# VERSO LA CONCORRENZA

- **Finora, i nostri programmi erano costituiti da *un unico flusso di esecuzione***
- **Nelle moderne applicazioni, però, sorge spesso la necessità/opportunità di *avere più flussi di esecuzione "concorrenti"***
  - **per lanciare altre attività senza bloccarsi (es. stampe, scaricamento file da rete, etc)**
- **Tale "concorrenza" è simulata (in presenza di un unico processore fisico) dal S.O., *suddivi-dendo il tempo del processore* fra le diverse attività.**

# PROCESSI

**Il termine *processo* denota la *singola attività* eseguita in modo totalmente indipendente e separato rispetto alle altre.**

- **Tipicamente, un programma è eseguito da un processo**
- **Il processo è l'istanza in esecuzione di un programma**
- **Più processi possono eseguire lo stesso programma.**

# PROCESSI

**Ogni processo:**

- **ha uno *spazio di indirizzamento riservato***
- **non condivide memoria con gli altri**
  - **anche le variabili globali sono private del processo**
  - **anche i puntatori puntano comunque a cose diverse (rilocazione dei puntatori)**
- **ha un proprio *stack riservato***
  - **anche le variabili locali sono private del processo.**

# PROCESSI

**La *comunicazione tra processi* distinti*:***

- **non può avvenire in modo diretto tramite condivisione di variabili**
- **deve sfruttare appositi meccanismi di sistema** (ad esempio: file, socket, pipe..)

I processi consentono l'esecuzione di attività *concorrenti ma indipendenti* le une dalle altre

  - si minimizzano le possibilità di interazione, e quindi di disturbo reciproco.

# THREAD

**Un *thread* (o *processo leggero*) è un'attività autonoma che *vive all'interno di un processo***

- tipicamente un processo ospita al suo interno più thread in esecuzione concorrente

**Ogni thread ha un proprio *stack riservato...***

- **le variabili locali sono quindi private del singolo thread, non condivise con gli altri**

**..ma *non un'area dati riservata***

- ***tutti i thread dello stesso processo condividono il medesimo spazio di indirizzamento*.**

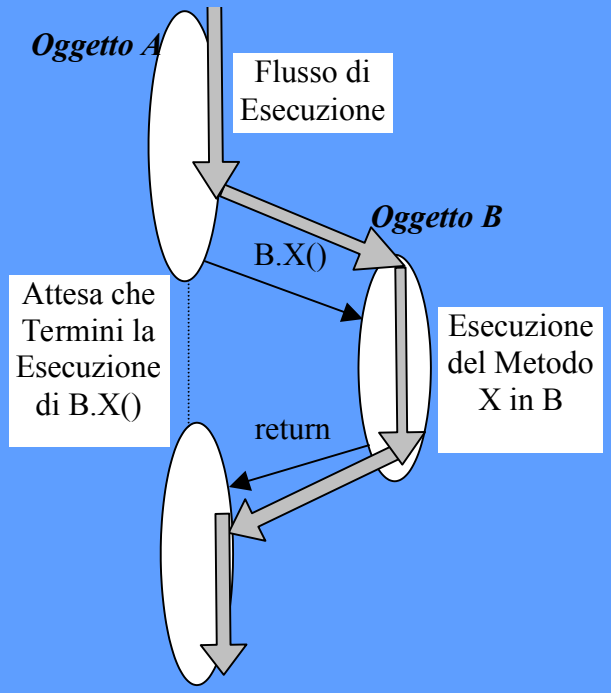
# THREAD

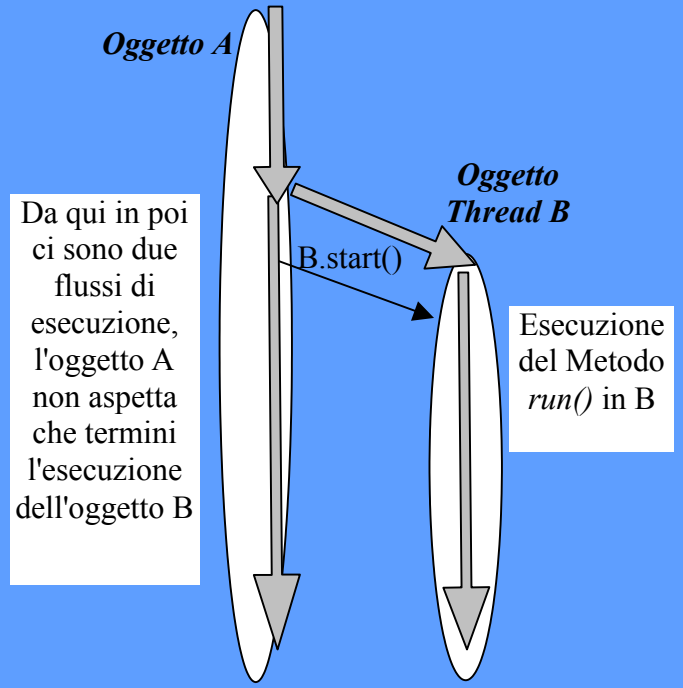
- La *comunicazione fra thread* può quindi avvenire in modo molto più semplice, tramite lo spazio di memoria condiviso
  - variabili globali, riferimenti a aree di memoria comuni, riferimenti a oggetti comuni
- Occorre però *disciplinare l'accesso* alle aree comuni → necessità di opportuni *meccanismi di sincronizzazione*
  - in un linguaggio ad oggetti, tipicamente più thread interagiscono tramite un oggetto condiviso, cui fanno tutti riferimento.

# THREAD IN JAVA

- **In Java, il concetto di Thread è parte del linguaggio**
- **I thread sono *oggetti* ai quali si *richiede l'attivazione di un servizio* mediante l'invocazione del metodo `start()`**
- **A differenza di una normale invocazione di funzione, non si aspetta che il servizio termini: il servizio procede in concorrenza al cliente che lo ha richiesto.**

# THREAD IN JAVA



# LA CLASSE `Thread`

**La classe `Thread` è la base per la gestione dei thread in Java:**

***Il cliente***

- ***crea un oggetto `Thread` opportuno…***
- **…e invoca su di esso il metodo `start()` per farlo partire.**

***Il thread (servitore)***

- **risponde iniziando a eseguire il corpo del metodo `run()`**

# USO DEI THREAD

*A priori*

- **si definisce una sottoclasse di `Thread` che ridefinisca opportunamente il metodo `run()` così da svolgere l'attività desiderata.**

*Il cliente*

- **crea una nuova istanza di tale classe**
- **attiva il nuovo thread invocando su tale istanza il metodo `start()`.**

# ESEMPIO (1/3)

**Si vuole realizzare un thread che, quando attivato, *stampa i quadrati dei primi 10 interi.***

- **si definisce una sottoclasse di `Thread` che ridefinisca opportunamente il metodo `run()` in modo da svolgere l'attività desiderata**

```
public class MyThread1 extends Thread {
   public void run(){
    for(int i=1; i<=10; i++)
     System.out.println(i + " " + i*i);
   }
   }
```

# ESEMPIO (2/3)

**Si vuole realizzare un thread che, quando attivato, *stampa i quadrati dei primi 10 interi.***

- **Poi si crea una nuova istanza della classe…**

```
public class EsThread1 {
    public static void main(String[]
a){
      MyThread1 t = new MyThread1();
      ...
}
```

# ESEMPIO (3/3)

**Si vuole realizzare un thread che, quando attivato, *stampa i quadrati dei primi 10 interi.***

- **poi si crea una nuova istanza di tale classe...**
- **… e si attiva il nuovo thread invocando su tale istanza il metodo `start()`.**

```
public class EsThread1 {
public static void main(String[] a){
 MyThread1 t = new MyThread1();
t.start();
}
```

# ESEMPIO (3/3)

**Si vuole realizzare un thread che, quando attivato, *stampa i quadrati dei primi 10 interi.***

- **poi si crea una nuova istanza di tale classe...**
- **... nvocando su ta ) .**

Unendo i due passi:

```
new MyThread1().start();
```

```
public  s EsThread1 {
public static void main(String[] a){
 MyThread1 t = new MyThread1();
t.start();
}
```

# ESEMPIO - note

- La versione con la variabile **`t`** esplicitamente definita è utile qualora si debbano invocare altri metodi oltre all'iniziale **`start()`**, altrimenti basta scrivere **`new MyThread1().start();`**
- Il nuovo thread continua di norma la propria esecuzione fino alla sua terminazione naturale
- **Ogni thread è sempre associato a un *nome univoco* (una stringa)** che viene assegnato o per default dal sistema o dall'utente. Per recuperarlo, si usa il metodo **`getName()`**.

# ESEMPIO - Thread con nome

**Modifichiamo il main in modo che visualizzi il *nome di default* del thread creato:**

```
public class EsThread1 {
public static void main(String[] a){
 MyThread1 t = new MyThread1();
t.start();
System.out.println("Nome: " +
    t.getName());
}
```

```
Nome: Thread-0
...
```

# ESEMPIO - Thread con nome

**È anche possibile *specificare un nome di nostra scelta*, passandolo (tramite super) al costruttore della classe `MyThread`:**

```
public class MyThread2 extends Thread {
   public MyThread2(String s) { super(s); }
   public void run(){
   for(int i=1; i<=10; i++)
    System.out.println(i + " " + i*i);
   }
}
```

# ESEMPIO - Thread con nome

**Riformulando l'esempio:**

```
public class EsThread2 {
public static void main(String[] a){
 MyThread2 t =
new MyThread2("Dieci quadrati");
t.start();
System.out.println("Nome: " +
   t.getName());
}
```

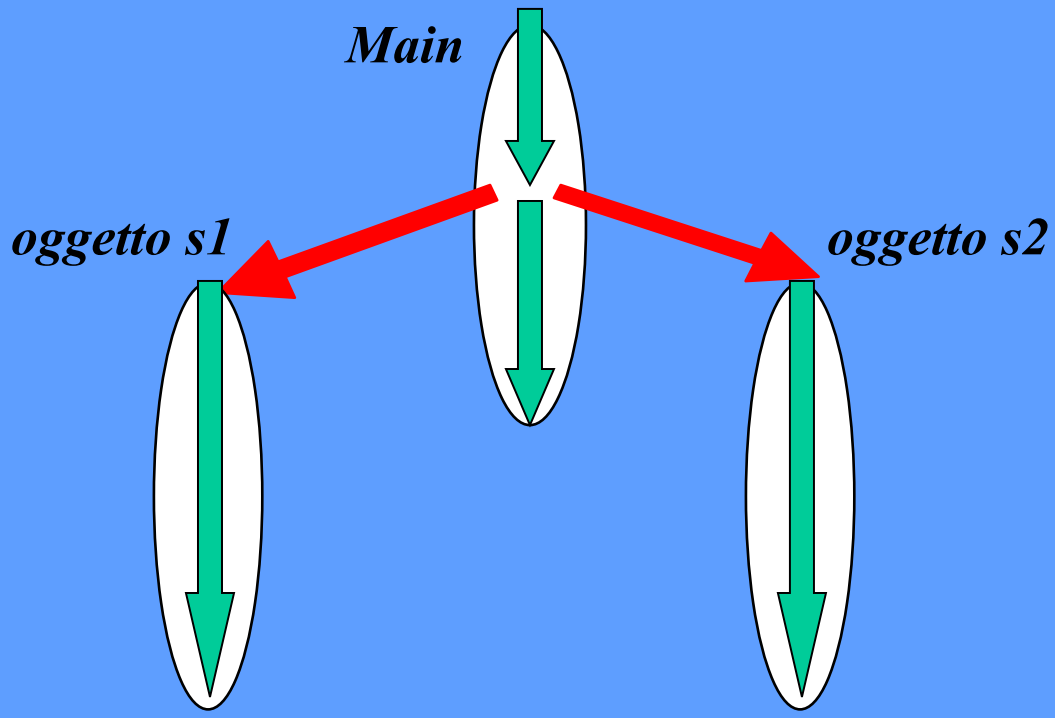
```
Nome: Dieci quadrati
…
```

# UN ALTRO ESEMPIO

- Si vuole definire una classe thread `Squares` il cui costruttore abbia come parametri un intero N e un oggetto `Writer`
- Il nuovo thread deve scrivere sul `Writer` specificato i quadrati dei primi N numeri.
- Esempio d'uso:

```
Squares s1 = new Squares(100,
 new FileWriter("100.txt"));
Squares s2 = new Squares(40,
  new OutputStreamWriter(System.out));
s1.start(); s2.start();
```

# UN ALTRO ESEMPIO - STRUTTURA



# LA CLASSE Squares

```
import java.io.*;
public class Squares extends Thread {
 private PrintWriter writer;
 private int num;
 public Squares(int n, Writer w) {
   super("Squares " + n); // nome del thread
   writer = new PrintWriter(w);   num = n;
 }
 public void run(){
   for(int i=1; i<=num; i++)
     writer.println(i + " " + i*i);
   writer.flush();
 }
}
```

# LA CLASSE DI PROVA

```
import java.io.*;
public class EsSquares {
 public static void main(String args[]) {
   FileWriter fout = null;
   try { fout = new FileWriter("100.txt"); }
   catch(IOException e) {
   System.err.println("File non aperto"); }
   Squares s1 = new Squares(100, fout);
   Squares s2 = new Squares(40,
     new OutputStreamWriter(System.out));
   s1.start(); s2.start();
 }
}
```

# LA CLASSE DI PROVA

```
import java.io.*;
public class EsSquares {
 public
   Fil
   try                                                    }
   catch(
   System.err.println("File                    rto"); }
   Squares s1 = new Squares(                    fout);
   Squares s2 = new Squares(
     new OutputStreamWriter(                stem.out));
   s1.start(); s2.start();
 }
}
```

**Cosa succede** se il main si mette a sua volta a scrivere su **`System.out`**?

# OLTRE LA CLASSE `Thread`

**Dover derivare ogni thread dalla classe `Thread` è una *forte limitazione*:**

- **dato che Java non supporta l'ereditarietà multipla tra classi, la classe che specifica il thread *non può essere sottoclasse di null'altro*!!**

**Non è accettabile che il progetto del sistema debba sottostare a vincoli di meccanismo!**

- **Occorre un'alternativa.**

# L'INTERFACCIA `Runnable`

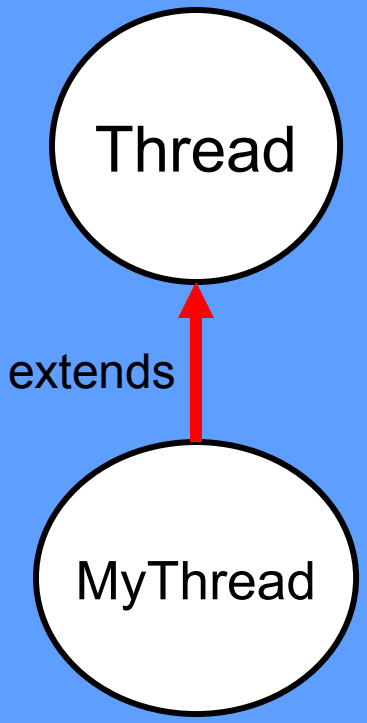
**Anziché derivare ogni thread dalla classe `Thread`, è possibile:**

- **definire la classe che fornisce il thread *come classe a sé stante…***
- **…facendo però in modo che implementi *l'interfaccia `Runnable`,* che dichiara il metodo `run()`.**

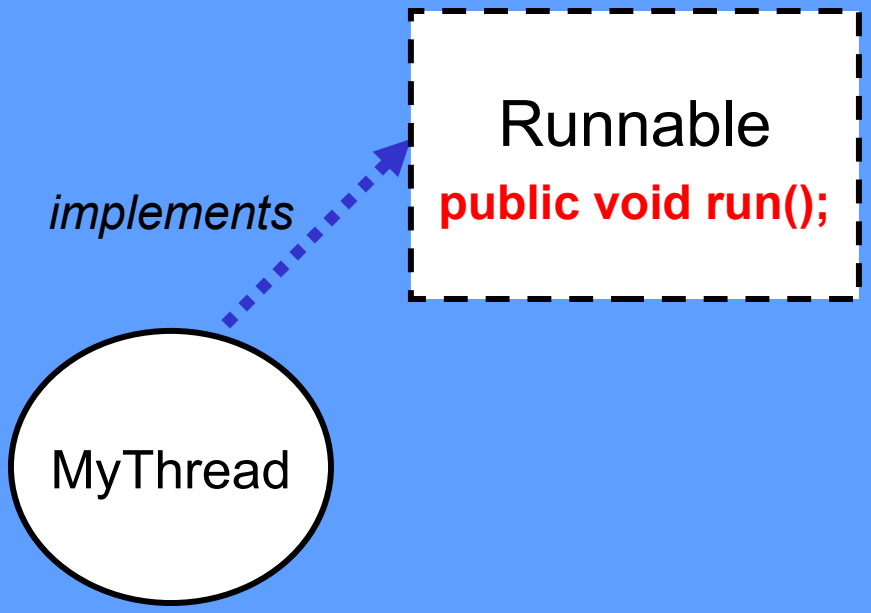
**Così, la classe che definisce il thread non è più obbligatoriamente sottoclasse di `Thread`, e può essere collocata ovunque nella gerarchia.**

# I DUE APPROCCI A CONFRONTO

***Prima***




***Adesso***



# IL NUOVO APPROCCIO

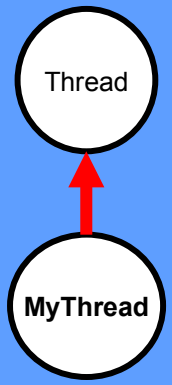
**Con questo approccio, per creare un thread occorre:**

- **prima definire la classe che implementa *l'interfaccia* `Runnable` e definisce il metodo `run()`**
- **poi creare un oggetto di tale classe**
- **infine istanziare un oggetto `Thread` *passandogli come parametro l'oggetto creato al punto precedente, e avviarlo*.**

# CREAZIONE DI UN THREAD

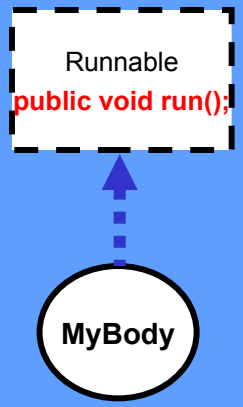
***Prima***

```
MyThread t = new MyThread();
t.start();
```




***Adesso***

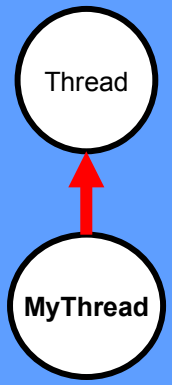
```
MyBody body = new MyBody();
Thread t = new Thread(body);
t.start();
```



# CREAZIONE DI UN THREAD

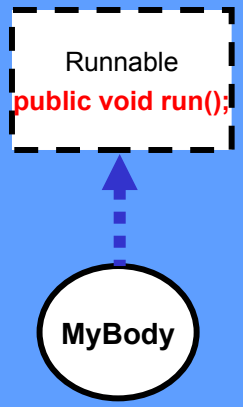
**Attenzione:**

- la classe che fornisce il corpo (body) è separata dalla classe Thread
- **ma occorre comunque creare un oggetto Thread** per avere un thread da avviare!




***Adesso***

```
MyBody body = new MyBody();
Thread t = new Thread(body);
t.start();
```



# L'ESEMPIO RIVISTO

- **Ora, la classe `MyBody` implementa l'interfaccia `Runnable` e definisce il metodo `run()`, *ma non è essa stessa il thread!***

```
MyBody body = new MyBody();
```

- **Il thread viene creato istanziando un oggetto `Thread` avente come parametro tale oggetto `MyBody`:**

```
Thread t = new Thread(body);
t.start();
```

# L'ESEMPIO RIVISTO

**Volendo dare un nome al thread:**

- **non si può più passarlo tramite `super()`, perché la classe che definisce il body *non deriva più da `Thread`***
- **si può però specificarlo come II° parametro in fase di costruzione dell'oggetto `Thread`:**

```
MyBody body = new MyBody();
Thread t = new Thread(body, "Quadrati");
t.start();
```

# CONTROLLO DEI THREAD

***E dopo che un thread è stato avviato…?***

- **si può fermarlo mediante `stop()`…**
- **sospenderne l'esecuzione con `suspend()`...**
- **...e riprenderne l'esecuzione con `resume()`.**

**Tuttavia, *questi metodi agiscono in modo brutale*: fermano, sospendono o riattivano il thread *senza tenere conto delle sue condizioni.* Quindi, sono deprecati perché possono portare a instabilità dell'applicazione.**

# UN ESEMPIO "DEPRECATO"

- Si attiva un thread, lo si lascia andare per 1 s..
- ..lo si sospende per 1,5 s, poi lo si riprende…
- …e infine, dopo 0,5 s, lo si blocca.

```
public class EsThread4 {
  public static void main(String args[]){
   MyThread4 t4 = new MyThread4("Stop & Go");
    try {
   t4.start();          Thread.sleep(1000);
   t4.suspend();        Thread.sleep(1500);
   t4.resume();         Thread.sleep(500);
   t4.stop();
    } catch(InterruptedException e){}
  }
}
```

**Tempi minimi di sospensione.**

# UN ESEMPIO "DEPRECATO"

```
public class MyThread4 extends Thread {
   public MyThread4(String s) { super(s); }
   public void run(){
    for(int i=1; i<=100000; i++)
   // dev'essere abbastanza lungo...
     System.out.println(i);
   }
}
```

- Il thread parte, arriva fin verso 3500...
- ..si ferma per 1,5 s, riprende fin verso 5400...
- …e infine si ferma, stroncato da `stop()`.

# UN ESEMPIO "DEPRECATO"

***Perché tale approccio è deprecato in Java2?***

- **Perché `stop()`, `suspend()` e `resume()` *agiscono in modo brutale* sul thread, *che non può "difendersi" in alcun modo!***
- **Così facendo rischiano di creare situazioni di *deadlock* (blocco)**

# IL PROBLEMA DEL DEADLOCK

**Un *deadlock* è una situazione di blocco:**

- **un thread A sta usando una risorsa (un file, un oggetto, un video, etc)…**
- **...viene interrotto da un thread B...**
- **..e anche B vuole usare la stessa risorsa, *che però è ancora occupata da A!***
- **Morale: A è bloccato da B, che a sua volta però è bloccato da A** (è in attesa che A liberi la risorsa) → **è *tutto bloccato:* nessuno dei due potrà *mai più* proseguire!**

# UN ESEMPIO "DEPRECATO"

**E infatti...**

- **se nel main si aggiunge *anche solo una println*, *dopo una stampa si blocca tutto!!!***

```
public class EsThread4Bis {
  public static void main(String args[]){
    ...
    t2.start();          Thread.sleep(1000);
    t2.suspend(); Thread.sleep(1500);
    System.out.println("Ci sono anche io!");
    t2.resume();  Thread.sleep(500);
    ...
}
```

# QUALE SOLUZIONE?

**In Java2 si suggerisce di *cambiare approccio:***

- **non più "ordini brutali"…**
- **..ma bensì *segnalare al thread l'opportunità di sospendersi o terminare...***
- **.. lasciando al thread decidere *quando farlo.***

---

**Un metodo utilizzabile a questo fine:**

- `Thread.yield()` ***provoca la cessione volontaria del controllo*** **a un altro thread pronto per l'esecuzione.**

# COROUTINING

***Due o più thread che si cedono reciprocamente il controllo si dicono "co-routines"***

- quando un thread è disposto a cedere il controllo chiama la funzione statica **`Thread.yield()`**
- il controllo passa al primo thread pronto per l'esecuzione (compatibilmente con la sua priorità)

**Esempio: Cip & Ciop**

- Due thread identici
- Ognuno stampa il suo nome, poi cede il controllo con `yield()`.

# COROUTINING TEST ("Cip & Ciop")

```
public class CipCiop {
 public static void main(String args[]){
  CipCiopBody body = new CipCiopBody();
  new Thread(body, "cip ").start();
  new Thread(body, "ciop").start();
   }
}

class CipCiopBody implements Runnable {
 public void run(){
  while (true) { System.out.println(
  Thread.currentThread().getName() );
    Thread.yield();
   }
 }
}
```

Provare con tre o più thread!

# SINCRONIZZAZIONE

- **Problema: se due thread accedono allo stessa risorsa contemporaneamente possono verificarsi *inconsistenze*.**
- **Per evitarle occorre disporre di appositi *meccanismi di sincronizzazione,* che garantiscano *accesso esclusivo* alla risorsa condivisa per tutta la durata dell'operazione critica.**

# ESEMPIO (1/3)

**Si supponga che la classe `ContoCorrente` fornisca un metodo `versamento(double v)` per versare sul conto la quantità `v`, e che ogni versamento sia soggetto a una tassa fissa:**

```
public class ContoCorrente {
  double tot;
  double final TASSA = 100;
  ...
 public void versamento(double v) {
  double newTot = tot + v - TASSA;
  tot = newTot;
 }
}
```

# ESEMPIO (2/3)

**Si supponga ora che due thread usino entrambi *il medesimo oggetto* `ContoCorrente`, e tentino di fare un versamento *nello stesso momento*.**

- **Se il valore iniziale del conto è 2000, ed essi versano uno 500 e l'altro 1000, ci si aspetterebbe che il totale finale fosse 3300 (tasse comprese)…**
- ***..ma le cose potrebbero non andare così:*** **dipende da *quando* si fanno i due versamenti.**

# ESEMPIO (3/3)

**Se però la successione delle operazioni è, per esempio, la seguente...**

```
c.versamento(1000);
// il thread invoca il metodo


newTot = tot + v - TASSA;
// newTot = 2000 + 1000 -100
// (variabile locale al metodo)



tot = newTot;
// tot vale 2900...
// (variabile DELL'OGGETTO)
```

```
c.versamento(500);
// il thread invoca il metodo


newTot = tot + v - TASSA;
// newTot = 2000 + 500 -100
// (variabile locale al metodo)
```

**… alla fine il totale sul conto è 2400 !!**

# SINCRONIZZAZIONE

- **Per evitare inconsistenze, occorre che *un solo thread per volta* possa eseguire la sezione critica.**
- **In Java la sincronizzazione può avvenire:**
  - **a livello di metodo**
    **si impedisce l'esecuzione concorrente di un intero metodo su un dato oggetto**
  - **a livello di porzione di codice**
    **si impedisce l'esecuzione concorrente solo di una sezione critica di un certo metodo su un dato oggetto.**

# METODI SINCRONIZZATI

- **Per sincronizzare l'accesso a un intero metodo, impedendone l'esecuzione concorrente sul medesimo oggetto, basta dichiarare il metodo *synchronized***
- **Ad esempio:**

```
public class ContoCorrente {
...
public synchronized void versamento(double v) {
    double newTot = tot + v - TASSA;
    tot = newTot;
}
}
```

# METODI SINCRONIZZATI

- **Per si… …metodo, impec… …ul mede… …todo *synci…***
- **Ad es…**

**Quando un thread invoca il metodo, l'accesso al metodo viene bloccato: un eventuale secondo thread che lo invochi sul medesimo oggetto deve aspettare che il primo esca.**

```
public class …corrente {
...
public synchronized void versamento(double v) {
    double newTot = tot + v - TASSA;
    tot = newTot;
}
}
```

**Niente più inconsistenze!**

Adesso, `tot` vale 3300.

# L'ESEMPIO CORRETTO

**Indipendentemente dalla successione delle invocazioni, le operazioni sono sequenzializzate:**

```
c.versamento(1000);
// il thread invoca il metodo


newTot = tot + v - TASSA;
// newTot = 2000 + 1000 -100

tot = newTot;
// tot vale 2900
// QUESTO THREAD TERMINA.
```

```
c.versamento(500);
// il thread invoca il metodo
// ma è costretto ad attendere


// FINALMENTE QUESTO PUÒ ENTRA
// ORA tot vale 2900
newTot = tot + v - TASSA;
// newTot = 2900 + 500 -100)
```

# L'ESEMPIO COMPLETO (1/4)

**La classe `ContoCorrente`:**

```
class ContoCorrente {
 double tot;  final double TASSA = 100;
 public ContoCorrente (double x) { tot = x; }
 public synchronized double valore() {
   return tot; }
 public synchronized void versamento(double v) {
  double newTot = tot + v - TASSA;
  // attesa per rendere visibile il fenomeno
  try{ Thread.sleep(1000); }
  catch(InterruptedException e){}
  tot = newTot;
 }
}
```

# L'ESEMPIO COMPLETO (2/4)

**La classe `CCBody`:**

```
class CCBody implements Runnable {
 ContoCorrente c;  double s;
 public CCBody(ContoCorrente cx, double v) {
   c = cx;  s = v;}
 public void run(){
  System.out.println( "Sto per versare " + s );
  System.out.println( "Saldo: " + c.valore() );
  c.versamento(somma);
  System.out.println( "Ho versato " + somma );
  System.out.println( "Saldo: " + c.valore() );
 }
}
```

# L'ESEMPIO COMPLETO (3/4)

**La classe main:**

```
public class EsContoCorrenteNo {
 public static void main(String args[]){
  ContoCorrente c = new ContoCorrente(2000);
  new Thread(new CCBody(c, 1000)).start();
  new Thread(new CCBody(c,  500)).start();
 }
}
```

# L'ESEMPIO COMPLETO (4/4)

***Senza synchronized:***

```
Sto per versare 1000.0
Saldo: 2000.0
Sto per versare 500.0
Saldo: 2000.0
Ho versato 1000.0
Saldo: 2900.0
Ho versato 500.0
Saldo: 2400.0
```

***Con synchronized:***

```
Sto per versare 1000.0
Saldo: 2000.0
Sto per versare 500.0
Saldo: 2900.0
Ho versato 1000.0
Ho versato 500.0
Saldo: 3300.0
Saldo: 3300.0
```