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

Programmazione Concorrente e Distribuita

Problemi tipici della programmazione concorrente e come evitarli

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it

Nondeterminismo

- Come mostrato negli esempi visti in precedenza, il numero di diversi possibili percorsi di esecuzione (non controllati dal programmatore) di un programma concorrente può essere molto grande anche per programmi molto semplici con un numero molto piccolo di thread.
- Questo aspetto prende il nome di nondeterminismo: il programmatore non può determinare l'ordine assoluto dell'esecuzione delle istruzioni appartenenti a trhead diversi.
- Il nondeterminismo implica che testare un programma concorrente è solitamente difficile.

Race Conditions

- Nei programmi concorrenti molti problemi sono causati dalle cosiddette “corse critiche” (o Race Conditions):

  tutte quelle situazioni in cui thread diversi operano su una risorsa comune, ed in cui il risultato viene a dipendere dall’ordine in cui essi effettuano le loro operazioni.

Non aver pensato alle Race Condition può causare problemi seri...

Non succede niente di così spettacolare, ma le conseguenze sono spesso serie.

Race Conditions: un semplice esempio

```
public class Counter {
   long count = 0;
   public void add (long value) {
      long tmp = this.count;
      tmp = tmp + value;
      this.count = tmp ;
   }
}
```

- Questo codice (molto semplice) può produrre risultati sbagliati se eseguito da più thread contemporaneamente!

Race Conditions: un semplice esempio

```
public class RaceExample extends Thread {
  Counter myCounter;
  public RaceExample(Counter c){
    myCounter=c;
  }
  public void run() {
    for(int i=0; i<10000; i++)
       myCounter.add(1);
    }
  public static void main(String args[])
      throws InterruptedException {
    Counter counter = new Counter();
    RaceExample p1 = new RaceExample(counter);
    RaceExample p2 = new RaceExample(counter);
    p1.start();  p2.start();
    p1.join(); p2.join();
    System.out.println("Counter = " + counter.count);
}}
```

Race Conditions: un semplice esempio

- Ci aspettiamo che alla fine, avendo chiamato 20000 volte il metodo add(1), count valga 20000.
- Provate e vedrete ...

Race condition: cosa va storto

```
public class Counter {
    long count = 0;
    public void add (long value) {
        long tmp = this.count;
        tmp = tmp + value;
        this.count = tmp ;
    }
}
```

Tempo

Task1	Task2
`tmp=this.count; //tmp=119`	
	`tmp=this.count; //tmp=119`
	`tmp=tmp+value; //tmp=120`
	`this.count=tmp; //count=120`
`tmp=tmp+value; //tmp=120`	
`this.count=tmp; //count=120`	

Abbiamo eseguito 2 volte il metodo add(1), una volta in task 1 e una volta in task 2, ma count è passato da 119 a 120, anziché diventare 121!

Race Conditions: quando può verificarsi?

- Sono necessarie due condizioni perché il programma possa fornire risultati diversi in diverse esecuzioni:
  1. Una risorsa deve essere condivisa tra due Thread (la variabile **`count`** nell'esempio precedente)
  2. Deve esistere almeno un percorso di esecuzione tra i Thread in cui una risorsa è condivisa in modo non sicuro (nell'esempio precedente il codice di **`add`**)

```
public class Counter {
    long count = 0;
    public void add (long value) {
        long tmp = this.count;
        tmp = tmp + value;
        this.count = tmp;
    }
}
```

Sezione critica

lettura

scrittura

il codice di un altro task può essere eseguito tra la lettura e la scrittura.

Cosa cambia se...

- Riscriviamo il codice in questo modo:

```
public class Counter {
    long count = 0;
    public void add (long value) {
      this.count += value;
    }
}
```

- **Non cambia nulla**.
  - I thread vengono interrotti a livello delle istruzioni di byte code, non a livello di istruzioni Java.
  - In altre parole, l'istruzione `this.count += value` non è atomica!

Race Conditions: come assicurare la correttezza del software?

- Non si può dimostrare la correttezza dei programmi concorrenti attraverso dei test fatti nel modo classico...
  - Se anche il programma si comporta correttamente N volte, la N+1-esima volta potrebbe verificarsi una sequenza di istruzioni diversa da tutte le N precedenti, e potrebbe essere proprio quella in cui si manifesta l'errore
- Per dimostrare la correttezza di un programma concorrente esistono due modi:
  - Dimostrare che il programma soddisfa tutte le condizioni sufficienti perché possa essere considerato sicuro.
    - Ma dimostrare *a posteriori* che il software soddisfa le condizioni è spesso troppo costoso...
  - Sviluppare il programma utilizzando strategie note che permettano di **evitare a priori situazioni di Race Conditions**.
    - Fare in modo che la risorsa condivisa sia in uno stato sicuro **prima** di consentire ad un altro thread di accedervi.
    - Bloccare l'accesso alle risorse condivise mentre sono in uno stato non sicuro è alla base delle strategie per ottenere programmi concorrenti sicuri

Come rendere sicuro il programma?

```
public class Counter {
    long count = 0;
    public void add (long value) {
        long tmp = this.count;
        tmp = tmp + value;
        this.count = tmp ;
    }
}
```

Qui bisogna bloccare

Qui si deve sbloccare

- L'oggetto **`counter`** (istanza di **`Counter`**) è condiviso.
  - C'è una «sezione critica» in cui avviene la lettura e scrittura (in tempi successivi) di una variabile condivisa.
- È necessario un meccanismo che blocca l'accesso alla sezione critica quando un thread entra, e lo sblocca quando il thread ne esce.
- In questo modo solo un thread alla volta può essere nella sezione critica.

Come bloccare l'accesso agli oggetti

- Con un semaforo
  - Il primo thread che accede alla risorsa condivisa blocca l'accesso mediante il semaforo
  - Quando ha finito di usare la risorsa, sblocca l'accesso, consentendo ad un altro thread di accedere
  - Questo a sua volta bloccherà eventuali altri thread
  - Ecc.
- Il semaforo è binario (rosso o verde).
  - **`acquire`** acquisisce l'accesso bloccando altri task
    - eventualmente ponendo in attesa il task chiamante, se il semaforo è rosso.
  - **`release`** rilascia la risorsa
    - ed eventualmente rende ready un task in attesa

Tipi di semafori

- Contatori
  - Il semaforo viene inizializzato a un valore intero positivo
  - A ogni richiesta viene decrementato
    - Solo quando è zero il thread che fa una richiesta viene bloccato
  - A ogni uscita viene incrementato
    - E si sblocca un thread in attesa (se c'è)
- Binari (alias mutex)
  - Il semaforo può essere solo zero o uno (rosso o verde)

- Java fornisce un semaforo generico n-ario mediante la classe **`java.util.concurrent.Semaphore`**.
  - Un mutex è un semaforo binario istanziato con valore 1.

Il codice a prova di race condition

```
import java.util.concurrent.Semaphore;
public class RaceExample extends Thread {
  private Counter myCounter;
  private Semaphore theSemaphore;
  public RaceExample(Counter c, Semaphore s){
    myCounter=c;
    theSemaphore=s;
  }
  public void run() {
    for(int i=0; i<10000; i++) {
      try {
        theSemaphore.acquire();
      } catch (InterruptedException e){ }
      myCounter.add(1);
      theSemaphore.release();
    }
  }
```

Prima di accedere all'istanza condivisa di **`Counter`** si acquisisce il **`lock`**.
Ci sarà sempre un solo thread che accede ad **`add`**.

Avendo finito di usare l'istanza condivisa di **`Counter`** si rilascia il lock.

Il codice a prova di race condition

```
  public static void main(String args[])
                              throws InterruptedException {
    Counter counter = new Counter();
    Semaphore sem = new Semaphore(1);
    RaceExample p1 = new RaceExample(counter, sem);
    RaceExample p2 = new RaceExample(counter, sem);
    p1.start();
    p2.start();
    p1.join();
    p2.join();
    System.out.println("Counter = " + counter.count);
  }
}
```

Codice a prova di race condition: alternativa

```
import java.util.concurrent.Semaphore;
public class Counter {
  private long count = 0;
  private Semaphore sem;
  public Counter() {
    sem = new Semaphore(1);
  }
  public void add (long value) { this.count += value; }
  public void lock(){
    try {
      sem.acquire();
    } catch (InterruptedException e) { }
  }
  public void unlock(){
    sem.release();
  }
  public long getValue(){
    return count;
  }
}
```

L'uso del semaforo è confinato nella classe **Counter**.

Codice a prova di race condition: alternativa

```
public class RaceExample extends Thread {
  private Counter myCounter;
  public RaceExample(Counter c){
    myCounter=c;
  }
  public void run() {
    for(int i=0; i<10000; i++) {
      myCounter.lock();
      myCounter.add(1);
      myCounter.unlock();
    }
  }
  public static void main(String args[])
                          throws InterruptedException {
    Counter counter = new Counter();
    RaceExample p1 = new RaceExample(counter);
    RaceExample p2 = new RaceExample(counter);
    p1.start();  p2.start();
    p1.join(); p2.join();
    System.out.println("Counter = " + counter.getValue());
}}
```

Prima di accedere all'istanza condivisa di **Counter** si acquisisce il lock.
Ci sarà sempre un solo thread che accede ad **add**.

Avendo finito di usare l'istanza condivisa di **Counter** si rilascia il lock.

Limiti delle soluzioni viste

- In entrambi i casi, il codice del Thread deve ricordarsi di gestire la corsa critica
  - Direttamente mediante semaforo
  - Chiamando lock e unlock

- Possiamo togliere al Thread l'incombenza di occuparsi della sezione critica?
  - Cioè renderla "trasparente" al thread?

Codice a prova di race condition: codice thread-safe

```
import java.util.concurrent.Semaphore;
public class Counter {
  private long count = 0;
  private Semaphore sem;
  public Counter() {
    sem = new Semaphore(1);
  }
  public void add (long value) {
    try {
      sem.acquire();
    } catch (InterruptedException e) { }
    this.count+=value;
    sem.release();
  }
  public long getValue(){
    return count;
  }
}
```

Il semaforo è usato (solo) nel contesto del metodo **add**.

Codice a prova di race condition: codice thread-safe

```
public class RaceExample extends Thread {
  private Counter myCounter;
  public RaceExample(Counter c){
    myCounter=c;
  }
  public void run() {
    for(int i=0; i<10000; i++) {
      myCounter.add(1);
    }
  }
  public static void main(String args[])
                          throws InterruptedException {
    Counter counter = new Counter();
    RaceExample p1 = new RaceExample(counter);
    RaceExample p2 = new RaceExample(counter);
    p1.start();  p2.start();
    p1.join(); p2.join();
    System.out.println("Counter = " + counter.getValue());
  }
}
```

Non ci preoccupiamo di possibili race condition.
Ci pensa la classe **Counter**.

L’uso dei semafori può essere pericoloso

- I programmi non sono tutti semplici come quello che abbiamo visto.
- In un programma complesso può capitare di dimenticare di rilasciare un lock (o di rilasciarlo più di una volta) in qualche circostanza.
- Questo porta facilmente a race conditions, deadlock, o altre situazioni scorrette.

Intrinsic lock (alias monitor lock)

- Associata **ad ogni istanza** della classe Object c'è un intrinsic lock (alias monitor lock).
- Poiché ogni oggetto in Java estende la classe Object, ogni oggetto ha un suo lock.

- Every object has an intrinsic lock associated with it.
- A thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them.
- A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

Intrinsic lock (alias monitor lock)

- Come si fa ad acquisire e rilasciare l'intrinsic lock?
- Essendo implicito, il suo uso è altrettanto implicito (non si sono istruzioni esplicite da usare).

- Da non confondere con **`java.util.concurrent.locks.Lock`**, un meccanismo di locking esplicito, che non trattiamo in questo corso.

Il modificatore `synchronized`

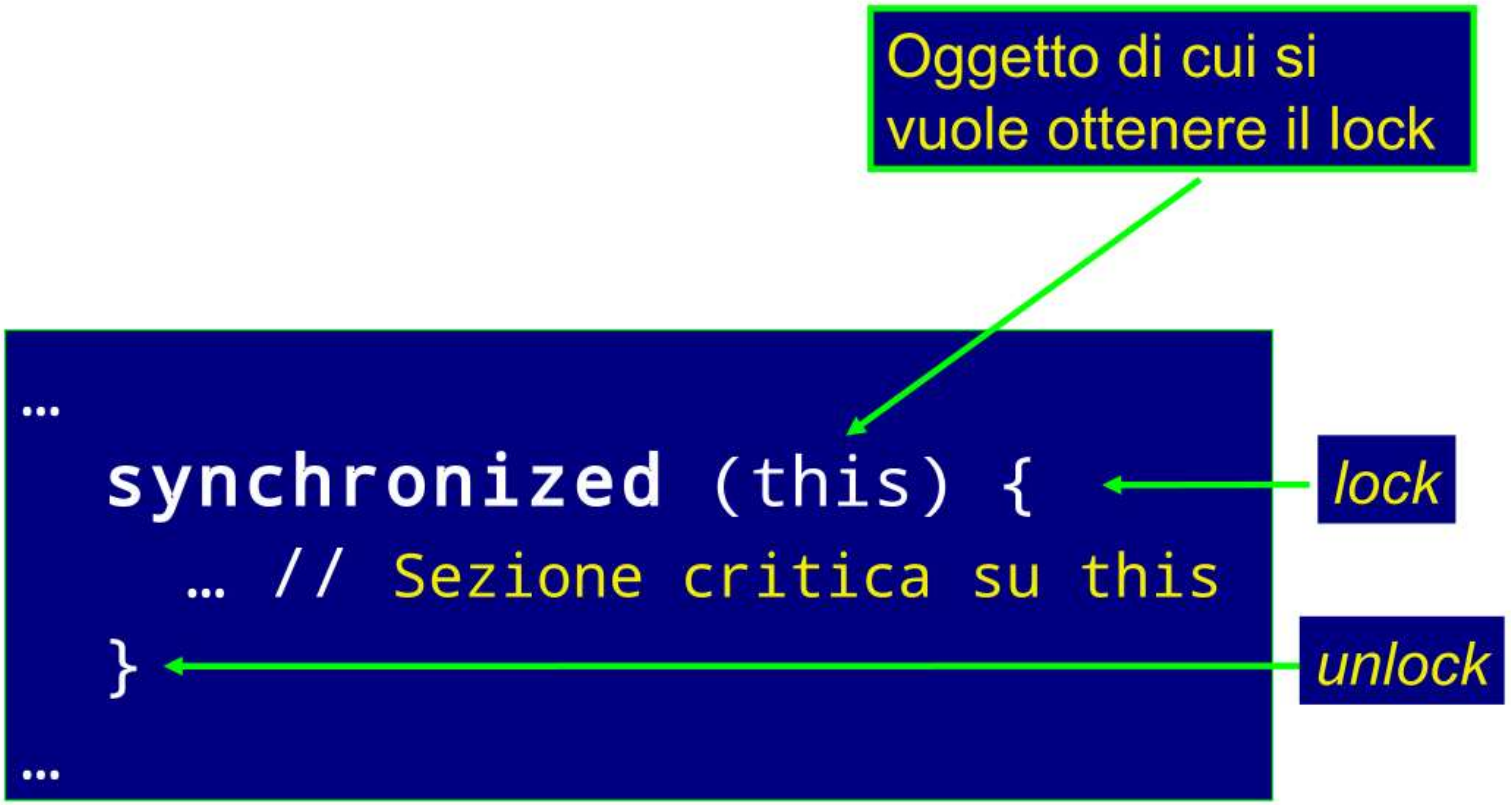
- La parola chiave **`synchronized`** applicata ad un metodo o ad una qualunque sezione di codice implica che:
  - Quando si entra nel metodo/sezione **`synchronized`** si cerca di acquisire il lock associato all'oggetto
  - Quando si esce dal metodo/sezione **`synchronized`** si rilascia il lock associato all'oggetto, quindi un eventuale thread in attesa può acquisire il lock ed entrare nel metodo/sezione **`synchronized`**
- NB: **`synchronized`** si riferisce ad un metodo o ad una sezione di codice, il lock si applica all'intero oggetto.
- Se il metodo **`m()`** è **`synchronized`**, la chiamata **`o.m()`** implica un lock sull'intero oggetto **`o`**
  - Risulteranno bloccati tutti gli accessi a **`o`** attraverso metodi **`synchronized`** (che cercano di acquisire il lock).
  - Si potrà accedere all'oggetto o mediante metodi non **`synchronized`** (che non cercano di acquisire il lcok).

Esempio che usa il modificatore **`synchronized`**

- La Race Condition dell'esempio precedente poteva essere facilmente risolta aggiungendo la parola chiave **`synchronized`** al metodo **`add`**,
  - Senza bisogno di usare **`Semaphore`**

```
public class Counter {
	private long count = 0;
	public synchronized void add (long value) {
		this.count += value;
	}
	public long getValue(){
		return count;
	}
}
```

Il modificatore `synchronized`



- Se un oggetto ha più di un blocco **`synchronized`**, uno solo di questi può essere attivo
  - Perché se un thread detiene il lock, tutti gli altri thread che vogliono accedere a blocchi o metodi **`synchronized`** devono aspettare fuori.

Esempio equivalente di uso del modificatore **`synchronized`**

```
public class Counter {
  private long count = 0;
  public void add (long value) {
    synchronized(this){
      this.count += value;
    }
  }
  public long getValue(){
    return count;
  }
}
```

In questo modo si controlla l'accesso alla sezione critica dentro al metodo **`add`**.

NB: il lock è comunque sull'intero oggetto!

Altro esempio di Race Condition

```
public class InsiemePosti {
  private int totPostiDisponibili=20;
  public boolean assegnaPosti(String cliente,
                              int numPostiRichiesti){
    if(numPostiRichiesti<=totPostiDisponibili) {
      totPostiDisponibili-=numPostiRichiesti;
      return true;
    } else { return false; }
  }
  public int getPostiDisponibili(){
    return totPostiDisponibili;
  }
}
```

Sezione critica: si legge **totPostiDisponibili** e poi si aggiorna.

- Se più thread eseguono **assegnaPosti()** concorrentemente, il numero di posti assegnato alla fine sarà maggiore di quello realmente disponibile! (Race condition)
- Non è detto che il problema si verifichi ad ogni esecuzione (non determinismo)

Completiamo l'esempio...

```
public class Richiedente extends Thread {
  private int numPosti;
  private InsiemePosti iPosti;
  public Richiedente(String nome, int n, InsiemePosti p){
    super(nome);
    this.numPosti=n;
    this.iPosti=p;
  }
  public void run(){
    System.out.println(getName()+" richiede "+numPosti);
    if(iPosti.assegnaPosti(getName(), numPosti)){
      System.out.println(getName()+" ottiene "+numPosti);
    } else {
      System.out.println(getName()+" NON ottiene "+
                         numPosti);
    }
  }
```

Completiamo l'esempio...

```
public static void main(String[] args)
                    throws InterruptedException {
 InsiemePosti iPosti= new InsiemePosti();
 Richiedente client1 = new Richiedente("cliente 1", 3,
                                       iPosti);
 Richiedente client2 = new Richiedente("cliente 2", 5,
                                       iPosti);
 Richiedente client3 = new Richiedente("cliente 3", 3,
                                       iPosti);
 Richiedente client4 = new Richiedente("cliente 4", 10,
                                       iPosti);
 client1.start(); client2.start();
 client3.start(); client4.start();
 client1.join(); client2.join();
 client3.join(); client4.join();
 System.out.println("Al termine restano disponibili "+
        iPosti.getPostiDisponibili() +" posti");
}
```

Un possibile output corretto

```
cliente 1 richiede 3
cliente 4 richiede 10
cliente 3 richiede 3
cliente 2 richiede 5
cliente 3 ottiene 3
cliente 4 ottiene 10
cliente 1 ottiene 3
cliente 2 NON ottiene 5
Al termine restano disponibili 4 posti
```

Un possibile output scorretto

```
cliente 1 richiede 3
cliente 4 richiede 10
cliente 3 richiede 3
cliente 2 richiede 5
cliente 1 ottiene 3
cliente 3 ottiene 3
cliente 2 ottiene 5
cliente 4 ottiene 10
Al termine restano disponibili -1 posti
```

Come si corregge l'esempio appena visto?

- Bisogna fare in modo che un thread possa ottenere il lock dell'oggetto **`InsiemePosti`**, delimitando la parte critica con **`synchronized`**.

```
synchronized public boolean assegnaPosti(String cliente,
                                         int numPostiRichiesti){
   if(numPostiRichiesti<=totPostiDisponibili) {
     totPostiDisponibili-=numPostiRichiesti;
     return true;
   } else { return false; }
 }
```

oppure ...

In questo caso si blocca l'oggetto solo per la durata della sezione critica.

```
public boolean assegnaPosti(String cliente,
                            int numPostiRichiesti){
   synchronized(this){
      if(numPostiRichiesti<=totPostiDisponibili) {
         totPostiDisponibili-=numPostiRichiesti;
         return true;
      }
   }
   return false;
}
```

Accesso sincronizzato

- Un metodo **`synchronized`** può essere eseguito da un solo thread alla volta.
- Non solo: mentre un thread sta eseguendo un metodo **`synchronized`**, nessun altro thread può eseguire alcun altro metodo **`synchronized`** dello stesso oggetto.
- I metodi **`synchronized`** hanno accesso **esclusivo** ai dati incapsulati nell'oggetto solo se a tali dati si accede esclusivamente con metodi **`synchronized`**
- I metodi non **`synchronized`** non sono esclusivi e quindi permettono **l'accesso concorrente** ai dati in qualunque momento

- Variabili locali
  - Poiché ogni thread ha il proprio stack, se più thread stanno eseguendo lo stesso metodo, ognuno avrà la propria copia delle variabili locali, senza pericolo di "interferenza"

Come funziona synchronized

- Synchronization is built around an internal entity known as the intrinsic lock or monitor lock.
  - The API specification often refers to this entity simply as a "monitor.“
- Every object has an intrinsic lock associated with it.
  - A thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them.
  - A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock.
  - As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.
- When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns (even if the return was caused by an uncaught exception.

Come funziona synchronized

- You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object.
- In this case, the thread acquires the intrinsic lock for the Class object associated with the class.
- Thus access to class's static fields is controlled by a lock that is distinct from the lock for any instance of the class.

`Synchronized` e variabili statiche

- Metodi e blocchi **`synchronized`** non assicurano l'accesso mutuamente esclusivo ai dati "statici"
  - I dati statici sono condivisi da tutti gli oggetti della stessa classe
- In Java ad ogni classe è associato un oggetto di classe Class
- Per accedere in modo sincronizzato ai dati statici si deve ottenere il lock su questo oggetto di tipo Class:
  - si può dichiarare un metodo statico come **`synchronized`**
  - si può dichiarare un blocco come **`synchronized`** sull'oggetto di tipo Class

- Attenzione
  - Il lock a livello classe non si ottiene quando ci si sincronizza su un oggetto di tale classe e viceversa.

`Synchronized` e variabili statiche

```
class StaticSharedVariable {
  // due modi per ottenere lock a livello di classe

  private static int shared;
  public int read() {
    synchronized(StaticSharedVariable.class) {
      return shared;
    }
  }

  public synchronized static void write(int i) {
    shared=i;
  }
}
```

Il parametro indica che si usa l'intrinsic lock della classe

Usa l'intrinsic lock della classe

Ereditarietà e `synchronized`

- La specifica **`synchronized`** non fa propriamente parte della segnatura di un metodo
- Quindi una classe derivata può ridefinire un metodo **`synchronized`** come non **`synchronized`** e viceversa

- Il fatto che **`synchronized`** non faccia parte della segnatura è molto comodo, perché ci consente di
  1. Definire classi adatte all'uso sequenziale senza preoccuparci dei problemi della concorrenza, e
  2. Poi modificare queste classi derivando sottoclassi che vengono rese adatte all'uso concorrente mediante **`synchronized`**.

Esempio

```
public class Counter {
      private long count = 0;
      public void add (long value) {
            this.count += value;
      }
      public long getValue(){
            return count;
      }
}
```

La classe **`Counter`** non garantisce l'accesso esclusivo a **`count`**

Esempio

```
public class ThreadSafeCounter extends Counter {
      public synchronized void add (long value) {
            this.count += value;
      }
}
```

La classe **ThreadSafeCounter** garantisce l'accesso esclusivo a **count**

Ereditarietà e `synchronized`

- Si può quindi ereditare da una classe “non sincronizzata” e ridefinire un metodo come **`synchronized`**, che richiama semplicemente l’implementazione della superclasse
- Questo assicura che gli accessi ai metodi nella sottoclasse avvengano in modo sincronizzato.

- NB: ovviamente non bisogna accedere a **`Counter`** direttamente. La protezione funziona solo se vi si accede sempre solo attraverso **`ThreadSafeCounter`**.