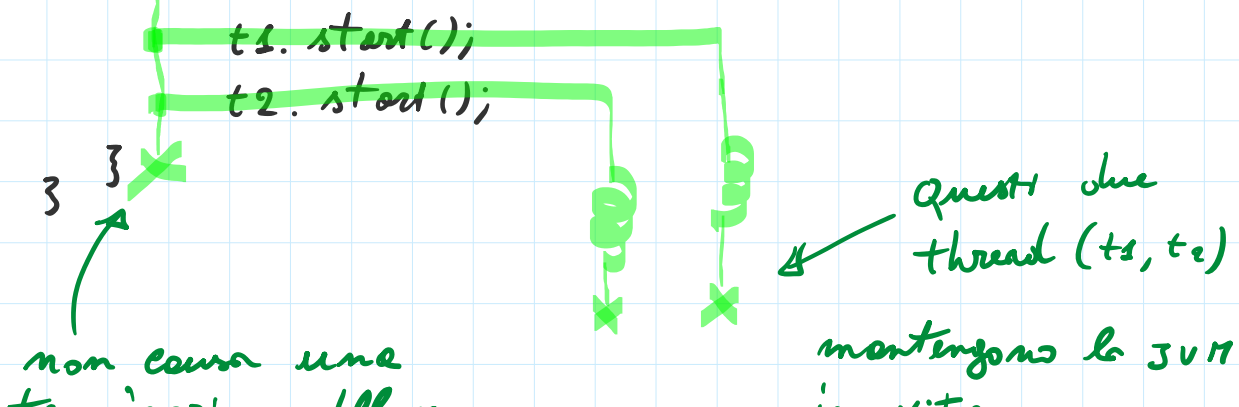


## Thread

Se un thread termina la propria esecuzione non può essere fatto ripartire chiamando il metodo `start()` una seconda volta.  
 È necessario creare un altro oggetto `Thread` (con `new`).

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

```
public class Prova {
    public static void main(String[] args) {
        MioThread t1 = new MioThread();
        MioThread t2 = new MioThread();
```



! non causa una  
terminazione della JVM

mantengono la JVM  
in vita  
quando t1 e t2 arrivano  
in fondo al loro  
metodo run()  
la JVM "esce"  
(termina la propria  
esecuzione)

Tutti i Thread vengono terminati  
quando viene chiamato il metodo

```
System.exit(-);
```

la JVM "esce" immediatamente

I Thread hanno un nome che può  
essere impostato con il costruttore

```
Thread(String name)
```

il nome è utile solo per debugging

Il nome può essere recuperato con

```
String getName()
```

Se non diamo un nome al thread  
allora avranno un nome predefinito del  
tipo Thread 1, Thread 2, Thread 3, ...

I thread hanno un livello di priorità

Thread.MIN\_PRIORITY (1)

Thread.MAX\_PRIORITY (10)

c'è un livello intermedio: Thread.NORMAL\_PRIORITY

Il livello predefinito del main thread è NORMAL\_PRIORITY

Quando viene creato un nuovo thread il suo livello di priorità viene "ereditato" dal padre

C'è un metodo che consente di impostare la priorità di un thread (setPriority)

La politica di scheduling è fixed-priority

Lo scheduler sceglie tra i thread pronti quello con la priorità più alta

C'è preemption: se è in esecuzione un thread con un certo livello di priorità e diventa pronto un thread con priorità più alta il processore viene revocato (a quello attualmente in esecuzione) e assegnato a quello a più alta priorità.

Se sono pronti più thread con lo stesso livello ne viene scelto uno a caso

## Thread: flussi di esecuzione e oggetti

```
class A {  
    void m() {  
        for (int i=0; i<100; i++)  
            System.out.println("ABC");  
    }  
}  
  
class ProvaThread extends Thread {  
    public void run() {  
        A a = new A();  
        a.m();  
    }  
}
```

```
ProvaThread p = new ProvaThread();  
p.start();
```

Altro esempio, più complicato

```
class AltroThread extends Thread {  
    int x;  
    void incrementa() {  
        x++;  
    }  
}
```

viene eseguito  
dal main thread

```

void incrementa() {
    x++;
}

```

del main thread

```

public void run() {
    for (int i=0; i<100; i++)
        System.out.println("xyz");
}
}

```

nel main :

```

AltroThread t1 = new AltroThread();
t1.start();
t1.incrementa();

```

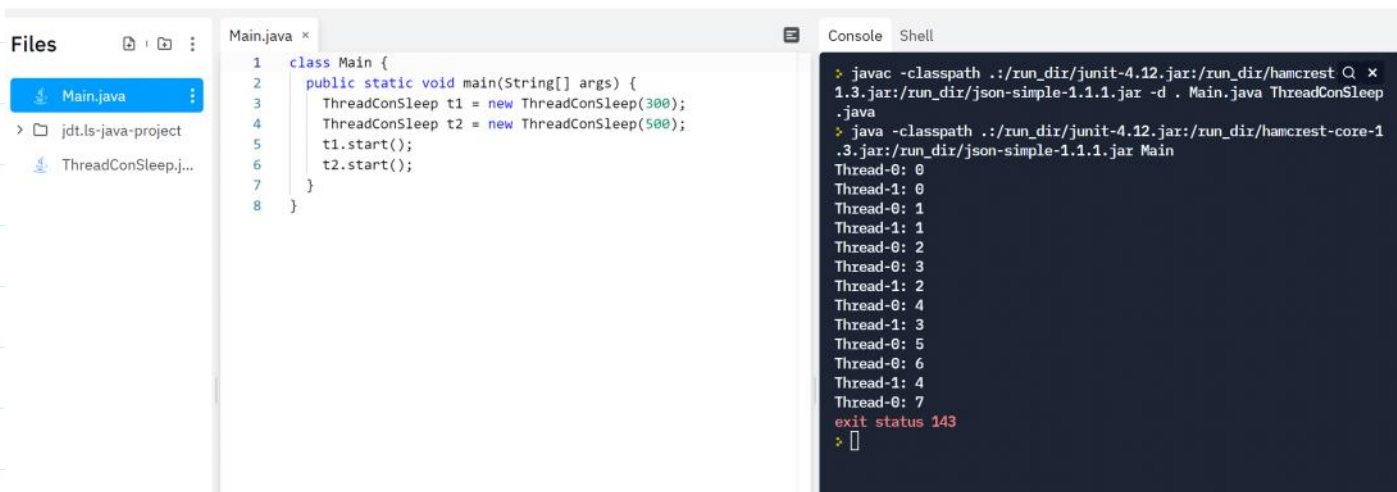
## Metodo sleep (class Thread)

static void sleep(long millis) throws InterruptedException

static void sleep(long millis, int menos) throws  
InterruptedException

Esempio

[PA2021 thread con sleep](#)



The screenshot shows an IDE with a project named 'jdt.ls-java-project'. The 'Main.java' file is open, showing a class 'Main' with a 'main' method. The method creates two 'ThreadConSleep' objects, 't1' (300ms) and 't2' (500ms), and starts them. The console output shows the execution of the program, displaying the thread names and the status of the threads. The output is as follows:

```
> javac -classpath ./run_dir/junit-4.12.jar:./run_dir/hamcrest-1.3.jar:./run_dir/json-simple-1.1.1.jar -d . Main.java ThreadConSleep.java
> java -classpath ./run_dir/junit-4.12.jar:./run_dir/hamcrest-core-1.3.jar:./run_dir/json-simple-1.1.1.jar Main
Thread-0: 0
Thread-1: 0
Thread-0: 1
Thread-1: 1
Thread-0: 2
Thread-0: 3
Thread-1: 2
Thread-0: 4
Thread-1: 3
Thread-0: 5
Thread-0: 6
Thread-1: 4
Thread-0: 7
exit status 143
> 
```

```
public class ThreadConSleep extends Thread {
```

```
    private String nome;
    private long quanto;
```

```
    public ThreadConSleep(long ms){
        quanto = ms;
        nome = getName();
    }
```

```
    public void run(){
        for(int i=0; i<100; i++) {
            try {
                System.out.println(nome + ": " + i);
                sleep(quanto);
            } catch (InterruptedException ie) {
                System.out.println("Sono stato
                interrotto...");
                return;
            }
        }
    }
}
```

*ereditato da Thread*

*blocco try-catch  
è necessario perché  
InterruptedException  
è di tipo controllato*

Altro esempio in cui vogliamo eseguire sleep ma siamo in una classe che non ha relazione con Thread

```
class Main {
    public static void main(String[] args) {
        Contatore c1 = new Contatore(300);
        ThreadCheConta t1 = new ThreadCheConta(c1);
        Contatore c2 = new Contatore(500);
    }
}
```

```

        ThreadCheConta t2 = new ThreadCheConta(c2);
        t1.start();
        t2.start();
    }
}

```

```

public class Contatore {
    private long quanto;

```

```

    public Contatore(long q) {
        quanto = q;
    }

```

```

    public void stampa(){
        for(int i=0; i<100; i++) {
            try{
                Thread t = Thread.currentThread();
                String n = t.getName();
                System.out.println(n + " " + i);
                Thread.sleep(quanto);
            } catch (InterruptedException ie) {
                System.out.println("Sono stato interrotto...");
                return;
            }
        }
    }
}

```

metodo static della classe  
Thread  
restituisce un riferimento  
al thread che esegue questa  
istruzione

ottengo nome del  
thread

Addeborando il thread  
che esegue questo  
pezzo di codice

Non sono in  
sottoclasse di Thread  
quindi NomeClasse.NomeMetodo

```

public class ThreadCheConta extends Thread {
    private Contatore c;

    public ThreadCheConta(Contatore c) {
        this.c = c;
    }

    public void run(){
        c.stampa();
    }
}

```

## Corse critiche

Esempio:

```

public class ContoBancario {
    int bilancio;

```

```

int 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();
    }
}

```

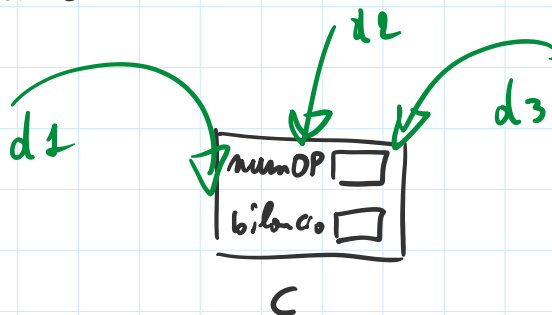
il vincolo non è rispettato  
 $27340 \neq 10 * 2735$

```

vecchio@DESKTOP-ARV2TPT: /mnt/c:/Users/aless_irdbul5/Dropbox/esempi/consecritiche$ java -ea Depositatore
Exception in thread "Thread-1" Exception in thread "Thread-2" Exception in thread "Thread-0" java.lang.AssertionError: numOperazioni=2735 bilancio=27340
    at ContoBancario.deposita(ContoBancario.java:8)
    at Depositatore.run(Depositatore.java:11)
java.lang.AssertionError: numOperazioni=2736 bilancio=27350
    at ContoBancario.deposita(ContoBancario.java:8)
    at Depositatore.run(Depositatore.java:11)
java.lang.AssertionError: numOperazioni=2735 bilancio=27340
    at ContoBancario.deposita(ContoBancario.java:8)
    at Depositatore.run(Depositatore.java:11)
vecchio@DESKTOP-ARV2TPT: /mnt/c:/Users/aless_irdbul5/Dropbox/esempi/consecritiche$

```

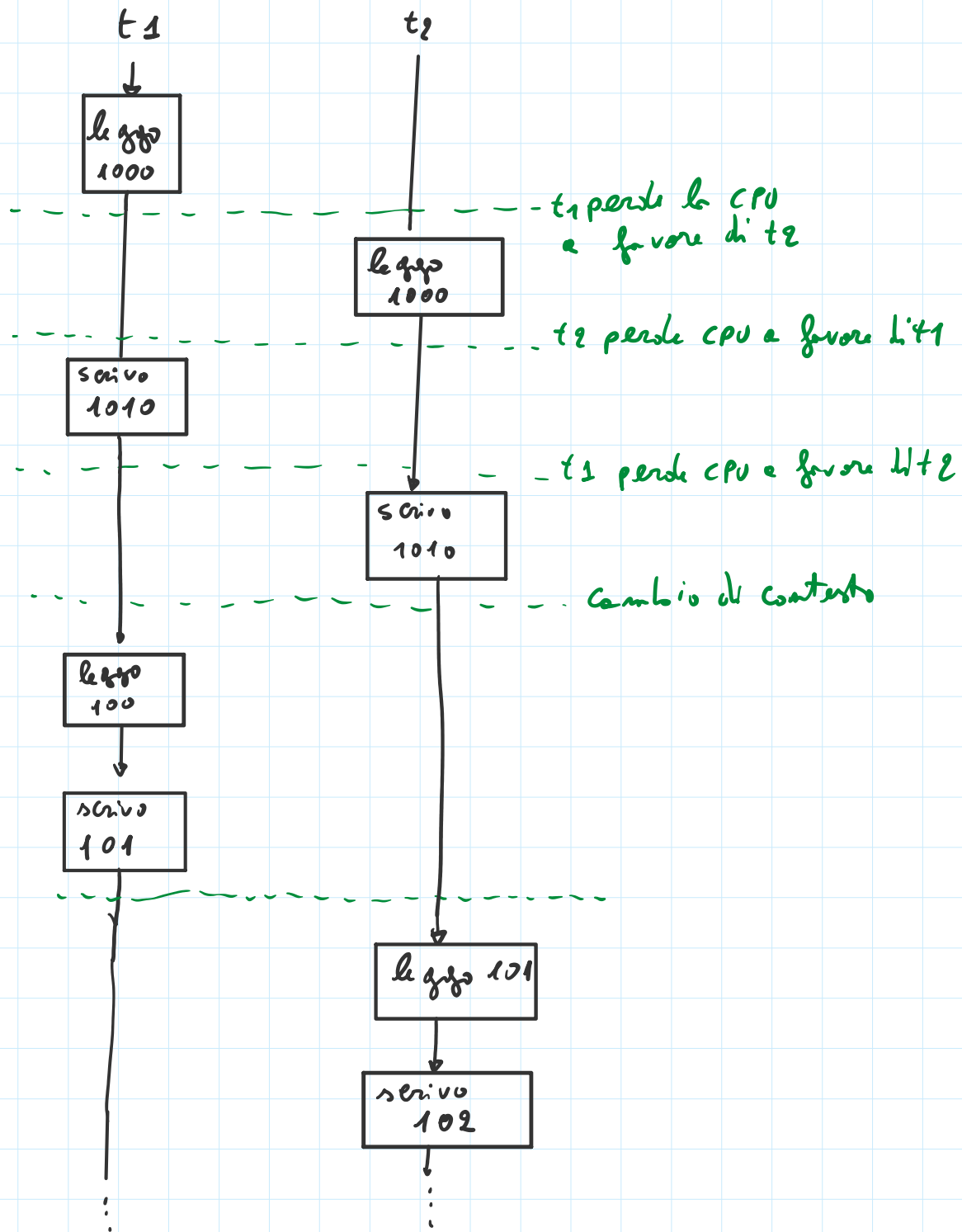
l'oggetto conto bancario è finito in uno stato inconsistente



Supponiamo di avere due soli thread  $t_1$  e  $t_2$   
 - che  $bilancio = 1000$  -  $numOperazioni = 100$



Supponiamo di avere due soli thread  $t_1$  e  $t_2$   
e che  $bilancio = 1000$  e  $numOperazioni = 100$

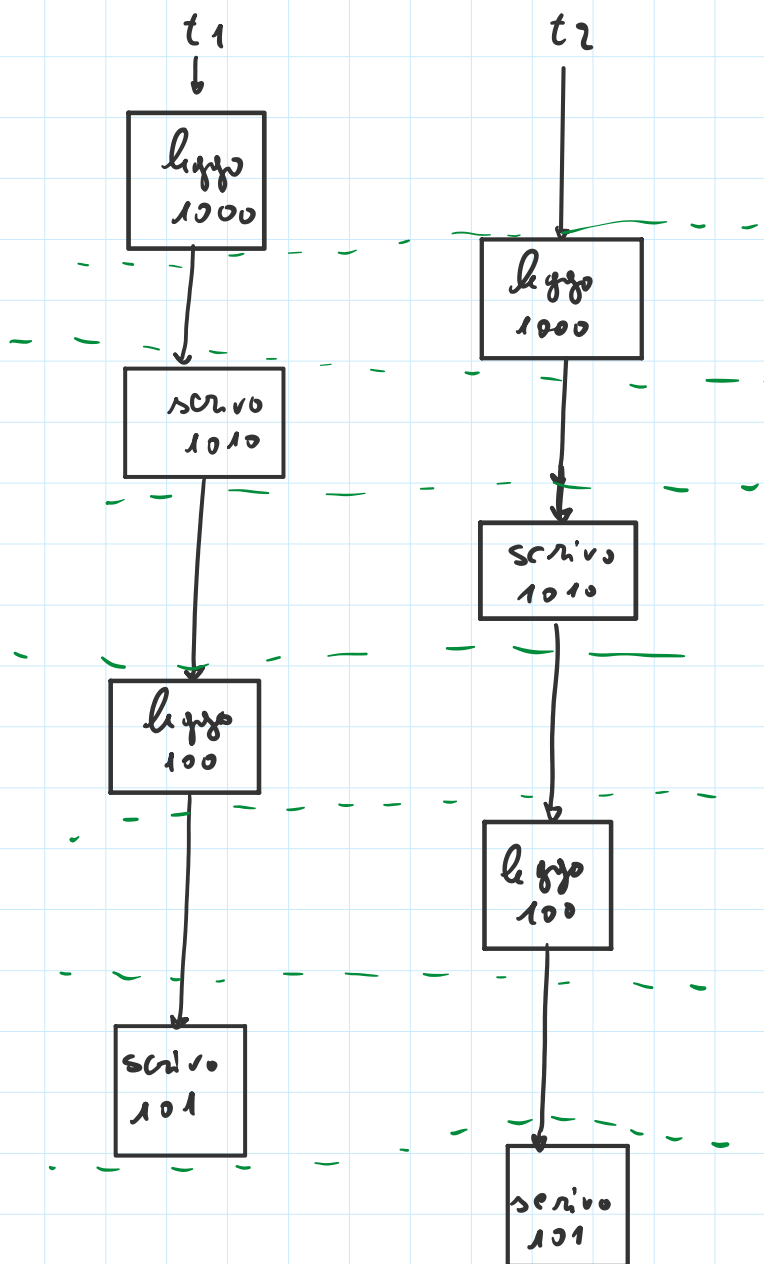


bilancio 1010  
numOp 102

I problemi derivano dal fatto che l'operazione  $deposita()$  non è atomica e un thread può

I problemi derivano dal fatto che l'operazione `deposita()` non è atomica e un thread può essere deschedulato quando l'oggetto conto bancario è in uno stato inconsistente.

Ci sono anche dei problemi che non sono in grado di rilevare, per esempio:



bilancio = 1010 , num Op. = 101

Ci sono state due operazioni di deposito

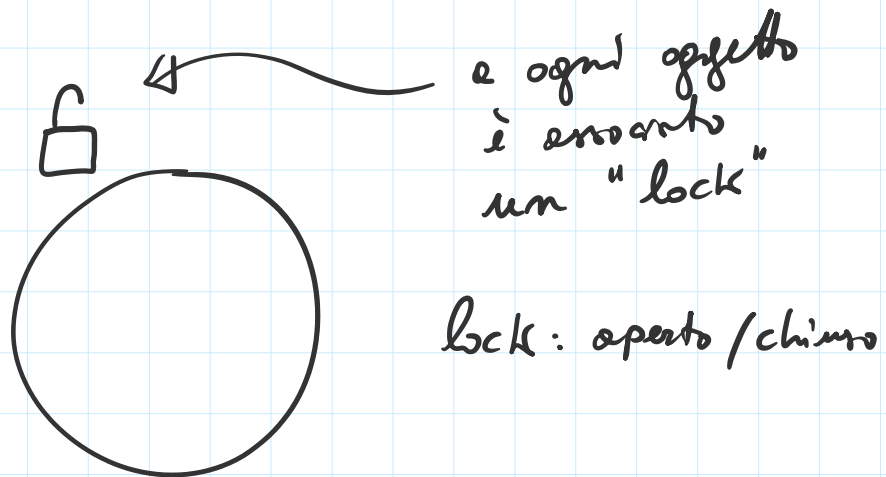
Ci sono state due operazioni di deposito

il valore di bilancio doveva essere 1020  
e quello di numOp. 102 //

l'assert che abbiamo usato esprime un  
vincolo relativo al singolo conto bancario

Non vengono espressi altri vincoli relativi  
al n° di operazioni e a quante volte  
deposito() è stato chiamato

In Java il problema viene risolto  
con il meccanismo dei monitor



i metodi possono essere indicati come  
synchronized

Se un thread vuole eseguire un  
metodo synchronized su un oggetto  
per poterlo fare deve avere il "lock" dell'oggetto

metodo `synchronized` su un oggetto  
per poterlo fare deve avere il "lock" dell'oggetto

Se un thread vuole acquisire il lock  
su un oggetto e il lock è posseduto da  
un altro thread allora il primo thread  
deve attendere che il secondo lo rilasci

Quindi, quando un thread esegue  
un metodo `synchronized` su un oggetto `x`

- 1) acquisisce il lock di `x`
- 2) esegue corpo metodo
- 3) rilascia il lock di `x`

Per quanto riguarda il nostro esempio di `ContoBancario` basta scrivere

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

quando un Depositatore chiama `deposita()`, deve per prima cosa ottenere il lock sull'oggetto `conto` e solo  
dopo passa a modificare l'oggetto. Al termine del metodo il lock viene rilasciato in modo tale che possa  
essere acquisito da altri Depositatori.

In pratica abbiamo una esecuzione in mutua esclusione del metodo `deposita()`

Se la classe fosse stata un po' più complessa, per esempio

```
public class ContoBancario2 {  
    int numOp;  
    int bilancio;  
    String intestatario;  
  
    public ContoBancario2(String s) {  
        intestatario = s;  
    }  
  
    public synchronized void deposita(){  
        bilancio += 10;  
        numOp++;  
    }  
  
    public synchronized void preleva() {  
        bilancio -= 10;  
        numOp++;  
    }  
  
    public String getIntestatario(){
```

*Nota: synchronized*

*non synchronized*

```
    return intestatario;  
  }  
}
```

Offriamo mutua esclusione deposito - preleva ,  
deposito - deposito , preleva - preleva

No mutua esclusione Tra get Intestatario  
e preleva o deposito