

Methods sleep

static void sleep(long millis) throws InterruptedException

static void sleep(long millis, int nanos) throws Intere.

PA1819_thread_con_sleep

```
run ▶ open in repl.it

Main.java History
1 class Main {
2     public static void main(String[] args) {
3         ThreadConSleep t1 = new ThreadConSleep(100);
4         ThreadConSleep t2 = new ThreadConSleep(500);
5         t1.start();
6         t2.start();
7     }
8 }
```

java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)

Nell'esempio precedente il metodo sleep() viene ereditato dalla superclasse Thread e invocato dal thread su se stesso.

Il metodo sleep può essere però usato in un punto qualunque del codice (non necessariamente in sottoclasse di Thread). Opera sul thread che chiama il metodo. In questo caso è necessario specificare il nome della classe a cui il metodo appartiene Thread.sleep()) (è un metodo statico).

PA1819_thread_che_conta

```
run ▶ open in repl.it

Main.java History
1 class Main {
2     public static void main(String[] args) {
3         Contatore c1 = new Contatore(100);
4         ThreadCheConta t1 = new ThreadCheConta(c1);
5         Contatore c2 = new Contatore(500);
6         ThreadCheConta t2 = new ThreadCheConta(c2);
7         t1.start();
8         t2.start();
9     }
10 }
```

java version "1.8.0_31"
Java(TM) SE Runtime Environment (build 1.8.0_31-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.31-b07, mixed mode)

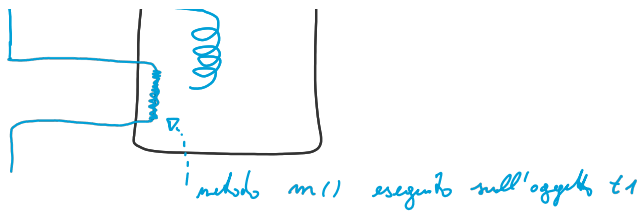
Oggetti e thread

```
public class MioThread extends Thread {
    :
    public void run() {
        for (int i = 0; i < 100; i++)
            System.out.println("Stampa " + i);
    }
    public void m() {
        =
    }
}
```

Supponiamo che il tutto sia fatto così:

```
public static void main(String[] args) {
    MioThread t1 = new MioThread();
    t1.start();
    t1.m();
    :
}
```





Mutua esclusione

Supponiamo di avere le seguenti classi:

```
public class ContoBancario {  
    long bilancio;  
    long numOperazioni;  
  
    void deposita(){  
        bilancio += 10;  
        numOperazioni++;  
        assert numOperazioni*10==bilancio : "numOperazioni=" + numOperazioni + " bilancio=" + bilancio;  
    }  
}
```

```
public class Depositatore extends Thread {
```

```
    ContoBancario c;
```

```
    Depositatore(ContoBancario c) {  
        this.c = c;  
    }
```

```
    public void run(){  
        while(true) {  
            c.deposita();  
        }  
    }
```

```
    public static void main(String[] args) {  
        ContoBancario conto = new ContoBancario();  
        Depositatore d1 = new Depositatore(conto);  
        Depositatore d2 = new Depositatore(conto);  
        Depositatore d3 = new Depositatore(conto);  
        d1.start();  
        d2.start();  
        d3.start();  
    }  
}
```

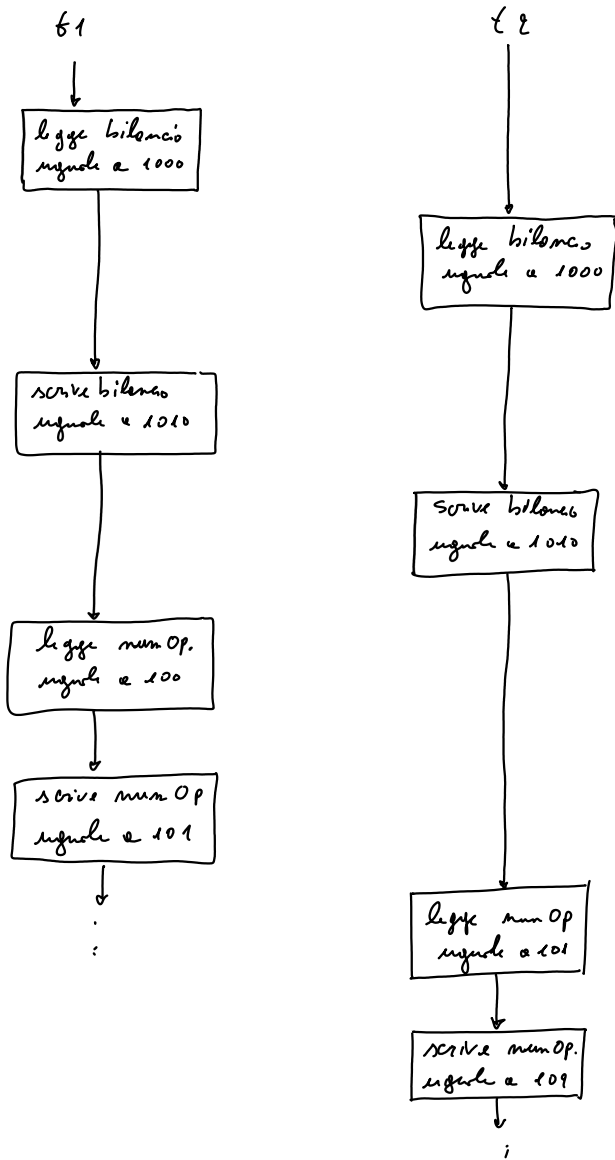
Se lanciamo l'applicazione con le asserzioni abilitate (java -ea ...) dopo un po' viene fuori il seguente errore:

```
java -ea Depositatore  
Exception in thread "Thread-1" Exception in thread "Thread-2" Exception in thread "Thread-0"  
java.lang.AssertionError: numOperazioni=1650 bilancio=16490  
    at ContoBancario.deposita(ContoBancario.java:8)  
    at Depositatore.run(Depositatore.java:11)  
java.lang.AssertionError: numOperazioni=1649 bilancio=16480  
    at ContoBancario.deposita(ContoBancario.java:8)  
    at Depositatore.run(Depositatore.java:11)  
java.lang.AssertionError: numOperazioni=1649 bilancio=16480  
    at ContoBancario.deposita(ContoBancario.java:8)  
    at Depositatore.run(Depositatore.java:11)
```

la struttura dati è finita in uno stato inconsistente

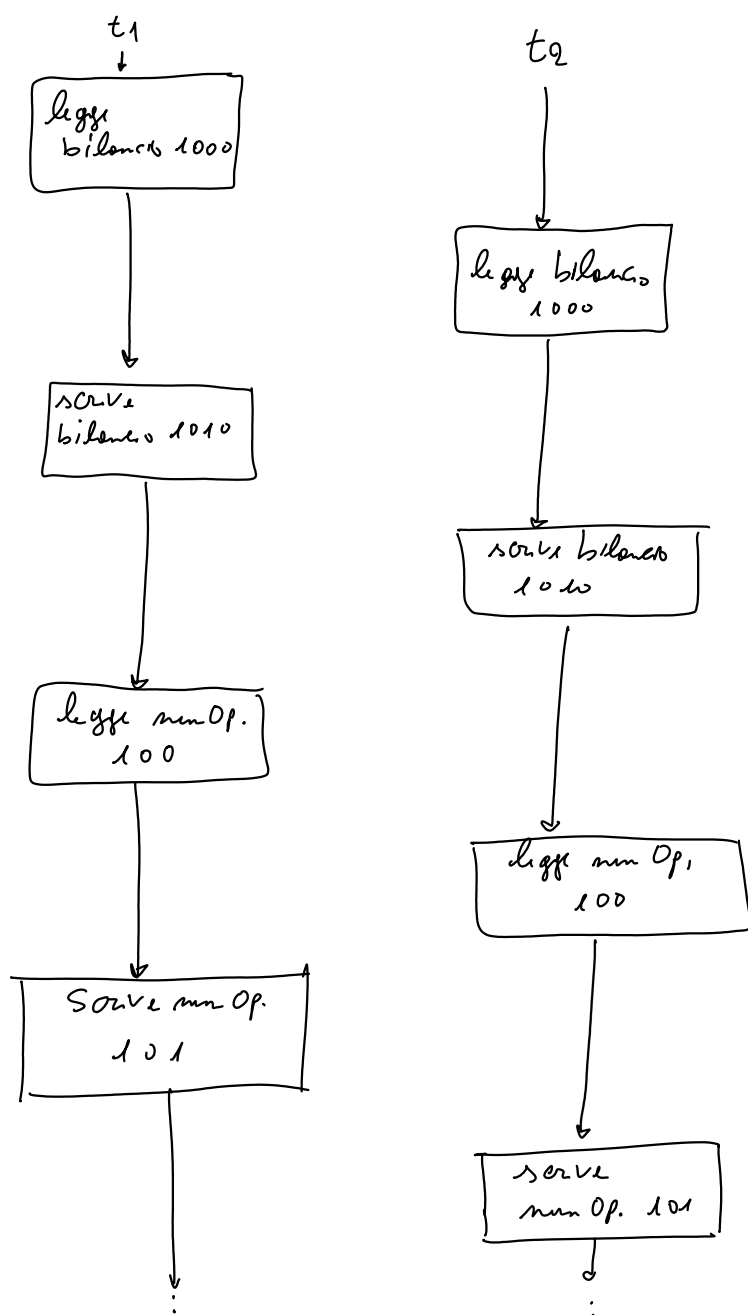
(ce ne siamo accorti tramite asserzione)

Supponiamo che il bilancio sia 1000 e che il n° di operazioni sia 100



↑
(Problema di corse critiche (il risultato dipende da come si "incontrano" le esecuzioni dei thread))

• sono altri mt che - a



↑ questo tipo di errore non viene rilevato
 dal vincolo $\text{bilancio} == \text{numOperazioni} * 10$
 (in quanto è un vincolo "locale" al conto)
 Ci siamo persi 10 di bilancio e 1 operazione
 (il vincolo è relativo a thread + conto)

Per risolvere il problema è necessario eseguire
 le sezioni di codice che manipolano lo stato dell'oggetto
 in mutua esclusione.

Protocollo:

- per poter eseguire operazioni su un oggetto, un thread deve prima acquisire il "lock" su tale oggetto.
- non è possibile acquisire il lock su un oggetto se è già stato acquisito da un altro thread
- un thread rilascia il lock una volta che ha terminato di lavorare sull'oggetto

In Java ogni oggetto ha un lock

Per acquisire il lock si usa la parola chiave `synchronized`

```
public class Conto {  
    private String intestatario;  
    private long bilancio;  
    private long numVers;  
    private long numPrelev;  
  
    public synchronized void versa() {  
        bilancio += 10;  
        numVers++;  
    }  
  
    public synchronized void preleva() {  
        bilancio -= 10;  
        numPrelev++;  
    }  
  
    public String getIntestatario() {  
        return intestatario;  
    }  
}
```

Se un thread esegue `c.preleva()` acquisisce il lock su `c` e lo rilascia quando esce.

Non è possibile eseguire contemporaneamente (2 thread)

`versa - versa`

`versa - preleva`

`preleva - preleva`

è possibile eseguire contemporaneamente `versa` e `getIntestatario` in quanto il secondo non è `synchronized` (lo stesso per `preleva` e `getIntestatario`).

Se un thread `t1` lavora su un conto `c1`
e un thread `t2` lavora su un conto `c2`
possono eseguire contemporaneamente `c1.preleva()` e `c2.preleva()` (o `versa`). Gli oggetti sono diversi, lock diversi.

I lock Java sono rientranti

```
public class X {  
  
    void synchronized m() {  
          
    }  
}
```

```

    ...
}
void synchronized m() {
    ...
}

```

Si accorge che il thread ha già il lock sull'oggetto e non si blocca.

• Il lock viene rilasciato sia nel caso di esecuzione normale, sia in caso di eccezione.

• Blocchi synchronized

```

synchronized (oggetto) {
    ...
}

```

Ridurre la parte da eseguire in mutua esclusione relativamente a un metodo

```

class Y {
    ...
    void m() {
        ...
        synchronized (this) {
            ...
        }
    }
}

```

Queste parti del corpo del metodo non sono eseguite in mutua esclusione

Incremento il grado di parallelismo del codice

metodo su ...

```
class A {
```

```
void synchronized m1() {
```

```
    =
```

```
}
```

```
void synchronized m2() {
```

```
    =
```

```
}
```

```
}
```

```
class A {
```

```
void m1() {
```

```
    synchronized (this) {
```

```
        =
```

```
}
```

```
}
```

```
void m2() {
```

```
    synchronized (this) {
```

```
        =
```

```
}
```

```
}
```

```
}
```

3 blocchi synchronized sono utili per accedere in modo mutuamente esclusivo su un array

```
void f(int[] v) {
```

```
    synchronized (v) {
```

```
        =
```

```
}
```

```
=
```

```
}
```

Utili per implementare mutua esclusione tra insiemi di metodi

```
class Separati {
```

```
    private int v1;
```

```
    private int v2;
```

```
    protected final Object l1 = new Object();
```

```
    protected final Object l2 = new Object();
```

```
    public int getV1() {
```

```
        synchronized (l1) {
```

```
            return v1;
```

```
        }
```

```
    }
```

```
    public void setV1(int v) {
```

```
        synchronized (l1) {
```

```
            v1 = v;
```

```

    }
    public int getV2() {
        synchronized (l2) {
            return v2;
        }
    }
    public void setV2(int v) {
        synchronized (l2) {
            v2 = v;
        }
    }
    public void K() {
        synchronized (l1) {
            synchronized (l2) {
                v1 = v2;
            }
        }
    }
}

```

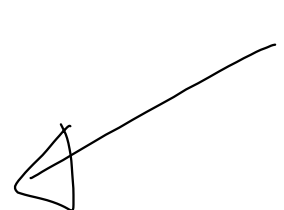
Le classi Java definiscono le proprie politiche di mutua esclusione / sincronizzazione.

```

class A {
    void m() {
        //
    }
}

```

NO



```

A a = new A();
synchronized (a) {
    a.m();
}

```

fonti di
errori:
mi devo ricordare
di farlo ogni volta
che accedo all'
oggetto a

Miglior

```

class A {
    void synchronized m() {

```

```

    }
}

```

Metodi statici sincronizzati

Se un metodo statico è `synchronized` viene acquisito il lock sulla classe (non ci sono istanze della classe, non c'è oggetto implicito)

```

class X {
    public static synchronized void s1() {
        // ...
    }
    public static synchronized void s2() {
        // ...
    }
    public synchronized void m1() {
        // ...
    }
}

```

Più mutua esclusione tra `s1-s1`, `s1-s2`, `s2-s2`
(se eseguiti da più thread)

ma non tra `s1-m1` e `s2-m1`

(`s1` e `s2` richiedono il lock sulla classe `X`, mentre `m1` lo richiede sull'oggetto su cui il metodo è invocato)

```

static synchronized void s1() {
    // ...
}

```

```

}

```

è come scrivere

```

static void s1() {
    synchronized (X.class) {
        // ...
    }
}

```



```

public class Conto {
    private int numero;
    private static int prossimo;

    public Conto() {
        numero = prossimo++;
    }
    :
}

```

Se più thread possono fare new Conto() possono venire fuori race condition.

Soluzione:

```

public class Conto {
    private int numero;
    private static int prossimo;

    public Conto() {
        numero = getProssimo();
    }

    private static synchronized int getProssimo() {
        return prossimo++;
    }
    :
}

```