



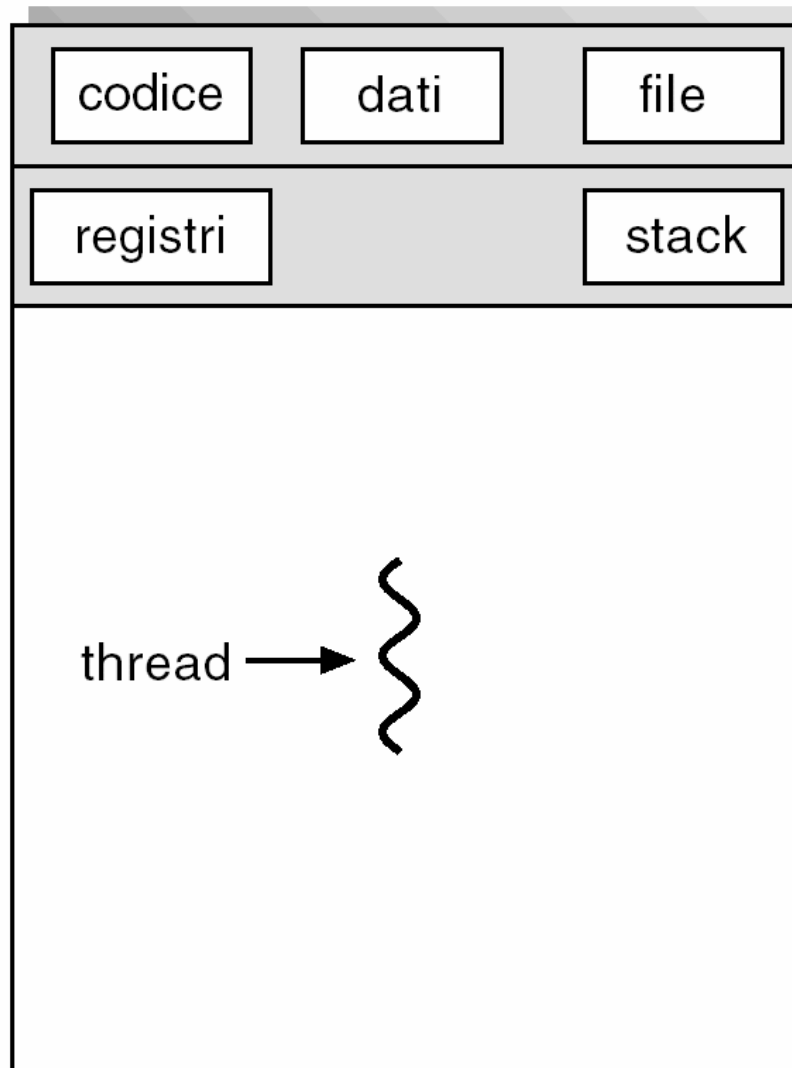
Threads e Programmazione Multithreading



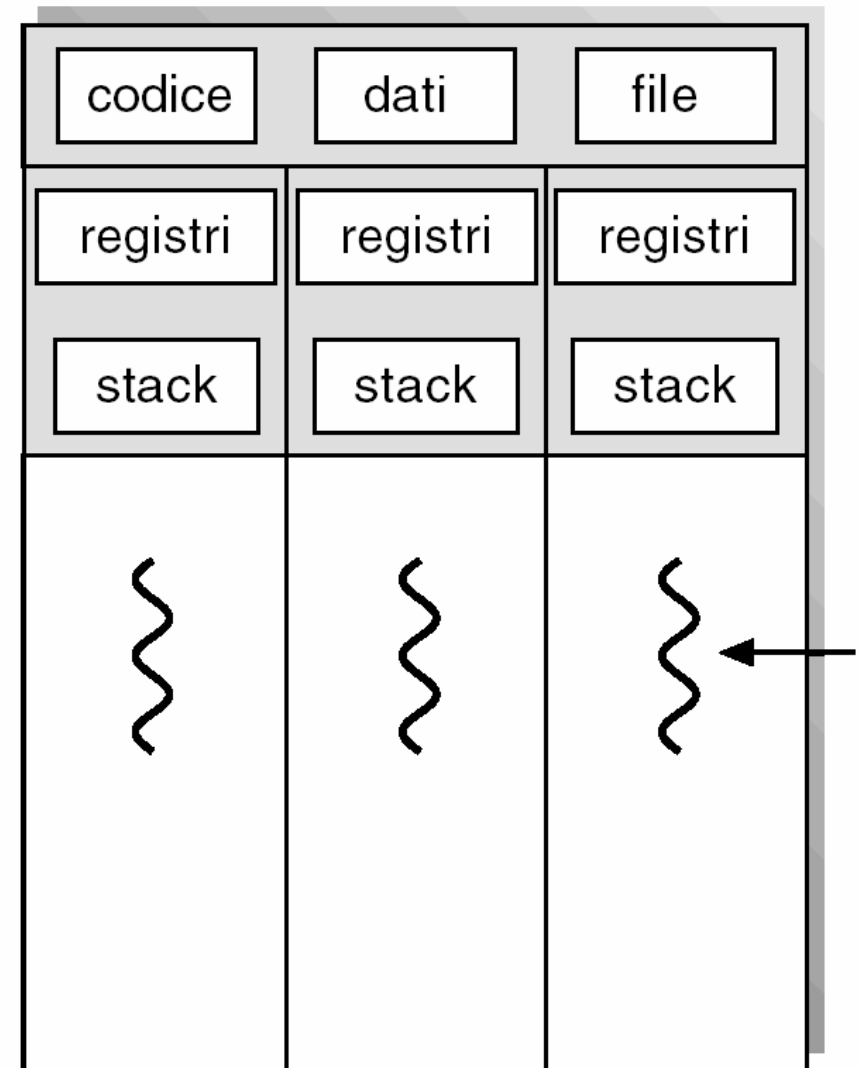
Cosa è un thread

- È un flusso di controllo all'interno di un programma
 - Ma ce ne può essere più di uno
- È una versione leggera di un processo
 - Un thread non ha allocate per sé tutte le risorse che ha un processo
 - Ad esempio non ha un proprio spazio di indirizzamento privato
- È un contesto di esecuzione
 - Ha un proprio program counter ed un proprio stack

Processi a singolo thread e multithread



processo a singolo thread



processo multithread



Utilizzo dei Thread

- Per migliorare l'interazione con l'utente
 - mantenendo rapide tutte le interazioni con l'utente
 - le operazioni lunghe vengono svolte da thread dedicati senza compromettere la reattività del thread che gestisce l'interazione con l'utente
- Per simulare attività simultanee
- Per sfruttare sistemi multi-processore
 - al fine di dedicare più processori all'esecuzione dello stesso programma




java.lang.Thread

- I Thread della JVM sono associati ad istanze della classe `java.lang.Thread`
- Gli oggetti istanza di tale classe svolgono la funzione di interfaccia verso la JVM che è l'unica capace di creare effettivamente nuovi thread
- Attenzione a non confondere il concetto di thread con gli oggetti istanza della classe `java.lang.Thread`
 - tali oggetti sono solo lo strumento con il quale è possibile *comunicare* alla JVM
 - di creare nuovi thread
 - di interrompere dei thread esistenti
 - di attendere la fine di un thread (join)
 - ...



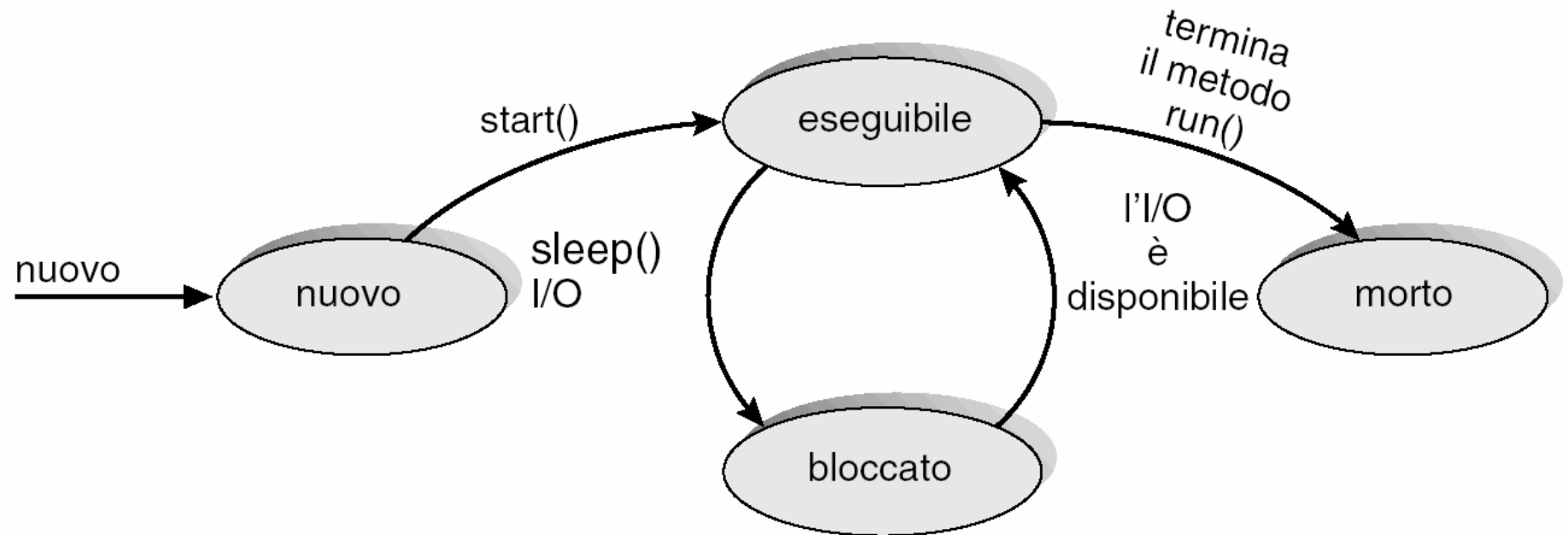
Il thread principale

- In Java ogni programma in esecuzione è un thread
- il metodo *main()* è associato al *main thread*
- per poter accedere alle proprietà del *main thread* è necessario ottenerne un riferimento tramite il metodo *currentThread()*



```
public class ThreadMain {
    public static void main(String args[ ]) {
        Thread t = Thread.currentThread();
        System.out.println("Thread corrente: " + t);
        t.setName("Mio Thread");
        System.out.println("Dopo aver cambiato il nome: " + t);
        try {
            for (int n=5; n>0;n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) { }
    }
}
```

Diagramma degli stati dei thread in Java





2 Modi per Creare Thread in Java

- Instanziare classi che derivano da `java.lang.Thread`
 - più semplice
 - ma non è possibile ereditare da altre classi
- Instanziare direttamente `Thread` passando al suo costruttore un oggetto di interfaccia `Runnable`
 - bisogna implementare l'interfaccia `Runnable` e creare esplicitamente l'istanza di `Thread`
 - ma la classe rimane libera di ereditare da una qualsiasi altra classe



Creazione di un nuovo thread

Primo modo per creare un nuovo thread: dichiarare una classe derivata da **Thread**, che deve sovrascrivere il metodo **run**

```
public class MioThread extends Thread {  
    MioThread(...) {  
        // codice del costruttore  
    }  
    public void run() {  
        // codice eseguito dal nuovo thread  
    }  
}
```

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

```
public class NuovoThreadThr extends Thread {  
    public NuovoThreadThr() { // creo un nuovo thread con il  
                               // costruttore  
        super("Thread secondario"); // chiamo il costruttore della  
                                     superclasse  
        System.out.println("\t\t Thread figlio: " + this);  
    }  
  
    public void run() { // corpo del nuovo thread  
        try {  
            for (int n=5; n>0;n--) {  
                System.out.println("\t\t Thread figlio: " + n);  
                Thread.sleep(500);  
            }  
        } catch (InterruptedException e) { };  
        System.out.println("\t\t Termine del thread figlio");  
    }  
}
```

```
public class ThreadPrincipaleThr {  
    public static void main(String args[]) {  
        System.out.println("Thread padre: " + Thread.currentThread());  
        Thread t=new NuovoThreadThr(); // creo un nuovo thread secondario  
        t.start();  
        try {  
            for (int n=5; n>0;n--) {  
                System.out.println("Thread padre: " + n);  
                Thread.sleep(1000);  
            }  
        } catch (InterruptedException e) { };  
        System.out.println("Termine del thread padre");  
    }  
}
```



Creazione di un nuovo thread

- Utilizzo dell'interfaccia Runnable:
 - dichiarare una classe che implementa l'interfaccia **Runnable**, che deve implementare il metodo **run()**
 - per crearla, la classe va istanziata, passata come argomento chiamando la classe **Thread**, e poi fatta partire con **start()**



Creazione di un nuovo thread (1)

```
public class MioThread implements Runnable
{
    MioThread(...) {
        // codice del costruttore
    }

    public void run() {
        // codice eseguito dal nuovo thread
    }
}
```

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

```

class NuovoThreadRun implements Runnable {
    public void run() { // corpo del nuovo thread
        try {
            for (int n=5; n>0;n--) {
                System.out.println("\t\tThread figlio: " + n);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) { };
        System.out.println("\t\tTermine del thread figlio");
    }
}

public class TestThreadWithRunnable {
    public static void main(String args[]) {
        System.out.println("Thread padre: " + Thread.currentThread());
        Thread t=new Thread(new NuovoThreadRun()); // creo un nuovo thread
        t.start();
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) { };
        System.out.println("Termine del thread padre");
    }
}

```

```
public class NuovoThreadRun implements Runnable {
    Thread t;
    public NuovoThreadRun() {
        t = new Thread(this, "Thread secondario");
        System.out.println("\t\tThread figlio: " + t);
        t.start(); // avvio il nuovo thread
    }
    public void run() { // corpo del nuovo thread
        try {
            for (int n=5; n>0;n--) {
                System.out.println("\t\tThread figlio: " + n);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) { };
        System.out.println("\t\tTermine del thread figlio");
    }
}
```


Versione a più thread

```
public class MultiThread implements Runnable {
    String nome;
    Thread t;
    public MultiThread(String nomeThread) { // creo un nuovo thread
        nome = nomeThread;
        t = new Thread(this,nome);
        System.out.println("\t\tNuovo Thread: " + t);
        t.start(); // avvio il nuovo thread
    }
    public void run() { // corpo del nuovo thread
        try {
            for (int n=5; n>0;n--) {
                System.out.println("\t\t"+nome+": " + n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) { };
        System.out.println("\t\tTermine del thread "+nome);
    }
}
```

Versione a più thread

```
public class ThreadMultipliDemo {  
    public static void main(String args[]) {  
        System.out.println("Thread padre: " + Thread.currentThread());  
        new MultiThread("Uno"); // creo un nuovo thread secondario  
        new MultiThread("Due");  
        new MultiThread("Tre");  
        try {  
            Thread.sleep(10000);  
        } catch (InterruptedException e) { };  
        System.out.println("Termine del thread padre");  
    }  
}
```



Ricongiunzione di thread

```
class JoinableWorker implements Runnable
{
    public void run() {
        System.out.println("Worker working");
    }
}

public class JoinExample
{
    public static void main(String[] args) {
        Thread task = new Thread(new JoinableWorker());
        task.start();
        try { task.join(); } // attende finché task termina
        catch (InterruptedException ie) { }
        System.out.println("Worker done");
    }
}
```

Versione a più thread

```
public class ThreadMultipliDemoOK {  
    public static void main(String args[]) {  
        System.out.println("Thread padre: " + Thread.currentThread());  
        MultiThread th1 = new MultiThread("Uno");  
        MultiThread th2 = new MultiThread("Due");  
        MultiThread th3 = new MultiThread("Tre");  
        try {  
            th1.t.join();  
            th2.t.join();  
            th3.t.join();  
        } catch (InterruptedException e) { };  
        System.out.println("Termine del thread padre");  
    }  
}
```



Cancellazione dei thread

```
Thread thrd = new Thread (new InterruptibleThread());  
thrd.start();  
. . .
```

```
thrd.interrupt(); // viene interrotta la sua esecuzione
```



La priorità dei thread

- Java fissa una scala di valori, che è diversa nelle varie implementazioni della JVM
- questa scala è garantita solo all'interno del linguaggio
- poi viene filtrata e adeguata dal S.O. e dall'ambiente hardware

```

public class Contatore implements Runnable {
    int conta = 0;
    Thread t;
    private boolean running = true;
    public Contatore(int p, String nomeThread) {
        t = new Thread(this,nomeThread);
        t.setPriority(p);
        System.out.println("Thread figlio: " + t);
    }
    public void run() { // corpo del nuovo thread
        while (running) {
            conta++;
            System.out.print("\r");
        }
    }
    public void start() {
        t.start();
    }
    public void stop() {
        running = false;
    }
}

```

```

public class ContatoreDemo {
    public static void main(String args[]) {
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        System.out.println("Thread padre: " + Thread.currentThread());
        Contatore th1 = new Contatore(Thread.NORM_PRIORITY + 2, "Uno");
        Contatore th2 = new Contatore(Thread.NORM_PRIORITY - 2, "Due");
        th1.start();
        th2.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) { };
        th1.stop();
        th2.stop();
        try {
            th1.t.join();
            th2.t.join();
        } catch (InterruptedException e) { };
        System.out.println("Thread ad alta priorita': "+th1.conta);
        System.out.println("Thread a bassa priorita': "+th2.conta);
    }
}

```




Thread in Java:

Creazione e Terminazione

- Creazione di Thread
 - si creano oggetti istanze della classe `java.lang.Thread`.
 - Il thread viene effettivamente creato dalla JVM non appena si richiama il metodo `start()`
 - Il nuovo thread esegue il metodo `run()`
- Terminazione di Thread
 - i thread terminano quando
 - finisce l'esecuzione del metodo `run()`
 - sono stati interrotti con il metodo `interrupt()`
 - eccezioni ed errori
- Join con Thread
 - richiamando il metodo `join()` su un oggetto Thread si blocca il thread corrente sino alla terminazione del thread associato a tale oggetto

Thread in Java: Creazione Classi interne anonime

```
Thread t = new Thread() { //an anonymous inner class that extends Thread
    public void run() { // override to specify the running behaviors
        for (int i = 0; i < 100000; i++) {
            if (stop) break;
            tfCount.setText("" + countValue);
            countValue++;
            // suspend itself and yield control to other threads
            try {
                sleep(10);
            } catch (InterruptedException ex) {}
        }
    }
};
t.start();
```

Thread in Java: Creazione Classi interne anonime

```
Thread t = new Thread(new Runnable() { // A anonymous inner class that
implements Runnable
    public void run() { // provide implementation to abstract method run()
        for (int i = 0; i < 100000; i++) {
            if (stop) break;
            tfCount.setText("" + countValue);
            countValue++;
            // suspend itself and yield control to other threads
            try {
                Thread.sleep(10);
            } catch (InterruptedException ex) {}
        }
    }
});
t.start();
```



Un semplice esempio

- Creiamo un oggetto **BankAccount**
- Creiamo due insiemi di thread:
 - *Ogni thread del primo insieme deposita in maniera ripetitiva \$100*
 - *Ogni thread del primo insieme preleva in maniera ripetitiva \$100*
- **deposit** e **withdraw** sono stati modificati per stampare messaggi:

```
public void deposit(double amount)
{
    System.out.print("Depositing " + amount);
    double newBalance = balance + amount;
    System.out.println(", new balance is " + newBalance);
    balance = newBalance;
}
```



Un semplice esempio

- Il risultato dovrebbe essere zero ma alcune volte non lo è
- Normalmente, il programma dovrebbe produrre un output del genere:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
. . .
Withdrawing 100.0, new balance is 0.0
```

- Ma alcune volte vengono stampati messaggi come:

```
Depositing 100.0Withdrawing 100.0, new balance is
100.0,
    new balance is -100.0
```



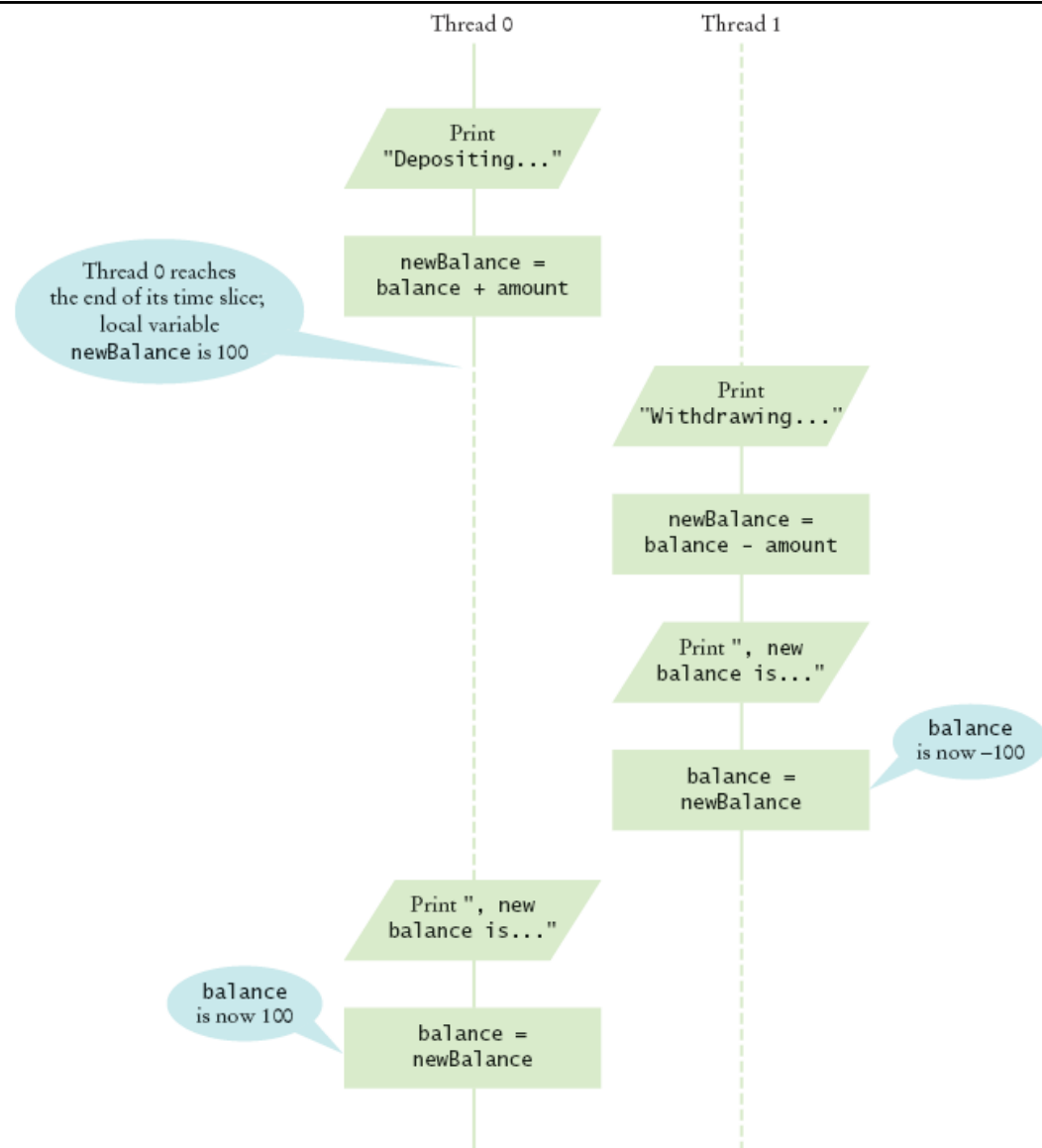
Scenario che produce risultato inferiore a zero

1. Un thread di deposito esegue la linee di codice
`System.out.print("Depositing " + amount);`
`double newBalance = balance + amount;`
`balance` è 0, e la variabile locale `newBalance` è 100
2. Il thread di deposito termina il suo tempo di esecuzione ed un thread di prelievo viene eseguito
3. Il thread di prelievo chiama il metodo `withdraw` che preleva \$100 dalla variabile `balance`; ora vale -100
4. Il thread di prelievo va in sleep
5. Il thread di deposito riprende l'esecuzione da dove era stato interrotto, esegue:

```
System.out.println(", new balance is " + newBalance);  
balance = newBalance;
```

La variabile `balance` vale 100 invece di 0 perchè il metodo deposit usa la vecchia variabile `balance`

Esecuzione



Sincronizzazione

- Accedere concorrentemente ai dati comuni in maniera sicura: **bloccare** un oggetto
- Metodi **synchronized**

```
public class Account {
    private double saldo;

    public Account (double somma_iniz) {
        saldo = somma_iniz;
    }

    public synchronized double getSaldo ( ) {
        return saldo;
    }

    public synchronized void deposito (double somma) {
        saldo += somma;
    }

    public synchronized void trasferisci(Account conto, double somma){
        saldo -= somma;
        conto.deposito(somma);
    }
}
```




Sincronizzazione

```
public class Account {  
    private double saldo;  
  
    public Account (double somma_iniz) {  
        saldo = somma_iniz;  
    }  
  
    public synchronized double getSaldo ( ) {  
        return saldo;  
    }  
  
    public synchronized void deposito (double somma) {  
        saldo += somma;  
    }  
}
```

- Il campo saldo deve essere privato per garantire la consistenza dei dati
- Il costruttore Account non è sincronizzato dato che può essere eseguito solamente alla creazione dell'oggetto
- Sono necessari altri meccanismi di sincronizzazione se si vuole imporre un ordine preciso tra due thread concorrenti nell'accesso ai dati comuni



Sincronizzazione

Istruzioni **synchronized**: bloccare un oggetto durante l'esecuzione di una istruzione

synchronized (expression) statement

l'oggetto risultante viene bloccato durante l'esecuzione di **statement**

```
public static void abs (int[ ] values) {  
    synchronized (values) {  
        for (int i = 0; i < values.length; i++) {  
            if (values[i] < 0)  
                values[i] = - values[i];  
        }  
    }  
}
```

Non è detto che l'oggetto bloccato sia poi effettivamente usato all'interno della istruzione sincronizzata



Classi non sincronizzate in programmi multithreaded

Problema: come fare ad usare in maniera consistente in una applicazione *multithreaded* una classe che non è stata progettata come **synchronized**?

- Soluz.1: Creare una sottoclasse ridefinendo ogni metodo come segue:

```
synchronized metodo (parametri)  {  
    super (parametri) ;  
}
```

- Soluz.2: Usare i metodi della classe **non synchronized** in istruzioni **synchronized**



Comunicazione fra i thread: wait e notify

`wait()` e `notify()` sono metodi della classe `Object` che possono essere chiamati solamente dall'interno di codice sincronizzato

`wait()` rilascia il blocco sull'oggetto e arresta il thread mettendolo in attesa

`notify()` sveglia il thread (ovvero, uno dei thread) in attesa sullo stesso oggetto

`notifyAll()` sveglia tutte i thread in attesa sullo stesso oggetto



wait e notify (esempio)

```
class Risorsse {  
    int quanti;  
    public synchronized void produci ( ) {  
        quanti++;  
        notify( );  
    }  
  
    public synchronized void consuma ( ) {  
        while (quanti == 0)  
            wait ( );  
        quanti--;  
    }  
}
```



Problemi di Deadlocks

Se abbiamo due threads T1 e T2 e due oggetti O1 e O2 su cui effettuare la sincronizzazione, può accadere che:

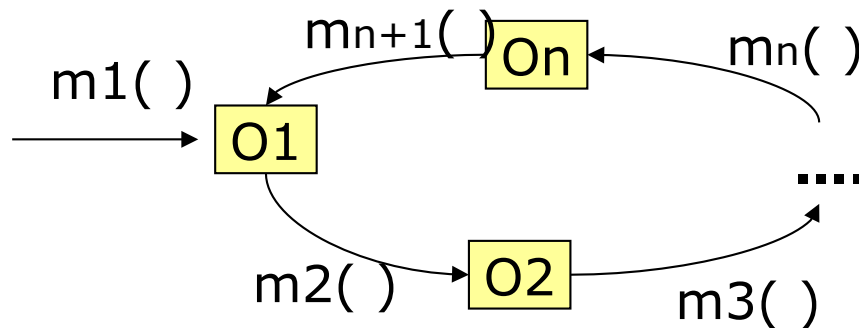
- T1 blocca O1
- T2 blocca O2
- T1 si arresta in attesa dello sblocco di O2
- T2 si arresta in attesa dello sblocco di O1

Nulla può proseguire e l'applicazione è bloccata indefinitamente:

deadlock

Deadlock in generale

L'oggetto O1 esegue un metodo sincronizzato che invoca un metodo sincronizzato su O2, che a sua volta invoca un metodo sincronizzato su On che infine invoca un metodo sincronizzato su O1



JAVA non ha alcun meccanismo per scoprire un deadlock, né potenziale (cioè in fase di compilazione), né effettivo (a run-time)

LA CORRETTEZZA DEI PROGRAMMI CONCORRENTI È RESPONSABILITÀ ESCLUSIVA DEL PROGRAMMATORE

Deadlocks: un esempio

```
public class Trasferimento extends Thread {
    private Account conto1;
    private Account conto2;
    private double importo;

    public Trasferimento (Account c1, Account c2, double quanto) {
        conto1 = c1;
        conto2 = c2;
        importo = quanto;
    }

    public void run ( ) { // trasferisci() è un metodo sincronizzato!!
        conto1.trasferisci(conto2, importo);
    }

    public static void main (String[ ] args) {
        double saldo1 = Integer.parseInt (args[1]);
        double saldo2 = Integer.parseInt (args[3]);
        Account c1 = new Account (args[0], saldo1);
        Account c2 = new Account (args[2], saldo2);
        new Trasferimento(c1, c2, 1000).start();
        new Trasferimento(c2, c1, 2000).start();
    }
}
```


Deadlocks: un esempio

```
public class Account {  
    .....  
    .....  
  
    public synchronized void trasferisci(Account c2, double quanto){  
        synchronized (c2) {  
            c2.deposito(quanto);  
            saldo -= quanto;  
        }  
    }  
}
```



I Lock

- In alternativa all'uso di **synchronized** si possono usare gli oggetti **lock**
- Un oggetto **lock** è usato per controllare thread che manipolano risorse condivise
- In Java: **Lock** è un'interfaccia ed esistono diverse classi che la implementano
 - **ReentrantLock**: è la classe più utilizzata
 - *I lock sono stati introdotti dalla versione 5.0*



I Lock

- Di solito, un oggetto lock viene aggiunto ad una classe i cui metodi condividono risorse, ad esempio:

```
public class BankAccount
{
    public BankAccount()
    {
        balanceChangeLock = new ReentrantLock();
        . . .
    }
    . . .
    private Lock balanceChangeLock;
}
```



I Lock

- Il codice che manipola risorse condivise è racchiuso tra invocazioni di **lock** e **unlock**:

```
balanceChangeLock.lock();
```

```
Code that manipulates the shared resource
```

```
balanceChangeLock.unlock();
```



I Lock

- Se tra una chiamata a **lock** e **unlock** viene lanciata un'eccezione, la chiamata ad **unlock** non viene mai fatta
- Per risolvere questo problema, occorre mettere l'operazione **unlock** nella clausola **finally** :

```
public void deposit(double amount)
{
    balanceChangeLock.lock();
    try
    {
        System.out.print("Depositing " + amount);
        double newBalance = balance + amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
    }
    finally {
        balanceChangeLock.unlock();
    }
}
```