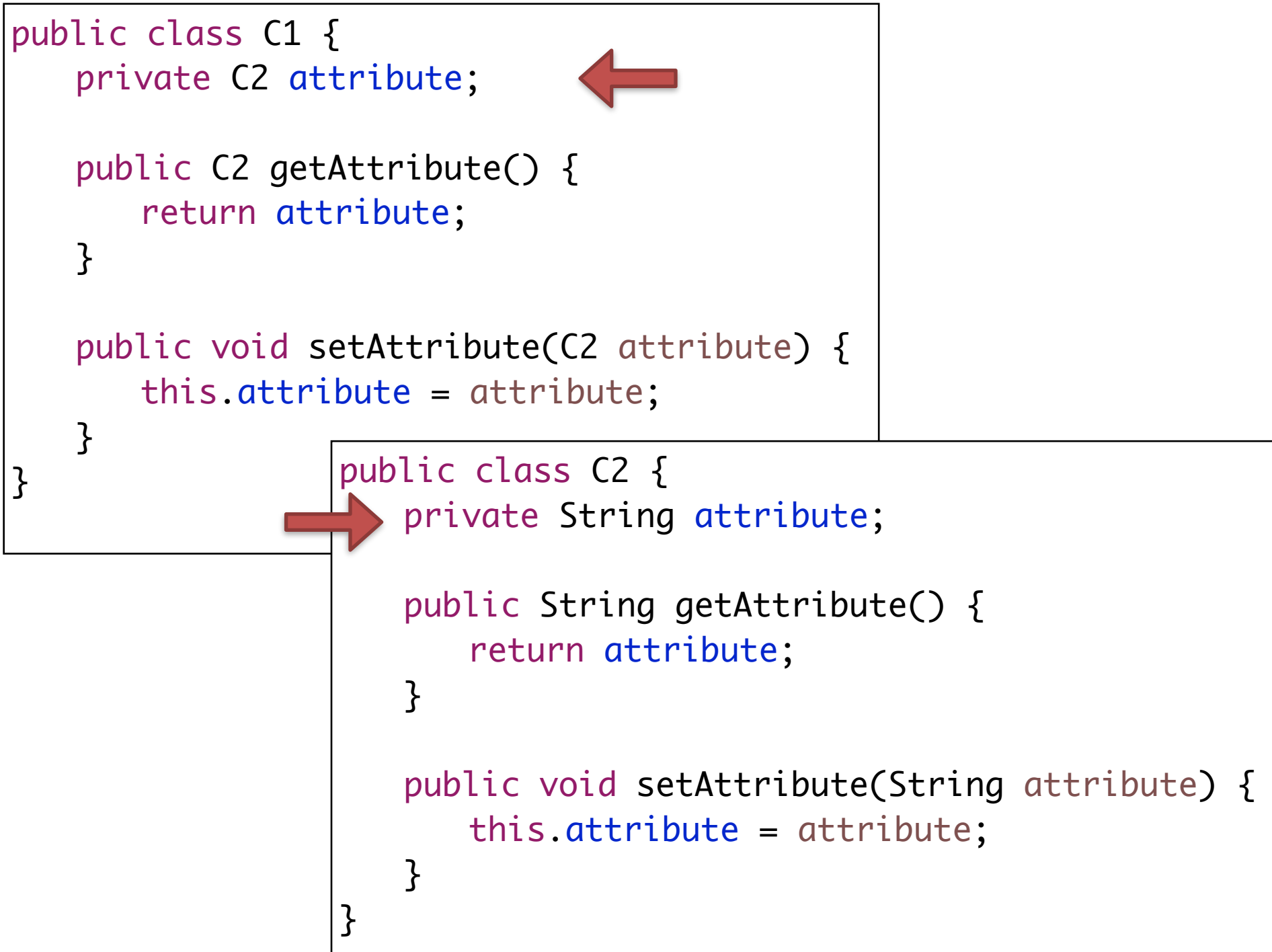
```
public class DataThread implements Runnable{

  private static String sharedString="";

  public void run() {
    try{
        String me=Thread.currentThread().getName();
        System.out.println(me+": Setting my string
                                  to be equal to my name.");
        DataThread.sharedString=me;
        System.out.println(me+": The string is "
               + DataThread.sharedString;
        Thread.sleep(2000);
        System.out.println(me+": After sleeping the string is "
               + DataThread.sharedString;
    }catch(InterruptedException e){}
  }
  //main goes here
}
```

Questo programma non è ancora corretto

```
public static void main(String[] args){

    DataThread d1=new DataThread();
    DataThread d2=new DataThread();
    Thread t1= new Thread(d1);
    Thread t2= new Thread(d2);
    t1.start();
    t2.start();
}
```

Output:

```
Thread-0: Setting my string to be equal to my name.
Thread-1: Setting my string to be equal to my name.

Thread-0: The string is Thread-0
Thread-1: The string is Thread-1

Thread-1: After sleeping the string is Thread-1
Thread-0: After sleeping the string is Thread-1
```

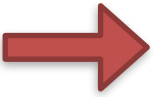

public class C1 {
private C2 attribute;
public C2 getAttribute() {
return attribute;
}
public void setAttribute(C2 attribute) {
this.attribute = attribute;
}
}
public class C2 {
private String attribute;
public String getAttribute() {
return attribute;
}
public void setAttribute(String attribute) {
this.attribute = attribute;
}
}

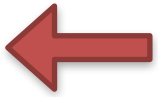
```
public class DataThread implements Runnable{

    private C1 trickyOne;


    public DataThread(C1 tricky){
        this.trickyOne=tricky;
    }
    //run and main go here
}
```

Questo programma non è ancora corretto

```
public void run() {
    try{
        String me=Thread.currentThread().getName();

        System.out.println(me+": Setting my string
                                    to be equal to my name.”);

        this.trickyOne.getAttribute().setAttribute(me);

        System.out.println(me+": The string is "
                + this.trickyOne.getAttribute().getAttribute();

        Thread.sleep(2000);

        System.out.println(me+": After sleeping the string is "
                + this.trickyOne.getAttribute().getAttribute();

    }catch(InterruptedException e){}
}
```

```
public static void main(String[] args) throws InterruptedException {
    C2 globalStr=new C2();
    globalStr.setAttribute("From main”);

    C1 str1= new C1();
    C1 str2= new C1();
    str1.setAttribute(globalStr);
    str2.setAttribute(globalStr);

    DataThread d1=new DataThread(str1);
    DataThread d2=new DataThread(str2);
    Thread t1= new Thread(d1);
    Thread t2= new Thread(d2);
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println("In main str1 is: “
              +str1.getAttribute().getAttribute());
    System.out.println("In main str2 is: “
              +str2.getAttribute().getAttribute());
}
```





Output:

```
Thread-0: Setting my string to be equal to my name.
Thread-1: Setting my string to be equal to my name.

Thread-0: The string is Thread-0
Thread-1: The string is Thread-1

Thread-1: After sleeping the string is Thread-1
Thread-0: After sleeping the string is Thread-1

In main global string in str1 is: Thread-1
In main global string in str2 is: Thread-1
```

Compito (per laboratorio)

- Calcolo pi greco
- Ufficio postale
- Testo su moodle (assignment 1 & 2)