



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Programmazione parallela con i Java Thread – parte 2 Sincronizzazione di Thread



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



Finora abbiamo visto esempi di thread indipendenti.

In molti casi i thread **si devono sincronizzare,**
sia per coordinarsi per la risoluzione di un problema,
sia perché competono per una risorsa.

Consideriamo ad esempio il caso di più thread che usano la stessa stampante. Se un thread sta stampando un documento sulla stampante, gli altri thread che cercano di usare la stampante si devono fermare fino a quando il primo thread ha terminato.



Supponiamo di avere una classe **Stampante** con un metodo **stampa()** che stampa un array di stringhe (**Stampa**)

```
class Stampante
{
    public void stampa(String[] a)
    {
        for(int i = 0; i < a.length; i++)
        {
            try {Thread.sleep((long)(Math.random() * 100));}
            catch(InterruptedException e) {}
            System.out.println(a[i]);
        }
    }
}
```



e una classe **ThreadStampa** che realizza un thread che stampa un array *a* sulla stampante *st*.

```
class ThreadStampa extends Thread
{
    String[] a;
    Stampante st;
    public ThreadStampa(String[] arr, Stampante s)
    {
        a = arr; st = s;
    }

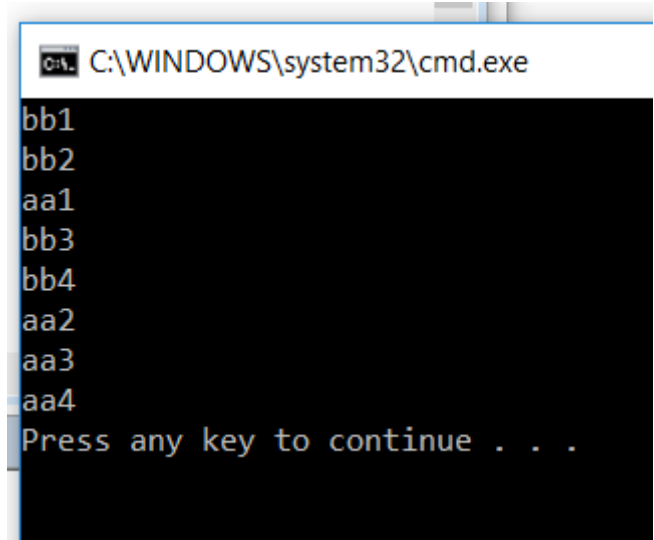
    public void run()
    {
        st.stampa(a);
    }
}
```



se si esegue questo main()

```
public static void main(String[] args)
{
    Stampante st = new Stampante();
    String[] a = {"aa1", "aa2", "aa3", "aa4"};
    String[] b = {"bb1", "bb2", "bb3", "bb4"};
    ThreadStampa t1 = new ThreadStampa(a,st);
    ThreadStampa t2 = new ThreadStampa(b,st);
    t1.start();
    t2.start();
}
```

le righe di stampa dei due array **a** e **b** vengono mescolate, perché i due thread eseguono contemporaneamente il metodo **stampa()** dell'oggetto **Stampante**.



```
C:\WINDOWS\system32\cmd.exe
bb1
bb2
aa1
bb3
bb4
aa2
aa3
aa4
Press any key to continue . . .
```



```
public class AccessoWrong {  
    public static void main(String[] args) {  
        C c = new C(); T t1 = new T(1, c); T t2 = new T(2, c);  
        t1.start(); t2.start();  
        try { t1.join();}  
            catch(InterruptedException e1) {System.out.println(e1.getMessage());}  
        try { t2.join(); }  
            catch(InterruptedException e1) {System.out.println(e1.getMessage());}  
        System.out.println("Main: c.i= " + c.i); }  
}
```

```
class C {  
    public int i=0;  
    public void m() {  
        for(int k = 0; k < 100000; k++) i++;  
        for(int k = 0; k < 100000; k++) i--;  
    }  
}
```

```
class T extends Thread {  
    private int num; private C c;  
    public T(int x, C y) { num = x; c = y; }  
    public void run() {  
        for (int i=0; i<10; i++) {  
            c.m();  
            System.out.println("Thread " + num + ": c.i= " + c.i);  
        }  
    }  
}
```

Output strumenti
Thread 2: c.i= -26978
Thread 1: c.i= 62714
Thread 2: c.i= 72795
Thread 1: c.i= -21658
Thread 2: c.i= 100669
Thread 2: c.i= 18008
Thread 1: c.i= 18008
Thread 2: c.i= 18008
Thread 2: c.i= 18008
Thread 1: c.i= 18008
Thread 2: c.i= 18008
Thread 1: c.i= 18008
Thread 1: c.i= 18008
Thread 2: c.i= 18008
Thread 1: c.i= 18008
Thread 2: c.i= 18008
Thread 2: c.i= 18008
Thread 1: c.i= 18008
Thread 1: c.i= 18008
Thread 1: c.i= 18008
Main: c.i= 18008
Procedura completata correttamente



Vediamo un altro caso di un oggetto condiviso.

```
class MiaClasse implements Runnable
{
    int counter = 0;
    public void run()
    {
        incr();
    }
    public void incr()
    {
        counter++;
    }
}
```

Se ci sono due thread che eseguono contemporaneamente il metodo run() dell'oggetto condiviso, questi potrebbero interferire l'uno con l'altro producendo effetti non desiderati.



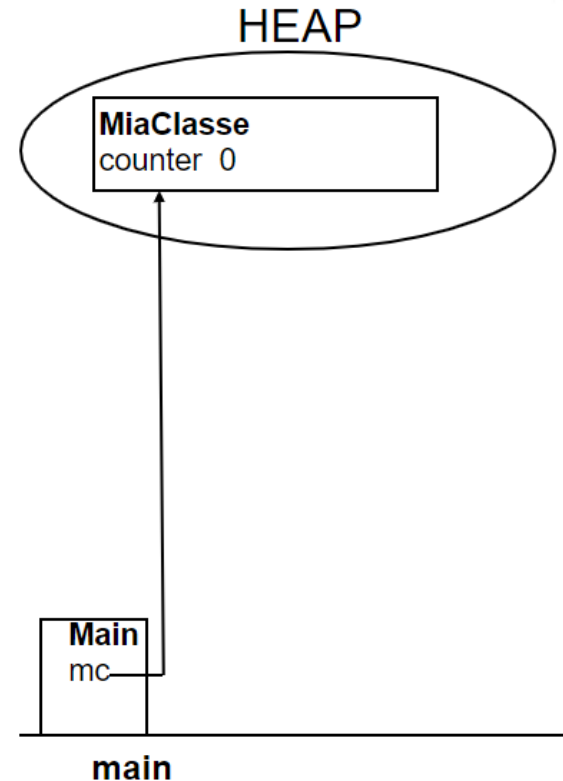
class MiaClasse implements Runnable

```
{  int counter = 0;
    public void run()
    {    incr();
    }
    public void incr()
    {    counter++;
    }
    public static void main(String[] args)
    {    MiaClasse mc = new MiaClasse();
        Thread t1 = new Thread(mc);
        Thread t2 = new Thread(mc);
        t1.start();
        t2.start();
    }
}
```

Running: main

In rosso l'istruzione appena eseguita

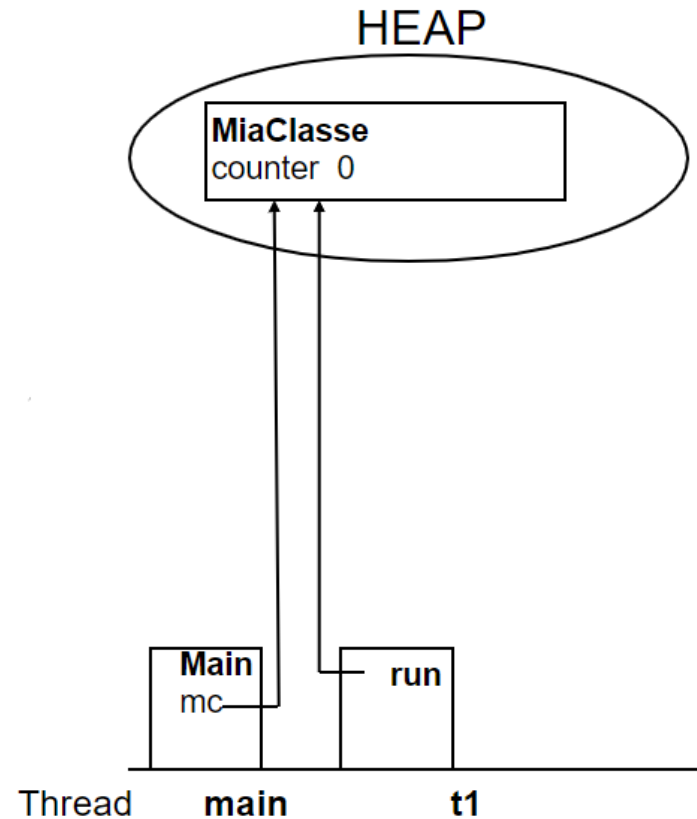
Thread





```
class MiaClasse implements Runnable {  
    int counter = 0;  
    public void run() {  
        incr();  
    }  
    public void incr() {  
        counter++;  
    }  
    public static void main(String[] args) {  
        MiaClasse mc = new MiaClasse();  
        Thread t1 = new Thread(mc);  
        Thread t2 = new Thread(mc);  
        t1.start();  
        t2.start();  
    }  
}
```

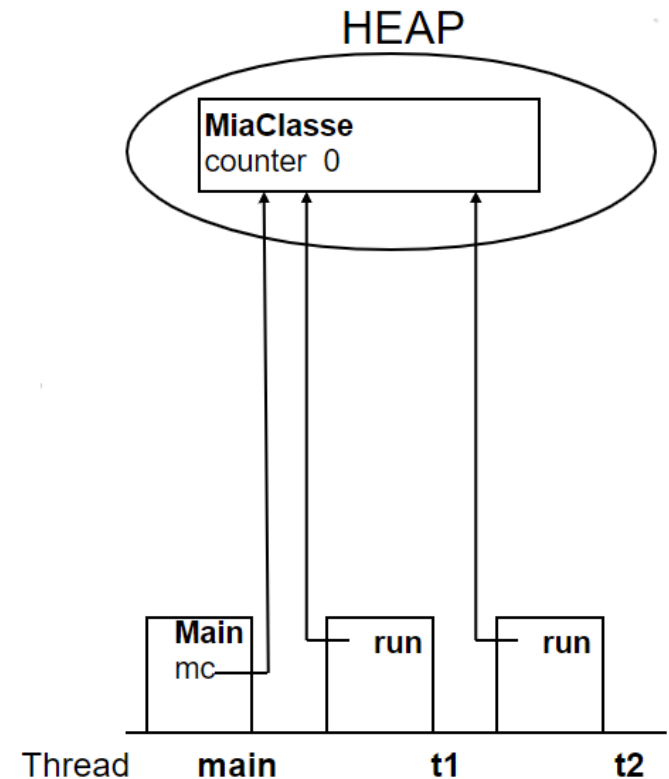
Running: main





```
class MiaClasse implements Runnable {  
    int counter = 0;  
    public void run() {  
        incr();  
    }  
    public void incr() {  
        counter++;  
    }  
    public static void main(String[] args)  
    {  
        MiaClasse mc = new MiaClasse();  
        Thread t1 = new Thread(mc);  
        Thread t2 = new Thread(mc);  
        t1.start();  
        t2.start();  
    }  
}
```

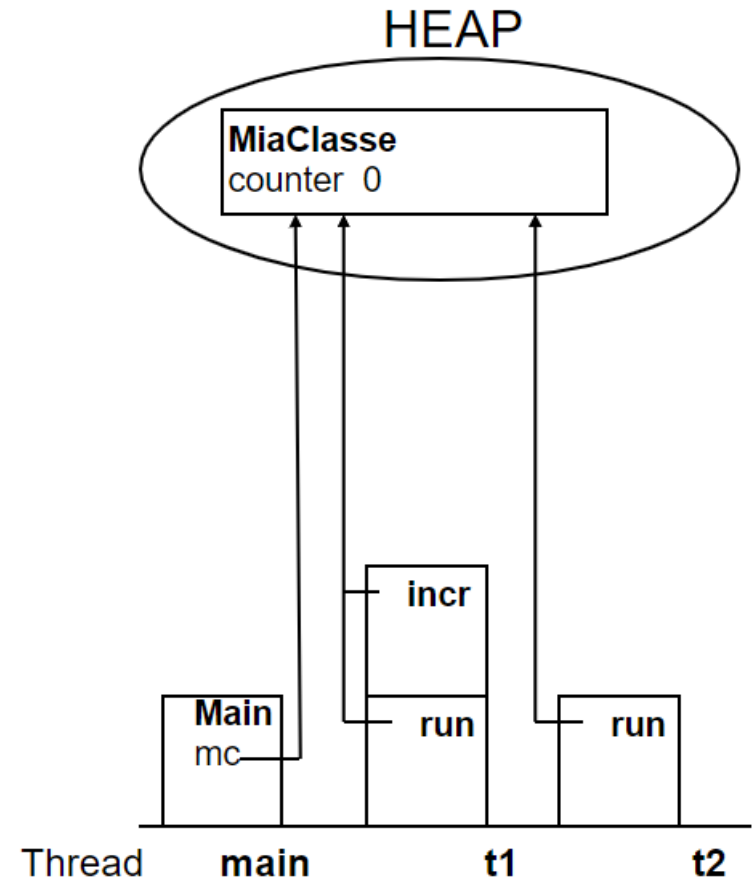
Running: main





```
class MiaClasse implements Runnable {  
    int counter = 0;  
    public void run() {  
        incr();  
    }  
    public void incr() {  
        counter++;  
    }  
    public static void main(String[] args) {  
        MiaClasse mc = new MiaClasse();  
        Thread t1 = new Thread(mc);  
        Thread t2 = new Thread(mc);  
        t1.start();  
        t2.start();  
    }  
}
```

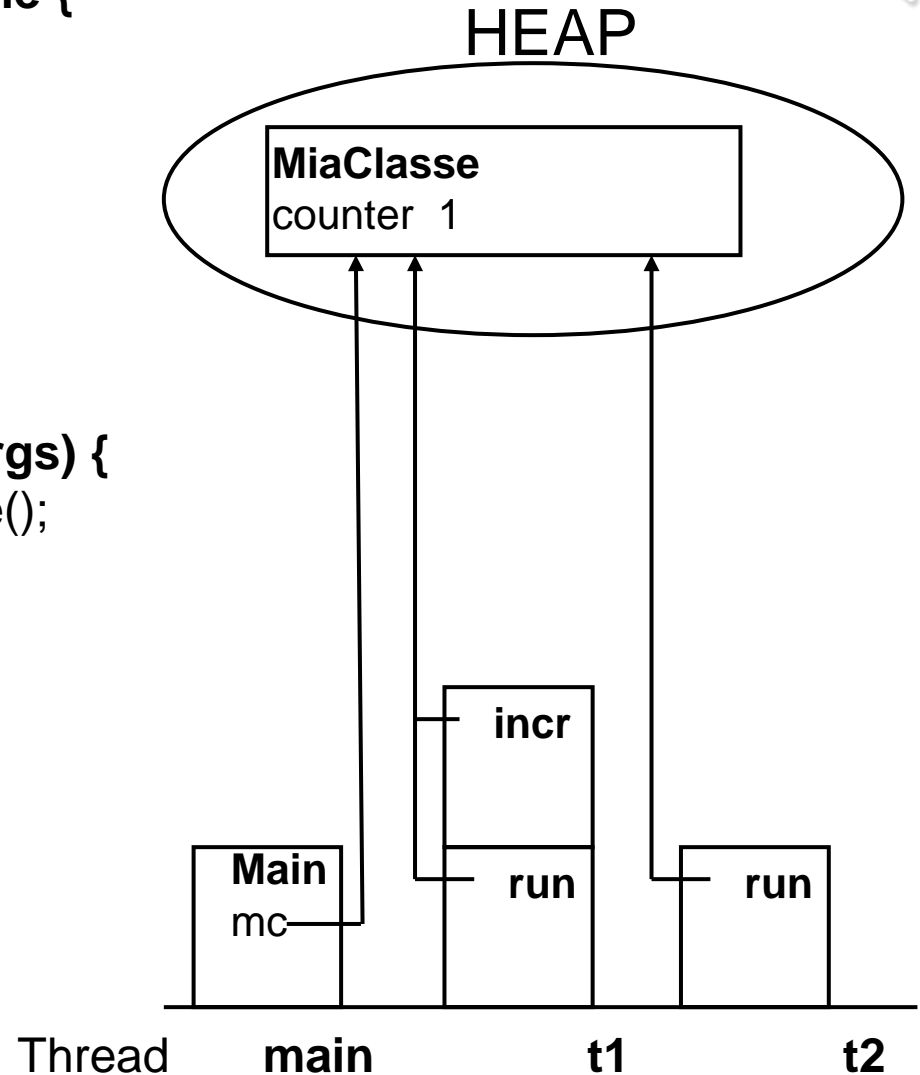
Running: t1





```
class MiaClasse implements Runnable {  
    int counter = 0;  
    public void run() {  
        incr();  
    }  
    public void incr() {  
        counter++;  
    }  
    public static void main(String[] args) {  
        MiaClasse mc = new MiaClasse();  
        Thread t1 = new Thread(mc);  
        Thread t2 = new Thread(mc);  
        t1.start();  
        t2.start();  
    }  
}
```

Running: t1

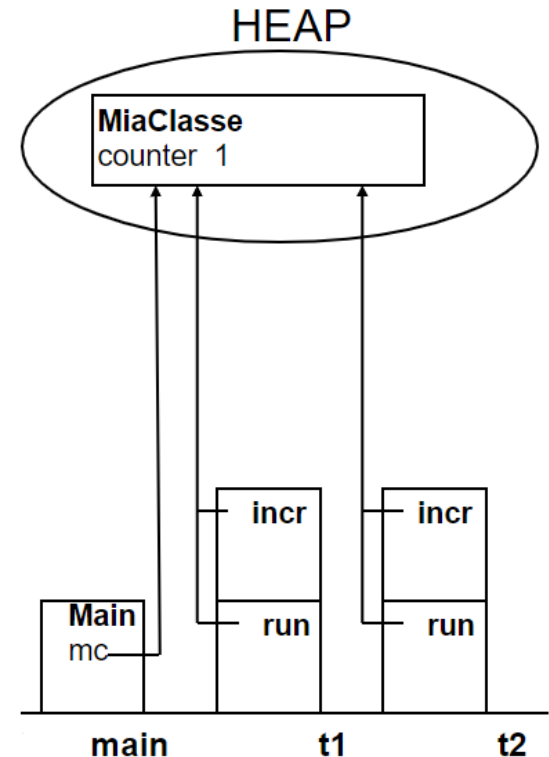




```
class MiaClasse implements Runnable {  
    int counter = 0;  
    public void run() {  
        incr();  
    }  
    public void incr() {  
        counter++;  
    }  
    public static void main(String[] args) {  
        MiaClasse mc = new MiaClasse();  
        Thread t1 = new Thread(mc);  
        Thread t2 = new Thread(mc);  
        t1.start();  
        t2.start();  
    }  
}
```

Running: t2

Thread





```
class MiaClasse implements Runnable {
```

```
    int counter = 0;
```

```
    public void run() {  
        incr();  
    }
```

```
    public void incr() {  
        counter++;  
    }
```

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

```
    {  
        MiaClasse mc = new MiaClasse();
```

```
        Thread t1 = new Thread(mc);
```

```
        Thread t2 = new Thread(mc);
```

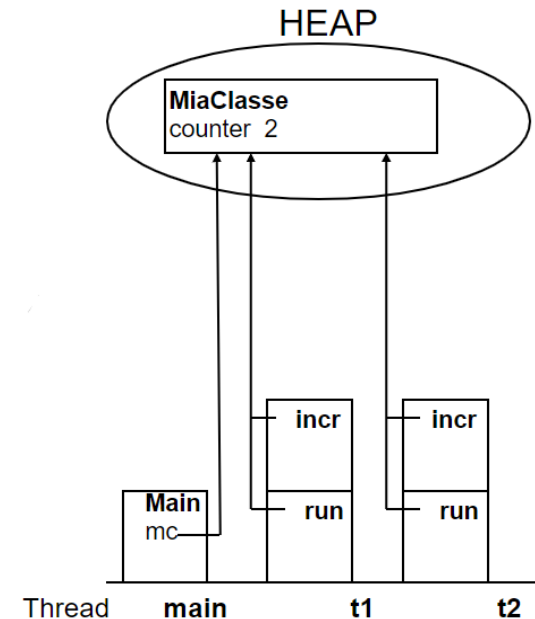
```
        t1.start();
```

```
        t2.start();
```

```
    }
```

```
}
```

Running: t2





Si potrebbe concludere che il programma termini sempre con **counter = 2.**

In realtà la conclusione si basa sull'assunzione che tutte le istruzioni, in particolare **counter++**, siano **atomiche**, ossia non interrompibili.

L'assunzione non è corretta. La macchina virtuale non esegue codice Java ma bytecode prodotto dal compilatore.

Ad esempio, l'istruzione **counter++** potrebbe essere tradotta in:

- carica **counter** in un registro
- incrementa il registro di 1
- sposta il risultato in **counter**

→ Se un thread si interrompe mentre esegue queste istruzioni, alla fine il **counter** potrebbe anche avere valore 1.

Esempio da JAVA2 (secondo volume) Abbiamo una classe **Bank** con un metodo **transfer()** per trasferire denaro da un conto ad un altro.



```
class Bank
{ public Bank(int n, int initialBalance)
  { accounts = new int[n];
    int i;
    for (i = 0; i < accounts.length; i++)
      accounts[i] = initialBalance;
  }

  public void transfer(int from, int to, int amount)
  { if (accounts[from] < amount) return;
    accounts[from] -= amount;
    accounts[to] += amount;
  }
  private int[] accounts;
}
```



```

class BankTestWrong {
    public static void main(String[] args) {
        Bank b = new Bank(10, 1000);
        for (int i=0; i < 10; i++)
            new TransferThread(b, 100).start();
    }
}

```

```

class BankAccount {
    private int balance;

    public void deposit(int amount) {
        int temp = balance;
        temp = temp + amount;
        balance = temp;
    }

    public boolean withdraw(int amount) {
        if (amount > balance) return false;
        int temp = balance;
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {}
        temp = temp - amount;
        balance = temp;
        return true;
    }

    public int getBalance() {
        return balance;
    }
}

```



```

Output strumenti
trasferimento da 1 a 0 di 43
trasferimento da 4 a 3 di 86
trasferimento da 6 a 4 di 27
trasferimento da 7 a 5 di 54
trasferimento da 7 a 1 di 88
trasferimento da 2 a 8 di 56
trasferimento da 7 a 5 di 30
trasferimento da 8 a 5 di 43
trasferimento da 2 a 1 di 79
trasferimento da 0 a 2 di 31
Somma totale di denaro in banca: 9877
Somma totale di denaro in banca: 9978
Somma totale di denaro in banca: 9908
Somma totale di denaro in banca: 9978
Somma totale di denaro in banca: 9978
Somma totale di denaro in banca: 9978
Somma totale di denaro in banca: 9978
Somma totale di denaro in banca: 9978
Somma totale di denaro in banca: 9978
Somma totale di denaro in banca: 9978
Procedura completata correttamente

```



Supponiamo di creare un oggetto **Bank** e di creare più thread che eseguono il metodo **transfer()** su questo oggetto, per eseguire dei trasferimenti fra i conti della banca.

I thread possono interferire causando una variazione sul totale delle somme presenti in tutti i conti (che dovrebbe rimanere invariato).

Ad esempio, supponiamo che il conto *i* abbia una somma di 5000, e che vengano eseguiti due thread che trasferiscono in questo conto le somme di 1000 e 2000, rispettivamente.

I due thread dovranno eseguire ciascuno l'istruzione

accounts[i] += amount;



Nella Java Virtual Machine l'istruzione **accounts[i] += amount;** viene realizzata con una **sequenza di istruzioni elementari**, che trasferiscono il valore di **accounts[i]** in un registro, lo incrementano e lo riportano in **accounts[i]**.

Se un thread viene interrotto dall'altro thread mentre esegue questa sequenza di istruzioni, il risultato può essere sbagliato.

thread1	registro1 = 5000	
thread1	registro1 = 6000	il thread 1 viene interrotto
thread2	registro2 = 5000	
thread2	registro2 = 7000	
thread2	accounts[i] = 7000	il thread 1 riprende
thread1	accounts[i] = 6000	



Il corpo del metodo **stampa()**, **incr()** o **transfer()** costituisce una sezione critica, che deve essere eseguita da un solo thread per volta senza interruzioni (mutua esclusione).

Come si può realizzare una sezione critica?

Con un semaforo: classe **Semaphore**.

Il costruttore **Semaphore(int n)** costruisce un semaforo con **n** permessi. Se $n = 1$ si ha un semaforo binario.

I metodi **acquire()** e **release()** acquisiscono o rilasciano un permesso. **Quando non ci sono più permessi un thread rimane in attesa nella coda del semaforo.**

Attn: mettere la **release() in blocco finally** per garantire il rilascio del semaforo e non bloccare gli altri thread.



Ad esempio la classe Stampante potrebbe essere modificata così:

```
class Stampante
{
    private Semaphore sem = new Semaphore(1);
    public void stampa(String[] a)
    {
        try
        {
            sem.acquire();
            for(int i = 0; i < a.length; i++)
            {
                Thread.sleep((long)(Math.random() * 100));
                System.out.println(a[i]);
            }
            //sem.release(); // non qui...
        }
        catch(InterruptedException e) {...}
        finally{sem.release();} // qui per garantire
        // rilascio del semaforo anche in caso di eccezioni
    }
}
```



Dato un oggetto *st* di tipo **Stampante**, il primo thread che esegue il metodo **stampa()** di *st* acquisisce il semaforo e blocca l'accesso a qualunque altro thread che cerchi di eseguire **stampa()** dello stesso oggetto. Gli altri thread che cercano di eseguire **stampa()** vengono messi in attesa.

Quando il primo thread termina di eseguire la sua sezione critica (il metodo **stampa()**) libera il semaforo, che potrà dare accesso ad un altro thread in attesa.

NOTARE: ogni istanza della classe Stampante ha il proprio semaforo (è una variabile di istanza) → programmi che usano stampanti diverse possono stampare in parallelo.



In Java **non è necessario usare esplicitamente un semaforo per realizzare una sezione critica** (la classe **Semaphore** non esisteva prima della versione 1.5).

Ogni istanza della classe **Object** (e quindi ogni oggetto) possiede un semaforo binario chiamato **lock**.

Se un metodo di qualunque oggetto viene dichiarato *synchronized*, automaticamente viene inserita una *acquire* del lock dell'oggetto all'inizio del metodo ed una *release* alla fine.

Non ci sono metodi per eseguire esplicitamente una *acquire* o una *release* del lock.



Lista di Lock: thread1, thread3, thread2, thread4



La classe **Stampante** può essere definita anche come segue:

```
class Stampante {  
    synchronized public void stampa(String[] a) {  
        try {  
            for(int i = 0; i < a.length; i++) {  
                Thread.sleep((long)(Math.random() * 100));  
                System.out.println(a[i]);  
            }  
        } catch (InterruptedException e) {}  
    }  
}
```

Questa formulazione è equivalente alla precedente che usava il semaforo, perchè il **lock** di un oggetto **Stampante** è usato esattamente come il semaforo binario *sem*.



Sincronizzazione

Per ogni classe in Java è possibile definire dei metodi **synchronized** (la classe può anche contenere metodi non sincronizzati)

Java associa **un lock ad ogni oggetto** della classe (+ un lock alla classe per sincronizzare i metodi statici).

Quando un thread chiama un metodo **synchronized**, acquisisce il **lock** dell'oggetto. Altri thread che tentino di accedere allo stesso oggetto chiamando metodi sincronizzati rimangono in coda sul lock fino a quando il thread precedente non lo rilascia, terminando l'esecuzione del metodo. A questo punto uno dei thread in coda può passare, bloccando di nuovo il lock.



Ogni oggetto ha il proprio lock

- **Due oggetti distinti hanno ciascuno il proprio lock.** I thread che eseguono metodi sincronizzati di un oggetto non interferiscono con i thread che eseguono metodi sincronizzati dell'altro.
- **Ogni oggetto ha un solo lock** e la sincronizzazione dei metodi synchronized avviene attraverso questo unico lock. **Se un thread sta eseguendo un metodo synchronized, nessun altro thread può eseguire alcun altro metodo synchronized dello stesso oggetto, fino a quando il primo thread non rilascia il lock.**



Il lock di un oggetto consente di realizzare *sezioni critiche*.

Se la sezione critica è costituita dal corpo di un metodo, è sufficiente dichiarare il metodo **synchronized**.

Tuttavia la sezione critica potrebbe essere solo una parte del corpo di un metodo. In questo caso si può sincronizzare un blocco:

```
synchronized(obj) {... sezione critica ...}
```

blocca il *lock* dell'oggetto **obj** per tutta l'esecuzione del blocco di istruzioni.

Es.: **synchronized(this) {counter++;}**



Si noti che le due formulazioni

```
synchronized public void stampa(String[] a)  
    {...blocco...}
```

e

```
public void stampa(String[] a) {  
    synchronized(this) {...blocco...}
```

sono equivalenti, perché nella seconda formulazione il blocco viene sincronizzato sul lock di `this`, ossia sul lock dell'oggetto stesso. Ma la seconda può avere granularità più fine rispetto all'intero corpo del metodo.

NB: una sezione critica demarca una sequenza di istruzioni da eseguire in modo atomico indipendentemente dal fatto che accedano o meno a variabili condivise.



AtomicInteger: contatori con incremento in mutua esclusione

Abbiamo visto che non è possibile considerare una singola istruzione come **counter++** come azione atomica.

Tuttavia Java fornisce dei meccanismi per rendere atomiche le operazioni su particolari variabili.
Ad esempio, la classe **AtomicInteger** implementa un valore **int** che può essere aggiornato atomicamente.

Il metodo

int incrementAndGet()

incrementa di 1 atomicamente il valore corrente e lo restituisce.



Sincronizzazione di thread lato client

- L'oggetto non viene protetto da accessi paralleli
- Tutti i client di un oggetto condiviso accedono all'oggetto attraverso blocchi sincronizzati sul lock dell'oggetto stesso
- Limitazioni: se un client non implementa correttamente gli accessi all'oggetto condiviso si ottiene un malfunzionamento
- Ma talvolta è necessario implementare la mutua esclusione in questo modo

Sincronizzazione di thread lato client – esempio



```
class Stampa extends Thread {  
    MiaClasse obj;  
    public Stampa(MiaClasse o) {obj = o;}  
    public void run() {  
        synchronized(obj) {  
            for (int i=0; i<7; i++) {  
                try {Thread.sleep(200);} catch (InterruptedException e)  
                    {System.out.println(e.getMessage());}  
                System.out.println(Thread.currentThread().getName() + ": " + obj);  
            }  
        }  
    }  
}  
  
class MiaClasse {  
    public String toString() {  
        return "stato dell'oggetto";  
    }  
}
```

```
C:\WINDOWS\system32\cmd.exe  
Thread-0: stato dell'oggetto  
Thread-0: stato dell'oggetto  
Thread-0: stato dell'oggetto  
Thread-0: stato dell'oggetto  
Thread-0: stato dell'oggetto  
Thread-0: stato dell'oggetto  
Thread-0: stato dell'oggetto  
Thread-1: stato dell'oggetto  
Thread-1: stato dell'oggetto  
Thread-1: stato dell'oggetto  
Thread-1: stato dell'oggetto  
Thread-1: stato dell'oggetto  
Thread-1: stato dell'oggetto  
Thread-1: stato dell'oggetto  
Premere un tasto per continuare . . .
```



Sincronizzazione di thread lato server

- L'oggetto protegge le variabili condivise e offre metodi `synchronized` per operare su di esse per cui si auto-protegge dagli accessi esterni.
- I client, invocando metodi `synchronized` sull'oggetto condiviso, automaticamente si sincronizzano nell'accesso all'oggetto stesso.
- Limitazioni: definire `synchronized` i metodi dell'oggetto potrebbe non essere sufficiente per sincronizzare le attività dei thread.



Sincronizzazione di thread lato server – esempio

```
class Stampa extends Thread {
```

```
    MiaClasse obj;
```

```
    public Stampa(MiaClasse o) {obj = o;}
```

```
    public void run() {
```

```
        for (int i=0; i<7; i++) {
```

```
            try {Thread.sleep(200);} catch (InterruptedException e)
```

```
                {System.out.println(e.getMessage());}
```

```
            System.out.println(Thread.currentThread().getName() + ": " + obj);
```

```
        }
```

```
    }}
```

```
class MiaClasse {
```

```
    public synchronized String toString() {
```

```
        return "stato dell'oggetto";
```

```
    }
```

```
}
```

Non basta sincronizzare lato client se si vuole
che le 7 stampe avvengano in sezione critica

```
C:\WINDOWS\system32\cmd.exe
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Thread-0: stato dell'oggetto
Thread-1: stato dell'oggetto
Premere un tasto per continuare . . .
```

Cooperazione fra thread



Spesso un thread non può eseguire un metodo sincronizzato, anche se ha ottenuto il possesso del lock, perché deve aspettare che si verifichi una condizione che non dipende da lui. Per esempio, deve aspettare che la risorsa condivisa entri in un certo stato prima di utilizzarla.

→ per non occupare inutilmente la CPU, **il thread deve rilasciare il lock.** Per fare questo si può mettere in attesa della condizione, **eseguendo il metodo `wait()`**, in modo che un altro thread possa entrare e realizzare la condizione.

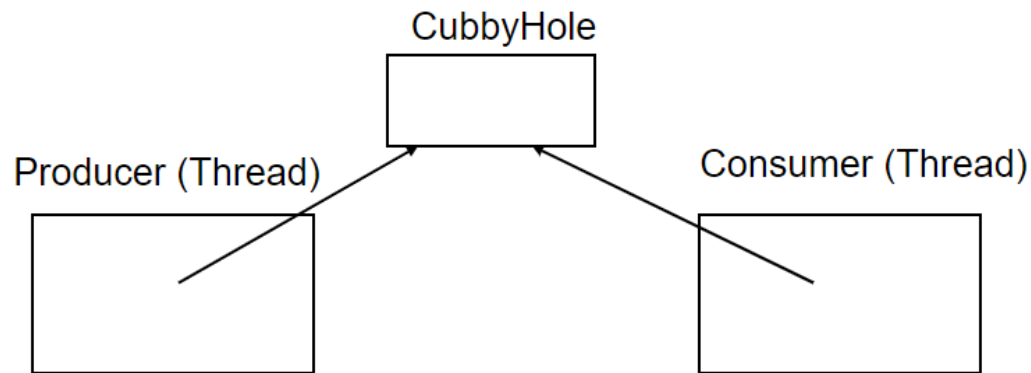
Quando un **thread realizza la condizione, avvisa, eseguendo `notify()` o `notifyAll()`**, i thread in wait → questi possono riprendere l'esecuzione.



Es. produttore-consumatore.

Come realizzare un buffer di un elemento (*cubbyhole*)

ProducerConsumerTest



Il produttore può inserire un elemento nel buffer quando questo è vuoto. Viceversa il consumatore può togliere un elemento dal buffer quando questo è pieno.



```
public class Producer extends Thread {  
    private CubbyHole cubbyhole;  
    private int number;  
  
    public Producer(CubbyHole c, int number) {  
        cubbyhole = c;  
        this.number = number;  
    }  
  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            cubbyhole.put(i);  
            System.out.println("Producer #" + this.number + " put: " + i);  
            try {  
                sleep((int)(Math.random() * 100));  
            } catch (InterruptedException e) { }  
        }  
    }  
}
```



```
public class Consumer extends Thread {
```

```
    private CubbyHole cubbyhole;
```

```
    private int number;
```

```
public Consumer(CubbyHole c, int number) {
```

```
    cubbyhole = c;
```

```
    this.number = number;
```

```
}
```

```
public void run() {
```

```
    int value = 0;
```

```
    for (int i = 0; i < 10; i++) {
```

```
        value = cubbyhole.get();
```

```
        System.out.println("Consumer #" + this.number + " got: " + value);
```

```
    }
```

```
}
```

```
}
```



```
public class ProducerConsumerTest {  
  
    public static void main(String[] args) {  
        CubbyHole c = new CubbyHole();  
        Producer p1 = new Producer(c, 1);  
        Consumer c1 = new Consumer(c, 1);  
  
        p1.start();  
        c1.start();  
    }  
}
```



Implementazione errata di CubbyHole

```
public class CubbyHole {  
    private int contents;  
  
    public int get() {  
        return contents;  
    }  
  
    public void put (int value) {  
        contents = value;  
    }  
}
```

- L'accesso alla risorsa condivisa non avviene in mutua esclusione
- Se il produttore è più lento del consumatore, il consumatore legge più volte lo stesso valore. Se il produttore è più veloce, il consumatore perde dei valori.



I thread **Producer** e **Consumer** si devono sincronizzare.

Dichiarare i metodi **get()** e **put()** **synchronized** è necessario per modificare il buffer in modo atomico. MA non è sufficiente, perché questi metodi possono essere eseguiti solo sotto certe condizioni (buffer pieno o vuoto rispettivamente). Questa condizione può essere descritta dal valore di una variabile booleana **available** (`available==true` significa che il buffer è pieno).

Un thread che vuole eseguire una `get()` la potrà eseguire solo se **available** è true, altrimenti dovrà mettersi in attesa che qualche altro thread esegua una `put()`.

Dopo che la `get()` è stata eseguita, **available** diventerà false. Viceversa per la `put()`.



Implementazione corretta:

```
public class CubbyHole {  
    private int contents;  
    private boolean available = false;  
  
    public synchronized int get() {  
        while (!available) {  
            try {  
                wait();  
            } catch (InterruptedException e) { ... }  
        }  
        available = false;  
        notifyAll();  
        return contents;  
    }  
    // continua...
```



```
public synchronized void put(int value) {  
    while (available) {  
        try {  
            wait();  
        } catch (InterruptedException e) { ... }  
    }  
    contents = value;  
    available = true;  
    notifyAll();  
}  
}  
  
// fine di CubbyHole
```

Perché la condizione di wait viene verificata all'interno di un **while**? Si potrebbe sostituire il **while** con un **if**?



```
public synchronized int get() {  
    while (!available) {  
        try {  
            wait();  
        } catch (InterruptedException e) { ... }  
    }  
    ....  
}
```

Un thread in wait che viene risvegliato dalla **notify non ha garanzia di riprendere subito l'esecuzione**. Quando il thread acquisisce il lock, la condizione di wait potrebbe essere nuovamente falsa, e il thread si deve rimettere in wait. È quindi indispensabile verificare la condizione di nuovo prima di continuare.



wait(), notify(), notifyAll()

wait(), notify() e notifyAll() sono metodi di **Object** → vengono ereditati da qualunque classe.

Se si tenta di invocarli da un metodo non sincronizzato, si ha un errore a runtime.

Per ogni oggetto ci sono due liste di thread:

- Lista del lock: contiene i thread in attesa di acquisire il lock dell'oggetto,
- Lista di wait: contiene i thread che sono in wait su quell'oggetto.

Ricordiamo che un thread è blocked se sta eseguendo una **sleep()** o una operazione di I/O, oppure si trova nella lista del lock o nella lista di wait di qualche oggetto.

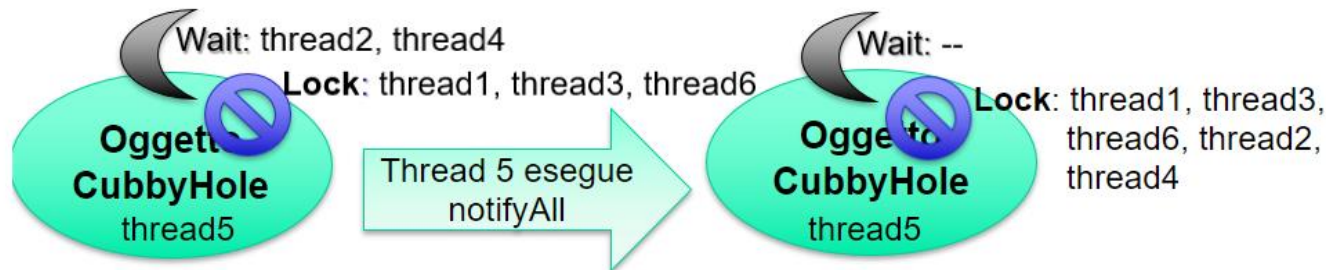


Da un metodo sincronizzato di un oggetto *ogg* si possono invocare i metodi:

wait() - rilascia il *lock* di *ogg* e mette il thread che lo ha eseguito nella *lista di wait* di *ogg*;

notify() - prende un thread a caso dalla *lista di wait* di *ogg* e lo sposta nella *lista del lock* di *ogg*;

notifyAll() - prende tutti i thread nella *lista di wait* di *ogg* e li sposta nella *lista del lock* di *ogg*.





Il thread che esegue **notify()** o **notifyAll()** non rilascia il lock.

I thread risvegliati dalla **notify()** o **notifyAll()** vengono inseriti nella coda del lock dell'oggetto e entrano in competizione per accedere all'oggetto, quando il lock verrà rilasciato.

Quando uno di questi thread acquisisce il lock, riparte dal punto in cui era andato in wait.

Non è garantito che un thread risvegliato dalla **notify()** passi prima di un thread che era già nella coda del lock.

Wait e notify – schemi di uso



Wait() e notify()/notifyAll() sono metodi di Object → ereditati da tutti gli oggetti. Per sincronizzare correttamente i thread su un oggetto obj condiviso, seguire questi due schemi (suggerimento):

```
synchronized(obj) {  
  while (!condition)  
    obj.wait();  
  ... istruz da eseguire  
  quando condition è  
  vera  
}
```

```
synchronized(obj) {  
  ... istruz che  
  modificano  
  condition  
  obj.notifyAll();  
}
```



Esempio: come realizzare un semaforo binario.

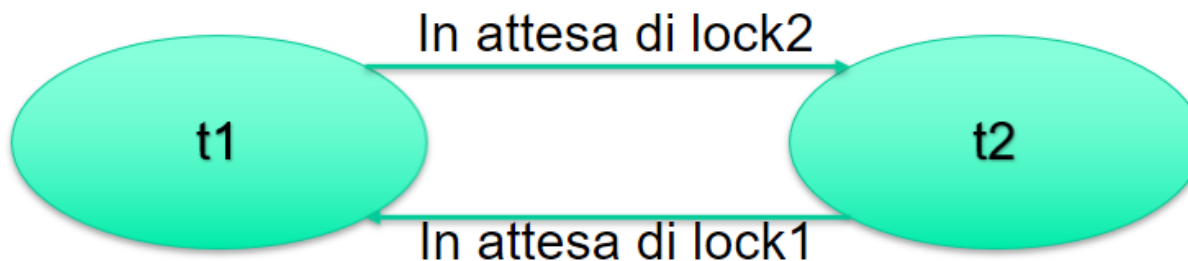
```
class Semaforo {  
    private boolean locked = false;  
  
    public synchronized void p() {  
        while (locked)  
            try {wait();}  
            catch (InterruptedException e){...}  
        locked = true;  
    }  
  
    public synchronized void v() {  
        if(locked) notify(); // notifyAll()  
        locked = false;  
    }  
}
```




Problemi che si possono verificare nella programmazione parallela

Deadlock: blocco fatale.

Per es., dati 2 soli thread, t1 e t2, si verifica quando t1 richiede una risorsa (qui, lock) detenuta da t2 che, a sua volta, richiede una risorsa detenuta da t1. Nessuno dei due può procedere con l'esecuzione e restano entrambi in attesa della risorsa.

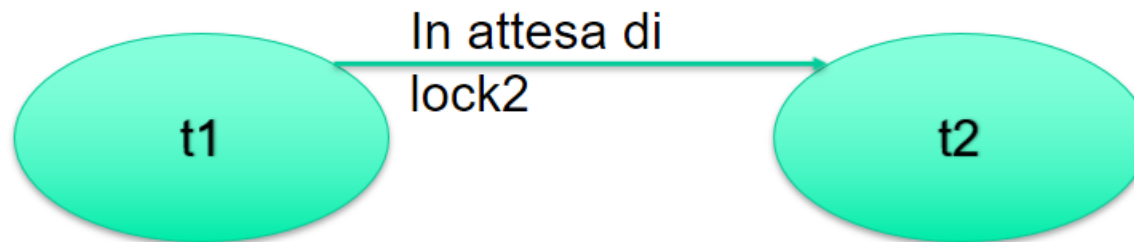


Problemi che si possono verificare nella programmazione parallela



Starvation: blocco di un thread che non riesce a continuare l'esecuzione → non termina.

Si verifica quando un thread richiede una **risorsa che non gli viene mai data** → non può procedere con l'esecuzione.

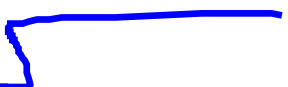


Nella programmazione parallela bisogna prevenire questi fenomeni con un'attenta progettazione della sincronizzazione tra thread



Passi di esecuzione di un programma concorrente (produttori-consumatori)

Un esempio di computazione che si può bloccare in una situazione di **deadlock**.





Consideriamo il problema dei produttori-consumatori e la sua implementazione vista in precedenza.

Supponiamo che il main crei tre thread **Producer**, **p1, p2 e p3**, e tre thread **Consumer**, **c1, c2 e c3**, che fanno tutti riferimento ad un unico oggetto **ch CubbyHole**. **Assumiamo anche che ogni thread esegua una sola operazione, put() o get(), su ch.**

Descriviamo i passi di una computazione di questo programma (fra le tante possibili).

Lo stato è costituito da tre elementi:

- lista dei thread in competizione per il lock di ch
- lista dei thread in wait
- stato dell'oggetto ch (valore della variabile booleana *available*)

Ad esempio lo stato iniziale sarà: [**p1, p2, p3, c1, c2, c3**], **<>**, **false**]



Questo programma può avere un comportamento diverso ad ogni esecuzione, perché non possiamo fare nessuna ipotesi su quale thread, fra quelli in attesa, acquisirà il lock quando questo viene rilasciato, né su quale thread verrà tolto dalla lista di wait se si esegue una notify().

Nel nostro esempio ipotizziamo che le liste siano gestite come code FIFO e che quindi venga sempre selezionato il primo elemento. In questo modo descriviamo una computazione fra tutte quelle possibili.

Lo stato iniziale è:

[<p1,p2,p3,c1,c2,c3>, <>, false]

Il thread p1 acquisisce il lock, esegue il metodo put() e termina rilasciando il lock.

Il nuovo stato sarà:

[<p2,p3,c1,c2,c3>, <>, true]



[<p2,p3,c1,c2,c3>, <>, true]

il thread p2 acquisisce il *lock*, inizia l'esecuzione della **put()**, verifica che la condizione non è soddisfatta, va in *wait* e rilascia il *lock*:

[<p3,c1,c2,c3>, <p2>, true]

Stessa cosa per p3:

[<c1,c2,c3>, <p2,p3>, true]

Il thread c1 acquisisce il *lock*, esegue il metodo **get()** → modifica la variabile *available*, invoca **notifyAll()** e rilascia il *lock*:

[<c2,c3, p2,p3 >, <>, false]

Il thread c2 acquisisce il *lock* e va subito in *wait*, e analogamente c3:

[<p2,p3 >, <c2,c3>, false]

Il thread p2 esegue la **put** e risveglia c2 e c3



[<p3,c2,c3>, <>, true]

Si procede analogamente ai passi precedenti:

[<c2,c3>, <p3>, true]

[<c3,p3>, <>, false]

[<p3>, <c3>, false]

[<c3>, <>, true]

[<>, <>, false]

Tutti i thread sono terminati e l'esecuzione termina correttamente.



Cosa succede se la **put()** e la **get()** fanno una **notify()** invece della **notifyAll()**? Si procede come prima fino a:

[<c1,c2,c3>, <p2,p3>, true]

c1 esegue la **get()** e alla fine esegue la **notify()** estraendo un solo thread dalla coda di wait:

[<c2,c3,p2>, <p3>, false]

Poi c2 e c3 acquisiscono il lock ma vanno subito in wait:

[<p2>, <p3,c2,c3>, false]

p2 esegue la **put()** e estrae un thread dalla lista di wait:

[<p3>, <c2,c3>, true]

p3 va in wait:

[<>, <c2,c3,p3>, true] e l'esecuzione si blocca (**DEADLOCK**).



Riprendiamo l'Esempio da JAVA2 (secondo volume):

Abbiamo una classe **Bank** con un metodo **transfer** per trasferire denaro da un conto ad un altro.

```
class Bank {  
    private int[] accounts;  
  
    public Bank(int n, int initialBalance) {  
        accounts = new int[n];  
        int i;  
        for (i = 0; i < accounts.length; i++)  
            accounts[i] = initialBalance;  
    }  
  
    public void transfer(int from, int to, int amount) {  
        if (accounts[from] < amount) return;  
        accounts[from] -= amount;  
        accounts[to] += amount;  
    }  
}
```



Creiamo un oggetto **Bank** con 10 conti inizializzati con 1000 euro ciascuno:

Bank bank = new Bank(10, 1000);

Creiamo più thread che eseguono il metodo **transfer()** su **bank**, per eseguire dei trasferimenti fra i conti della banca.

I thread possono interferire causando una variazione sul totale delle somme presenti in tutti i conti (il totale dovrebbe rimanere invariato).

```
C:\WINDOWS\system32\cmd.exe
trasferimento da 7 a 8 di 3
trasferimento da 8 a 3 di 73
trasferimento da 8 a 1 di 84
trasferimento da 4 a 0 di 61
trasferimento da 2 a 4 di 55
trasferimento da 1 a 8 di 61
trasferimento da 2 a 8 di 34
trasferimento da 5 a 7 di 66
trasferimento da 0 a 8 di 13
trasferimento da 8 a 0 di 96
Somma totale di denaro in banca: 10073
Somma totale di denaro in banca: 10073
Somma totale di denaro in banca: 10073
Somma totale di denaro in banca: 10073
Somma totale di denaro in banca: 10052
Somma totale di denaro in banca: 10107
Somma totale di denaro in banca: 10107
Somma totale di denaro in banca: 10107
Somma totale di denaro in banca: 9957
Somma totale di denaro in banca: 9800
Press any key to continue . . .
```



Sincronizzazione

Per risolvere il problema è sufficiente dichiarare il metodo **transfer() synchronized**. In questo modo solo un thread per volta lo potrà eseguire.

Quando un thread riesce ad acquisire il *lock* della banca e ad eseguire il metodo **transfer()**, potrà procedere fino al termine del metodo senza che nessun altro thread possa interferire.



Cooperazione fra thread

Il metodo **transfer()** non esegue il trasferimento se la somma presente sul conto di partenza è più bassa di quella che si vuole trasferire.

Modifichiamo `transfer()` in modo che, nel caso di somma **insufficiente, il thread che lo esegue rimanga in attesa** che la somma sul conto sia sufficiente per eseguire il trasferimento.

Il metodo può essere modificato come segue:



```
public synchronized void transfer(int from, int to, int amount) {  
    try {  
        while (accounts[from] < amount)  
            wait();  
        accounts[from] -= amount;  
        accounts[to] += amount;  
        ntransacts++;  
        notifyAll();  
    }  
    catch (InterruptedException e) {...}  
}
```

ATTENZIONE ALLA GRANULARITA' DELLA SINCRONIZZAZIONE



Se la sincronizzazione viene fatta sul lock della banca, si può eseguire solo un trasferimento alla volta.

Vediamo un altro esempio in cui vengono introdotte anche le classi **BankAccount**.

Nella prima versione ([BankTest](#)) non c'è sincronizzazione.

La classe **BankAccount** contiene i metodi **deposit()** e **withdraw()**. Il metodo **withdraw()** è booleano e restituisce *false* quando il saldo del conto non è sufficiente per fare il prelievo.

Per potere osservare gli effetti di una esecuzione concorrente di questi metodi, è stata introdotta una variabile *temp* ed è stata inserita una *sleep()* nel metodo **withdraw()**.

```
class BankAccount {  
    private int balance = 0;
```



```
    public synchronized void deposit(int amount) {  
        int temp = balance;  
        temp = temp + amount;  
        balance = temp;  
        notifyAll();  
    }
```

```
        // se la soglia di prelevamento è troppo alta withdraw si blocca  
    public synchronized void withdraw(int amount) {  
        while (amount > balance)  
            try { wait(); }  
                catch (InterruptedException e){}  
        int temp = balance;  
        try { Thread.sleep(1); }  
            catch(InterruptedException e) {}  
        temp = temp - amount;  
        balance = temp;  
    }
```

```
    public synchronized int getBalance() {  
        return balance;  
    }  
}
```



BankTestSynchr: dichiaro i due metodi **deposit()** e **withdraw()** di **BankAccount** come **synchronized**. Inoltre il metodo **withdraw()** si mette in **wait** se il suo saldo non è sufficiente per il prelievo.

In questo caso la sincronizzazione è fatta sul lock di un **BankAccount**, quindi sul singolo conto corrente e non sulla banca intera

- Due thread si devono sincronizzare solo se eseguono ambedue una operazione sullo stesso conto.
- Non c'è bisogno di sincronizzare il metodo **transfer()** di **Bank** perché sia il metodo di prelievamento sul singolo conto che quello di versamento sono sincronizzati.



Eseguendo questo programma si nota che spesso il programma non termina. Questo succede quando un thread rimane bloccato in wait nella **withdraw()** di un conto, in attesa che ci sia una cifra sufficiente, e nessun altro thread esegue un versamento sullo stesso conto.

Questo comportamento è più probabile più è alta la soglia sulle cifre da trasferire.

BankTestSynchr.java

```
class BankTestSynchr {
    public static void main(String[] args) {
        Bank b = new Bank(10, 1000);
        b.printTotal();
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%\n\n");
        int num = 10;
        TransferThread[] tt = new TransferThread[num];
        for (int i=0; i < num; i++) {
            // se metto una soglia di prelievamento altissima rischio la non terminazione
            // del programma perché le operazioni di prelievamento si bloccano
            //tt[i] = new TransferThread(b,80000);
            tt[i] = new TransferThread(b,80);
            tt[i].start();
        }

        // attendo fine di tutti i trasferimenti per stampare totale in banca
        for (int i=0; i < num; i++)
            try {
                tt[i].join();
            }
            catch (InterruptedException e) {System.out.println(e.getMessage());}
        System.out.println("\n\n%%%%%%%%%%%%%%%%%%%%%%%%");
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%");
        b.printTotal();
    }
}

class BankAccount {
    private int balance = 0;

    public synchronized void deposit(int amount) {
        int temp = balance;
        temp = temp + amount;
        balance = temp;
    }
}
```

Pipe - Comunicazione fra thread attraverso stream di I/O - I



Per far comunicare thread all'interno di una singola Java Virtual Machine si **possono usare pipe** – tubi. Le pipe sono canali di comunicazione su cui si possono scrivere e leggere dati.

È comodo usare le pipe quando si deve gestire la sincronizzazione tra thread produttore e thread consumatore su uno stream di byte:

- **Il thread produttore produce dati nello stream, e se produce troppo velocemente rispetto al consumatore, si blocca;**
- **Il thread consumatore legge dallo stream man mano che riceve dati, bloccandosi quando non ci sono dati.**

Con le pipe, non è necessario sincronizzare esplicitamente produttore e consumatore perché la sincronizzazione avviene in automatico man mano che scrivono e leggono sul/dal canale di comunicazione.

Comunicazione fra thread attraverso stream di I/O - II

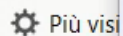


Esempio: con uso di PipedWriter e PipedReader in quanto il produttore genera caratteri UNICODE.

Per scrivere in uno stream di tipo pipe, usare un PipedWriter. Per leggere da pipe, usare un PipedReader impostato sullo stream del writer:

```
class PipeTest {  
    public static void main(String[] args) {  
        PipedWriter out = new PipedWriter();  
        PipedReader in = null;  
        try { in = new PipedReader(out); } catch(IOException e)  
            {System.out.println(e.getMessage());}  
        new Sender(out).start();  
        new Receiver(in).start();  
    }  
}
```

Il Sender e Receiver corrispondono ad un produttore e ad un consumatore rispettivamente. La pipe realizza un buffer.



Più vis



PipeTest.java x

```
        sleep(1000);
    } catch (IOException e) {}
    catch (InterruptedException e) {System.out.println(e.getMessage());}
}

class Receiver extends Thread {
    private Reader in;

    public Receiver (Reader r)
    {in = r;}

    public void run() {
        try {
            while (true) {
                sleep(1000);
                System.out.println("leggi " + (char)in.read());
            }
        } catch (IOException e) {System.out.println("fine input");
            //System.out.println(e.getMessage());
        }
        catch (InterruptedException e) {System.out.println(e.getMessage());}
    }
}

class PipeTest {
    public static void main(String[] args) {
        PipedWriter out = new PipedWriter();
        PipedReader in = null;
        try {
            in = new PipedReader(out);
        } catch (IOException e) {System.out.println(e.getMessage());}
        new Sender(out).start();
        new Receiver(in).start();
    }
}
```

Comunicazione fra thread attraverso stream di I/O - III



Vedere l'esempio **PipeTestBynaryData** per comunicazione tra 3 thread in pipeline, scambiando dati binari. Qui il produttore:

```
class Producer extends Thread {  
    private DataOutputStream out;  
    private Random rand = new Random();  
public Producer(OutputStream os) {out = new DataOutputStream(os);}  
    // os viene Istanziato con un PipedOutputStream  
public void run() {  
    while (true) {  
        try {  
            double num = rand.nextDouble();  
            out.writeDouble(num);  
            out.flush();  
            sleep(Math.abs(rand.nextInt() % 1000));  
        } catch (Exception e) {System.out.println("Error: " + e);}  
    }  
} // end Producer
```

Ringraziamenti



Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.