UNIVERSITÀ DI PISA

Programmazione di Reti
Corso B

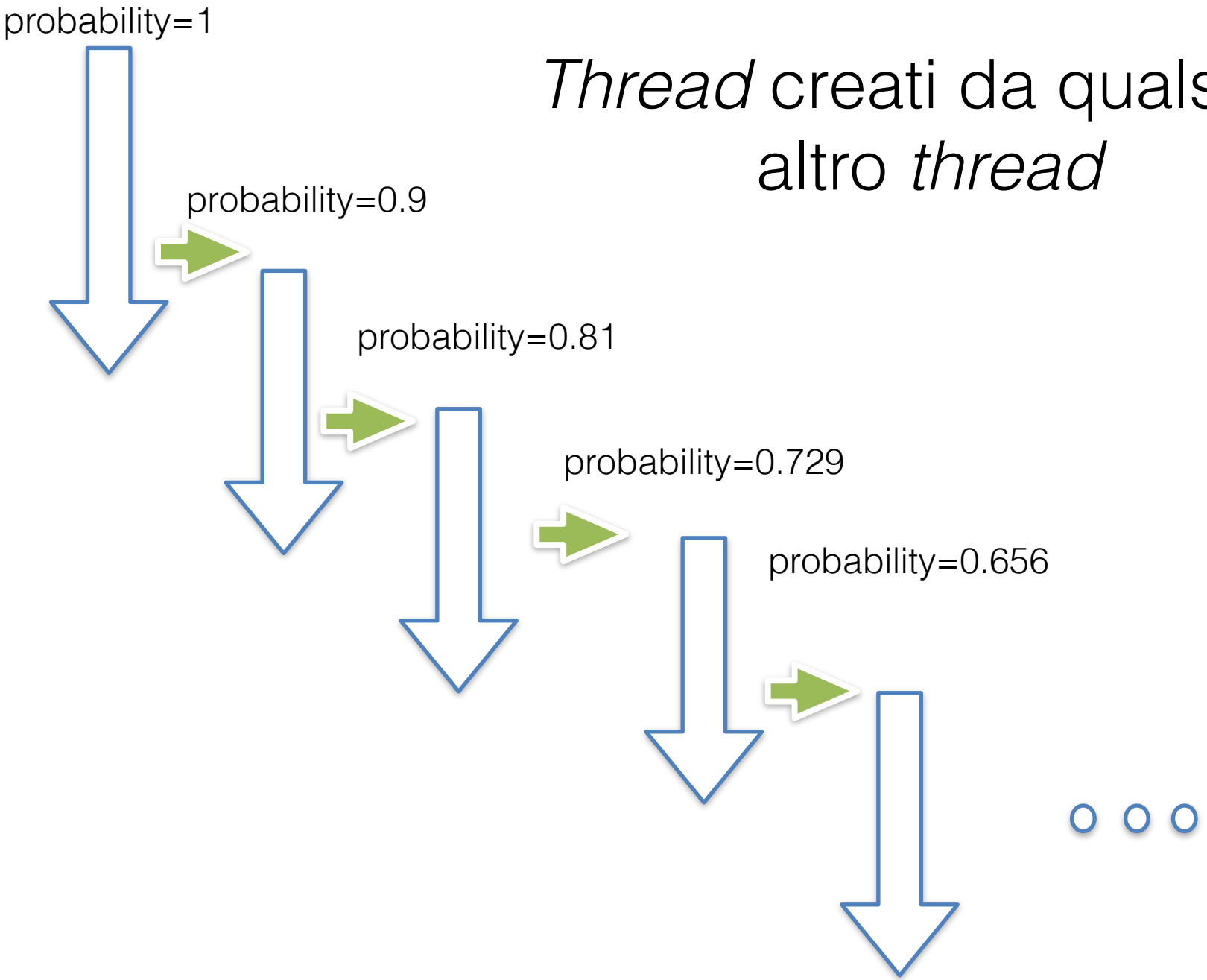
23 Febbraio 2016

Lezione 1(b)

Contenuti

- Ieri :
  - creare un *thread* - in due modi
- Oggi:
  - pausa, interruzione, aspettare thread
  - *shared memory*
  - sincronizzazione

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:

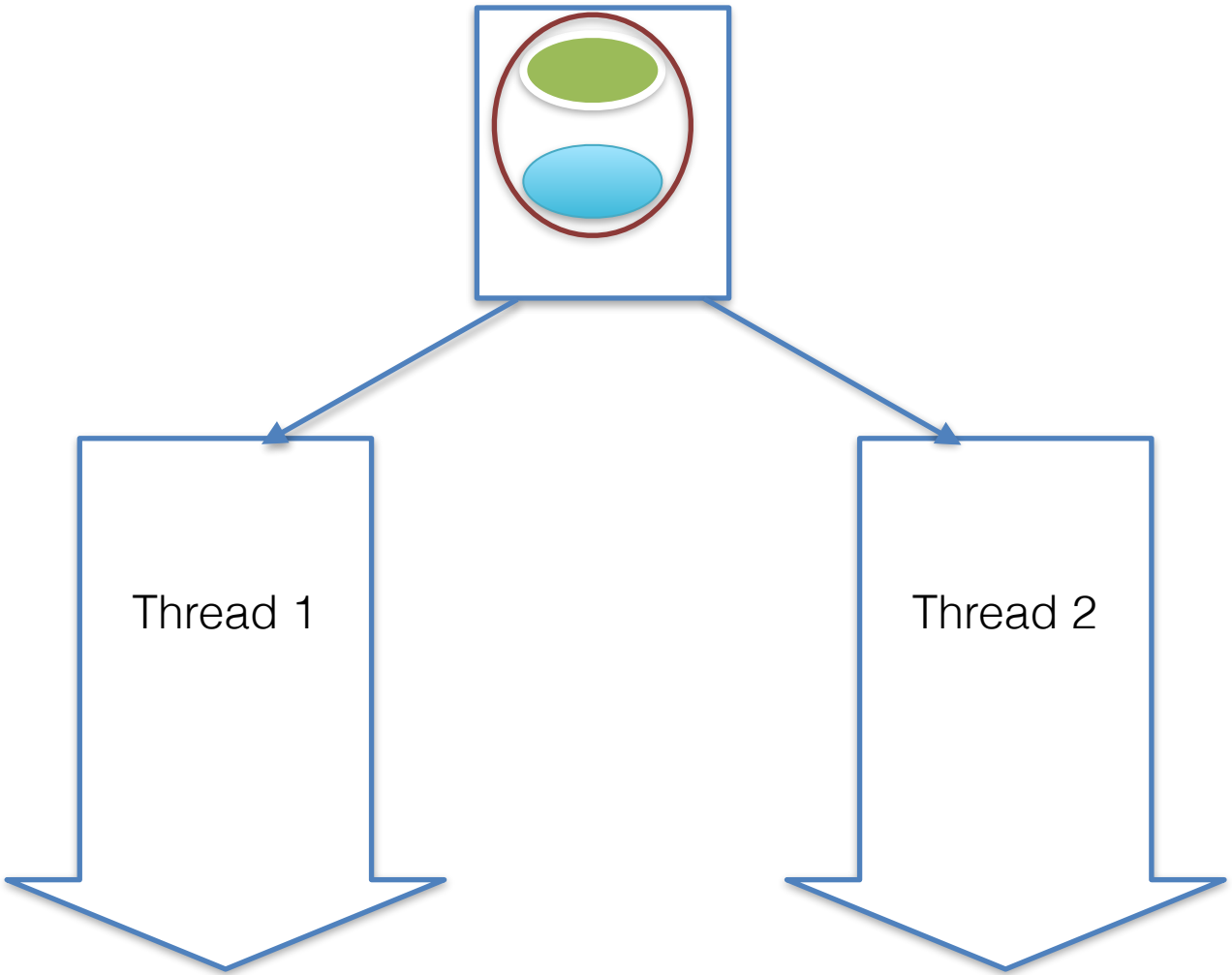
```
Thread[main,5,main]: Main thread started
Thread[Thread-0,5,main]: Just started
Thread[Thread-1,5,main]: Just started
Thread[Thread-2,5,main]: Just started
Thread[Thread-3,5,main]: Just started
Thread[Thread-4,5,main]: Just started
Thread[Thread-5,5,main]: Just started
Thread[Thread-6,5,main]: Just 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
```

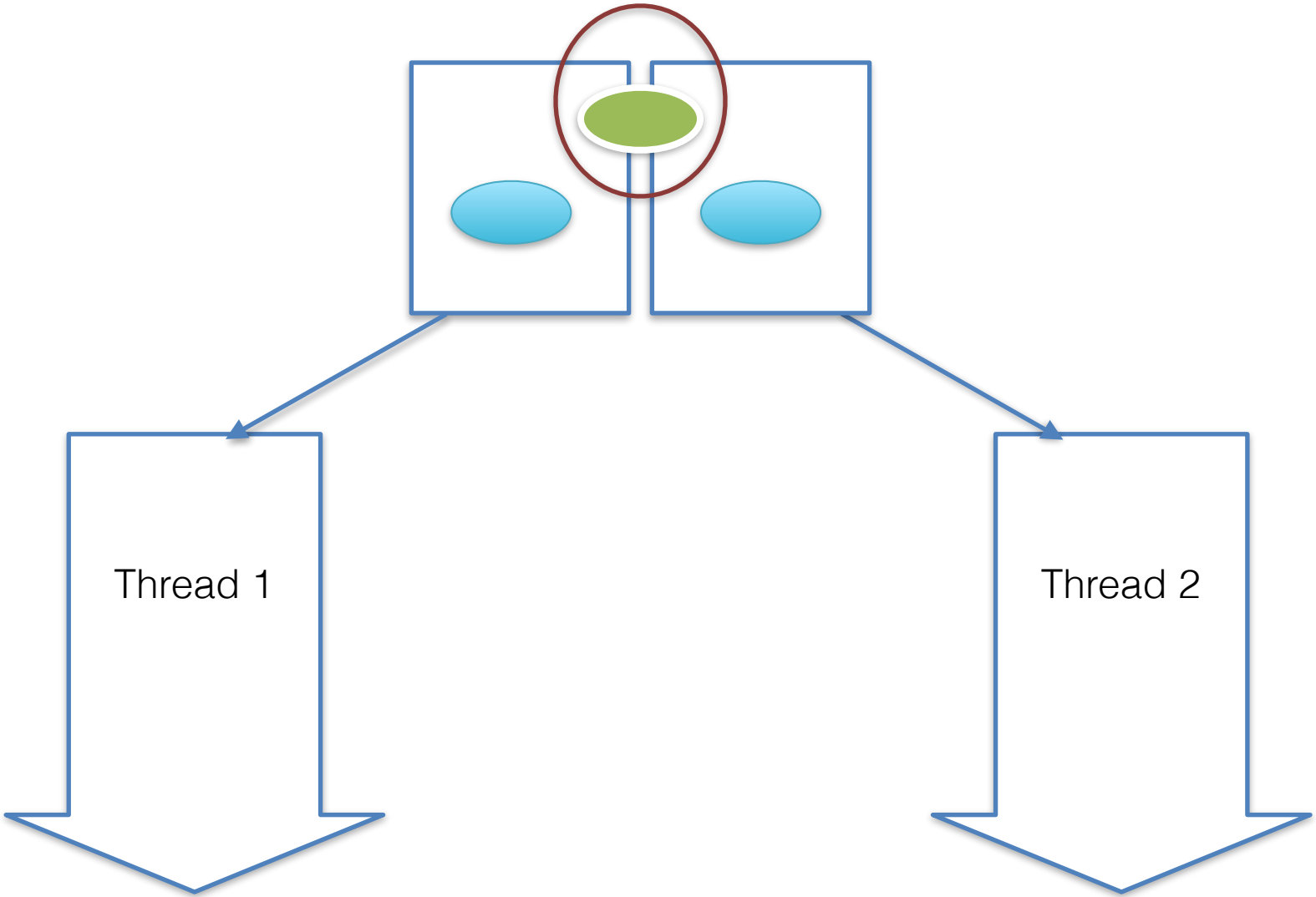
Shared memory

- 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
```

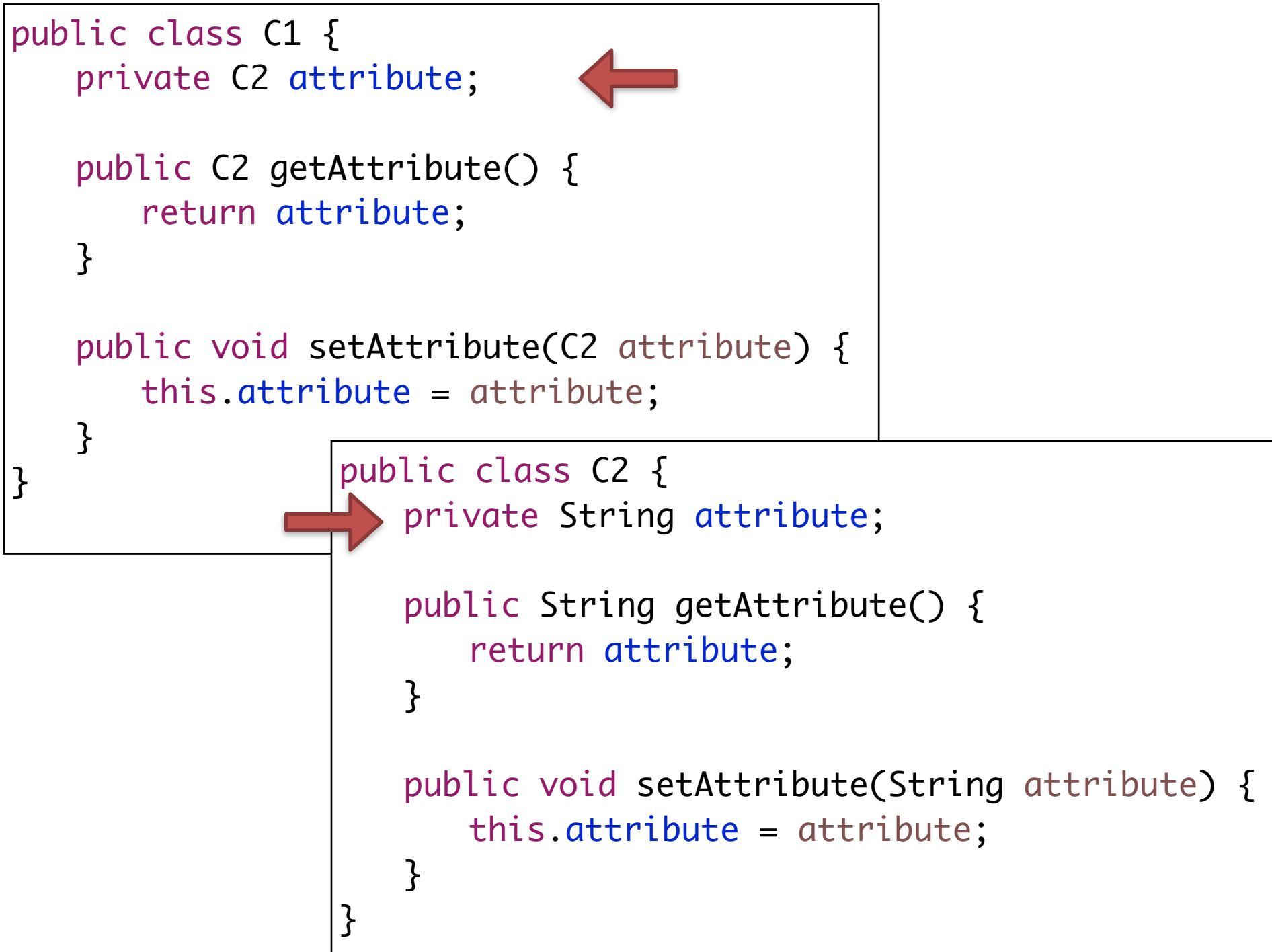
```
public class DataThread implements Runnable{

  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
```

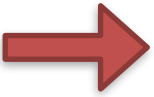

public class C1 {
private C2 attribute;
public C2 getAttribute() {
return attribute;
}
public void setAttribute(C2 attribute) {
this.attribute = attribute;
}
}
public class C2 {
private String attribute;
public String getAttribute() {
return attribute;
}
public void setAttribute(String attribute) {
this.attribute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

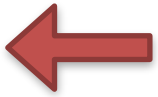
```
public class DataThread implements Runnable{

    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
```

Il caso delle matrici di prima

```
public static void main(String[] args) {
    //read two matrices from file
    ArrayList<ArrayList<Double>> m1, m2;
    m1=readMatrix("matrix1.txt");
    m2=readMatrix("matrix2.txt");

    //create runnables
    MultiplicationTask task1= new MultiplicationTask(m1,m2);
    AdditionTask task2= new AdditionTask(m1,m2);

    //start threads
    (new Thread(task1)).start();
    (new Thread(task2)).start();
    }
```

File

- Anche i file possono essere condivisi tra thread
- Un oggetto *file handler* per tutti gli *thread*
  - sia static che non static: ogni *thread* legge un sottoinsieme di righe, scrive in ordine aleatorie
- Un oggetto *file handler* per ogni *thread*
  - ogni *thread* legge tutto il file, e lo scrive in blocchi grandi in ordine aleatorie

```
public class ConcurrentFileReader implements Runnable {

    static BufferedReader reader;
    static BufferedWriter writer;

    public void run(){
        try {
            System.out.println(Thread.currentThread().getId()
                        +" running….");

            String line="";
            while (line!=null){
                line=ConcurrentFileReader.reader.readLine();
                ConcurrentFileReader.writer.write(
                        Thread.currentThread().getId()+” read: “
                        +line+"\n");
            }
            System.out.println(Thread.currentThread().getId()
                        +" finished.");
        } catch (IOException e) {e.printStackTrace();}
    }
    //main goes here
}
```

Questo programma non è ancora corretto

```
public static void main(String[] args) {
    try {
        ArrayList<Thread> threads= new ArrayList<Thread>();
        ConcurrentFileReader.reader = new BufferedReader(
                                new FileReader("file.txt"));
        ConcurrentFileReader.writer = new BufferedWriter(
                                new FileWriter("fileCopy.txt"));
        for (int i=0;i<5;i++){
            ConcurrentFileReader r=new ConcurrentFileReader();
            Thread t= new Thread(r);
            t.start();
            threads.add(t);
        }
        for (Thread t : threads){
            t.join();
        }//all threads are done, I can close files
        ConcurrentFileReader.reader.close();
        ConcurrentFileReader.writer.close();
    } catch (Exception e) {e.printStackTrace();}
}
```

Output file:

```
9 read: line    1
13 read: line  2
9 read: line    3
13 read: line  4
9 read: line    5
13 read: line  6
13 read: line  8
13 read: line  9
13 read: line  10
9 read: line    7
9 read: line    12
13 read: line  11
9 read: line    13
13 read: line  14
9 read: line    16
13 read: line  17
11 read: line  15
9 read: line    18
13 read: line  19
11 read: line  20
9 read: line    21
13 read: line  22
11 read: line  23
13 read: line  24
```


KEEP
CALM
AND
ASK ME
QUESTIONS

Shared memory

- *Shared memory* molto efficiente per scambiare informazioni
- Suscettibile agli errori
  - interferenza dei thread
    - errori di calcolo quando thread multipli usano la stessa variabile shared

Interferenza (*race condition*)

- E.g. Applicazione bancaria
  - Thread 1 : legge il saldo del conto corrente e aggiunge gli interessi (1%)
  - Thread 2: legge il saldo e aggiunge lo stipendio
  - Sfortuna:
    - Thread 1 legge il saldo ($100)
    - Thread 2 legge il saldo ($100)
    - Thread 2 aggiunge stipendio ($1100)
    - Thread 1 aggiunge interessi ($101)
    - !!!!!!!!!

Shared memory

- Suscettibile agli errori
  - memoria incoerente (*memory consistency errors*)
    - variabili non volatili - il nuovo valore potrebbe non essere visibile agli *reader thread* subito dopo il *write*
    - volatile fa il nuovo valore visibile subito

```java
public class B implements Runnable{
    private Counter c;

    public B(Counter c){
        this.c=c;
    }
    public void run() {
        System.out.println(Thread.currentThread()+": "+c.getCount());
    }
}
```

Questo programma non è ancora corretto

```java
public class A  implements Runnable{

    private Counter c;

    public A(Counter c){
        this.c=c;
    }
    public void run() {
        this.c.increment();
    }
}
```

```java
public class Counter {
    private int count;

    public Counter(){
        this.count=0;
    }

    public void increment(){
        this.count++;
    }

    public int getCount() {
        return count;
    }
}
```

```
public static void main(String[] args) {
    Counter c= new Counter();
    (new Thread(new A(c))).start();
    (new Thread(new B(c))).start();
    (new Thread(new A(c))).start();
    (new Thread(new B(c))).start();

}
```

```
Output:
Thread[Thread-1,5,main]: 2
Thread[Thread-3,5,main]: 2
```

Sincronizzazione

- Soluzione per tutti i problemi
- Basata su l'esistenza di operazioni atomici : operazioni che succedono in un unico passo, non e possibile interrompere il thread nel mezzo
    - read/write referenze e variabile primitive (no long e double) `a=1;`
    - read/write tutti variabili *volatile*
    - a++ non e atomica

Sincronizzazione

- Evitare errori di calcolo
  - Definire sezioni critici : set di istruzioni che possono essere eseguite da un solo thread alla volta
- Evitare memoria incoerente
  - Proteggere dati condivisi usando i lock
  - Ogni accesso a variabili condivisi deve essere protetto (anche read), se c'è almeno un thread che modifica il oggetto!
- Fermare i thread quando non hanno niente da fare - usando condizioni, semafori, join, etc.
- **Tutti i programmi di questa lezione dovevano usare la sincronizzazione!**

Mutex

- *Mutual exclusion lock* : *lock* che può essere acquisito da un solo *thread* alla volta
- ogni oggetto java (derivato da Object, no int or double) ha un *mutex* incluso, chiamato *monitor* o *lock* “intrinseco”
- Il monitor può essere acquisito usando la parola chiave `synchronized`

Metodi `synchronized`

- Il più semplice modo di proteggere i dati

```
public synchronized void setName(String name) {
    this.name = name;
}
```

- Solo un thread alla volta può richiamare un metodo synchronized di un oggetto

- Gli oggetti diversi sono protetti da monitor diversi, quindi lo stesso metodo può essere richiamato in contemporaneo per due oggetti diversi

```
public class Student {
    private String name;
    private int math_grade, prog_grade;

    public Student(String name){
        this.name=name;
        this.math_grade=0;
        this.prog_grade=0;}

    public synchronized String getName() {return name;}
    public synchronized void setName(String name) {this.name = name;}

    public synchronized int getMath_grade() {return math_grade;}
    public synchronized void setMath_grade(int math_grade) {
        this.math_grade = math_grade;}

    public synchronized int getProg_grade(){return progr_grade;}
    public synchronized void setProg_grade(int prog_grade){
        this.prog_grade = prog_grade;}

    public synchronized double getAverage(){
        return (this.math_grade+this.prog_grade)/2;}
}
```

Student class

- *Synchronized object*
- Siamo sicuri che nessun *read* o *write* sarà fato in contemporaneo con un altro
- Non ci saranno problemi di memoria invalida

```
public class Professor implements Runnable{

    private String subject;
    private ArrayList<Student> students;

    public Professor(String subject,ArrayList<Student> students){
        this.subject=subject;
        this.students=students;
    }

    public void run() {
        try{
            for (Student s: this.students){
                long sleeptime=System.nanoTime()%10;
                Thread.sleep(sleeptime*1000);
                this.grade(s);
                System.out.println(Thread.currentThread()+" graded "
                +s.getName()+ " average is " + s.getAverage());
            }
        }catch (InterruptedException e){}
    }
    //grade method goes here
}
```

```
private void grade(Student s){
    switch(subject)
    {
    case "math":
        s.setMath_grade(25);
        break;
    case "prog":
        s.setProg_grade(29);
        break;
    default: break;
    }
}
```

```
public static void main(String[] args) throws
InterruptedException {
    ArrayList<Student> students= new ArrayList<Student>();
    students.add(new Student("Alina Sirbu"));
    students.add(new Student("Alessandro Lulli"));

    Professor prof1= new Professor("math",students);
    Professor prof2= new Professor("prog",students);

    Thread t1= new Thread(prof1);
    Thread t2= new Thread(prof2);
    t1.start();
    t2.start();
    t1.join();
    t2.join();

    for (Student s: students){
        System.out.println(s.getName()+ " : " + s.getAverage());
    }
}
```

Output:

```
Thread[Thread-1,5,main] graded Alina Sirbu average is 14.0
Thread[Thread-0,5,main] graded Alina Sirbu average is 27.0
Thread[Thread-0,5,main] graded Alessandro Lulli average is 27.0
Thread[Thread-1,5,main] graded Alessandro Lulli average is 27.0
Alina Sirbu : 27.0
Alessandro Lulli : 27.0
```

Metodi synchronized

- La classe di un oggetto e un oggetto
- `Student.class` ha il suo monitor
- Un metodo *static* sincronizzato
  - acquisisce il monitor della classe (non del oggetto)
  - protegge tutti i membri static della classe

Blocchi synchronized

- Stessa parola chiave:

```
synchronized(o)
```

- Metodo synchronized equivalente con `synchronized(this)`:

```
public void setMath_grade(int math_grade) {
        synchronized(this){
            this.math_grade = math_grade;
        }
}
```

Blocchi *synchronized*

- Sincronizzazione dettagliata
- I due prof lavorano su attributi diversi
- Possiamo usare un monitor di un oggetto diverso per ogni attributo di Student
- Per il nome possiamo usare il monitor del nome stesso
- Dobbiamo acquisire i due monitor dei voti insieme quando calcoliamo la media.

```
public class Student {
    private String name;
    private int math_grade, prog_grade;
    private Integer math_lock, prog_lock, name_lock;

    public Student(String name){
        this.name=name;
        this.math_grade=0;
        this.prog_grade=0;
        this.math_lock=new Integer(0);
        this.prog_lock=new Integer(0);
        this.name_lock=new Integer(0);
    }
    public String getName() {
        synchronized(this.name_lock){
            return name;
    }}
    public void setName(String name) {
        synchronized(this.name_lock){
            this.name = name;
    }}
```



```
public int getMath_grade() {
    synchronized(this.math_lock){
        return math_grade;
}}
public void setMath_grade(int math_grade) {
    synchronized(this.math_lock){
        this.math_grade = math_grade;
}}

public int getProg_grade() {
    synchronized(this.prog_lock){
        return prog_grade;
}}
public void setProg_grade(int programming_grade) {
    synchronized(this.prog_lock){
        this.prog_grade = programming_grade;
}}

public double getAverage(){
    synchronized(this.prog_lock){
        synchronized(this.math_lock){
            return (this.math_grade+this.prog_grade)/2;
        }
    }
}}
```

Variabili di condizione

- Secondo meccanismo di sincronizzazione
- Quando un blocco di codice deve procedere solo se una condizione è soddisfatta (*guarded blocks)*
- Non dobbiamo controllare la condizione in un *busy loop* - consuma risorse

```
while(!condition){}
```

- Possiamo fare aspettare il thread
- Il thread verra notificato automaticamente quando c'è stato un cambiamento di stato

Variabili di condizione

- Struttura del codice:

```
synchronized(o){
        while(!condition){
            o.wait();
        }
        //make changes to o and other data
        o.notify();
    }
```

Variabili di condizione

- Ogni oggetto Java ha una variabile di condizione associata
- Verifica della condizione in ambiente protetto - blocco `synchronized`
- `o.wait()` : Thread va in sleep quando la condizione e falsa
- `o.notify()`: Un thread in aspettativa e svegliato - deve verificare di nuovo se la condizione è vera
- `o.notifyall()`: Tutti i thread in aspettativa sono svegliati
- il monitor e la variabile di condizione dello stesso oggetto sono legati
  - non puoi fare `o.wait()` o `o.notify()` prima di fare `synchronized(o)`

Produttore - consumatore

- Paradigma molto comune in multithreading
- Produttore: crea dati, risultati.
- Consumatore: usa i dati o risultati.
- Esempio semplice:
  - Produttore produce un String
  - Consumatore consuma il String (facendolo sparire)
  - Il consumatore aspetta se il String e vuoto
  - Il produttore aspetta se il String non è stato consumato
  - Mamma , bambino e il cibo

```
public class Food {
    String food;

    public synchronized boolean isEmpty() {
        return food.equals("");
    }

    public synchronized void cook(String food) {
        this.food = food;
    }

    public synchronized void eat(){
        this.food="";
    }

    public Food(){
        this.food="";
    }
}
```

```
public class Mother implements Runnable{
    Food food;
    public Mother(Food food){this.food=food;}
    private void print(String m){
        System.out.println(System.currentTimeMillis()/1000+ " - Mother: "+m);
    }
    public void run() {
        try{
            this.print("Created");
            while(true){
                synchronized(this.food){
                    while(!this.food.isEmpty()){
                        this.print("Waiting for baby to eat");
                        this.food.wait();
                    }
                    this.print("No food left, cooking");
                    this.food.cook("Soup");
                    this.food.notify();
                }
                this.print("Now resting a bit");
                Thread.sleep(10000);
            }
        }catch (InterruptedException e){this.print("Interrupted"); return;}
}}
```

Acquisisce monitor

Lascia monitor

Acquisisce monitor

Lascia monitor

Reentrant lock

- Il monitor di un oggetto può essere acquisito varie volte dallo stesso *thread* - rientrante
- Nel nostro esempio, nel metodo `run` sono richiamati metodi *synchronized* del oggetto `Food` in un blocco *synchronized* sullo stesso oggetto
- Questo è possibile a causa del meccanismo dei *lock* rientranti
- Altrimenti il programma blocca se stesso
- Tutti i *lock* in Java sono rientranti

```
synchronized(this.food){
    while(!this.food.isEmpty()){
        this.print("Waiting for baby to eat");
        this.food.wait();
    }
    this.print("No food left, cooking");
    this.food.cook("Soup");
    this.food.notify();
}
```

```
public synchronized void cook(String food) {
        this.food = food;
    }
```

```
public class Child implements Runnable {
    Food food;
    public Child(Food food){this.food=food;}
    private void print(String m){
        System.out.println(System.currentTimeMillis()/1000+ "- Child: "+m);
    }
    public void run() {
        try{
            this.print("Just born");
            while(true){
                this.print("I am hungry, I want to eat.");
                synchronized(food){
                    while (this.food.isEmpty()){
                        this.print("There is no food :(");
                        food.wait();
                    }
                    this.print("Yay, eating now :)");
                    food.eat();
                    food.notify();
                }
                this.print("Going back to sleep");
                Thread.sleep(5000);
            }
        }catch (InterruptedException e){ this.print(“Interrupted");
            return;}
}}
```

```
public static void main(String[] args) {
    try{
        Food food= new Food();
        Mother mother = new Mother(food);
        Child child= new Child(food);
        Thread motherThread=new Thread(mother);
        Thread childThread= new Thread(child);
        motherThread.start();
        childThread.start();
        Thread.sleep(20000);
        motherThread.interrupt();
        childThread.interrupt();
    } catch(InterruptedException e){}

}
```

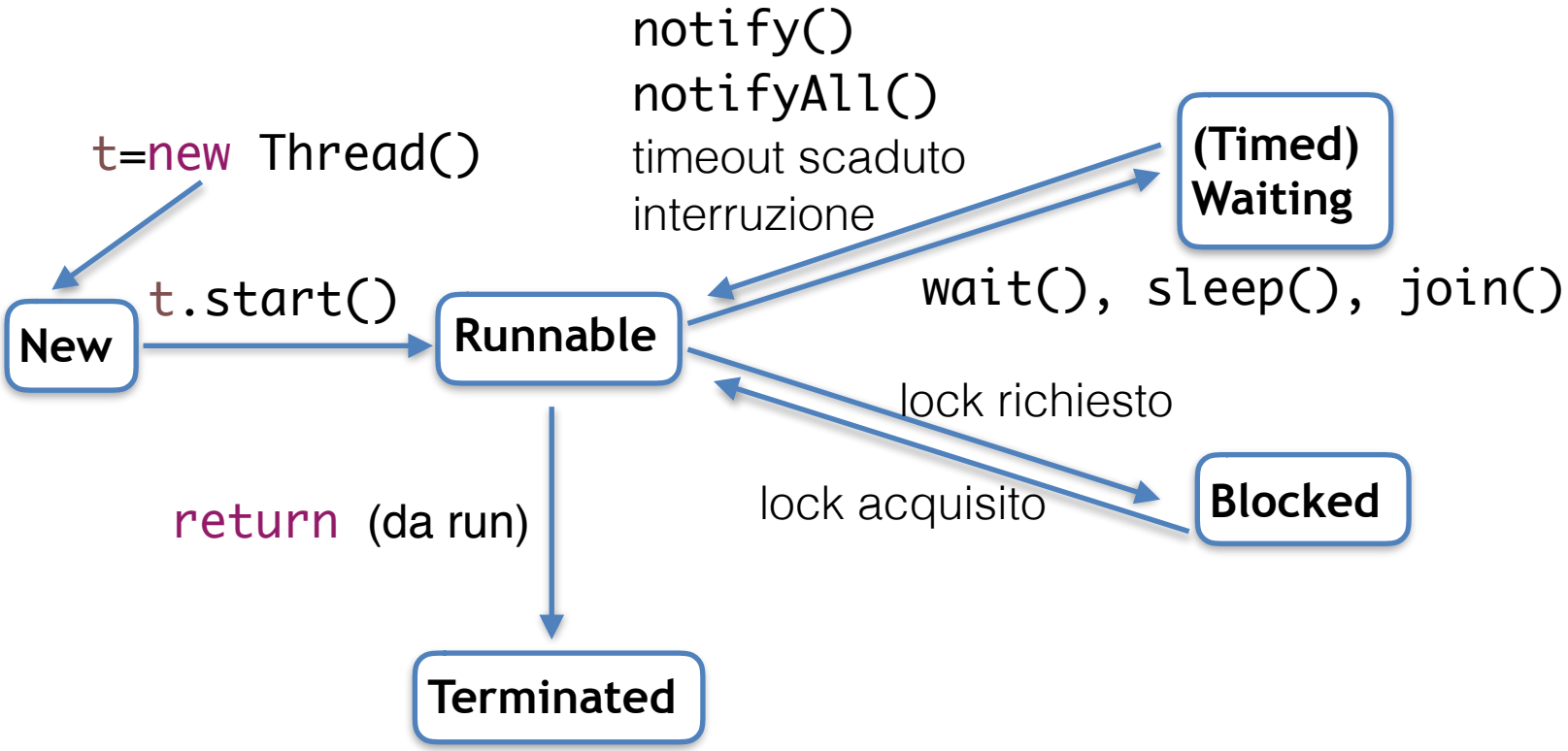
```
1455968233- Child: Just born
1455968233- Child: I am hungry, I want to eat.
1455968233 - Mother: Created
1455968233- Child: There is no food :(
1455968233 - Mother: No food left, cooking
1455968233 - Mother: Now resting a bit
1455968233- Child: Yay, eating now :)
1455968233- Child: Going back to sleep
1455968238 - Mother: No food left, cooking
1455968238 - Mother: Now resting a bit
1455968243- Child: I am hungry, I want to eat.
1455968243- Child: Yay, eating now :)
1455968243- Child: Going back to sleep
1455968243 - Mother: No food left, cooking
1455968243 - Mother: Now resting a bit
1455968248 - Mother: Waiting for baby to eat
1455968253 - Mother: Interrupted
1455968253- Child: Interrupted
```

```
1455968342- Child: Just born
1455968342- Child: I am hungry, I want to eat.
1455968342- Child: There is no food :(
1455968342 - Mother: Created
1455968342 - Mother: No food left, cooking
1455968342 - Mother: Now resting a bit
1455968342- Child: Yay, eating now :)
1455968342- Child: Going back to sleep
1455968347- Child: I am hungry, I want to eat.
1455968347- Child: There is no food :(
1455968352 - Mother: No food left, cooking
1455968352 - Mother: Now resting a bit
1455968352- Child: Yay, eating now :)
1455968352- Child: Going back to sleep
1455968357- Child: I am hungry, I want to eat.
1455968357- Child: There is no food :(
1455968362- Child: Interrupted
1455968362 - Mother: Interrupted
```

Conclusioni

- Creare thread molto facile
- Sincronizzazione importante!
- Sempre proteggere dati condivisi!
- 2 meccanismi molto semplici di sincronizzazione
  - molto potenti
  - serve attenzione
  - serve creatività

Thread state machine



Esercizi

- **Esercizio 1: Word lookup.**

- Dobbiamo verificare se un file di testo molto grande contiene una parola.
- Scrivete un programma che usa 5 thread diversi per fare la ricerca.
- La parola chiave viene fornita come **parametro alla linea di comando**.
- Ogni thread legge una riga alla volta e cerca la parola.
- Il thread che trova la parola chiave per primo deve interrompere tutti gli altri.
- Attenzione: L'accesso al file di input deve essere sincronizzato!

Esercizi

- **Esercizio 2: Prendere la parola.**
- Un gruppo di n studenti organizzano una conferenza.
- Ogni uno deve prendere la parola una volta, poi va a casa. Può parlare solo uno studente alla volta.
- Modellare gli studenti con i thread.
  - Quando uno studente prende la parola, scrive il suo discorso in una variabile String. Consideriamo che il discorso è semplicemente il suo nome.
  - Prima che un altro studente possa parlare, tutti gli studenti rimasti devono leggere il discorso dell'ultimo speaker (si devono contare gli studenti che hanno letto)
  - Alla fine è rimasto uno solo studente che deve parlare. Scrive il suo nome è si spegne. Nessuno deve leggere, l'ultimo ha parlato da solo!