Programmazione concorrente in Java

Seminario di

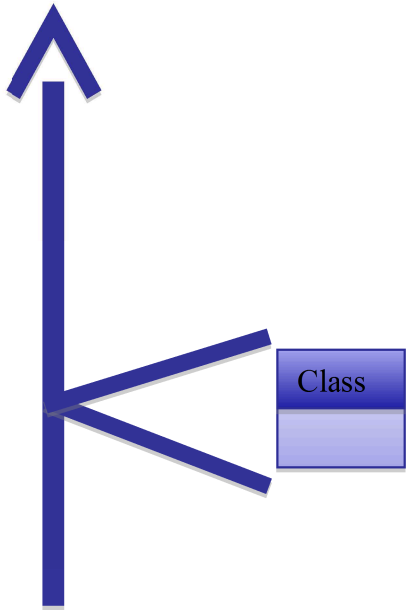
Principi di Sistemi Operativi - LM

Ph.D. Ing. Mariachiara Puviani - mariachiara.puviani@unimore.it
http://mars.ing.unimo.it/wiki/index.php/User:Mariachiara

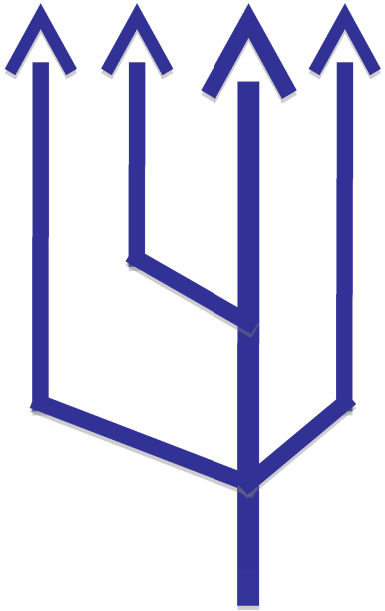
Modena, Lunedì 06 Ottobre 2014

Programmazione Standard e Concorrente

- Svolgimento del programma → metodo main
- Si possono richiamare oggetti da altre classi
- Il main riprende da dove ha chiamato l'oggetto



- Thread paralleli al main
- Processi eseguiti parallelamente, non al suo interno



Processi

- Nei sistemi concorrenti, il termine processo denota la *singola attività eseguita in modo 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

- Paragone con Classi e Oggetti:
  - un **programma** agisce come una *classe*,
  - il **processo** rappresenta una *istanza* in esecuzione del programma

- Processi → Pesanti
  Leggeri (thread)

Un processo (pesante)

- È del tutto **autonomo** rispetto agli altri processi
- Ha un proprio spazio di indirizzamento riservato e non condivide memoria con altri processi: anche le variabili globali sono private al processo (duplicazione dell'area di memoria) e i puntatori puntano comunque a cose diverse (rilocazione dei puntatori)
- Ha un proprio stack riservato (le variabili locali sono private al processo)
- La comunicazione tra processi distinti non può avvenire per condivisione di variabili, ma deve sfruttare appositi meccanismi di sistema
- I processi servono ad avere attività in esecuzione **concorrente MA autonoma**, minimizzando le possibilità di interazione, e quindi di disturbo reciproco
- Modello di processi ad **ambiente locale**

Processi UNIX

- Un nuovo processo viene creato tramite la chiamata di sistema **fork()**:
  - duplica il processo che la esegue (processo padre) creando un identico processo figlio

- Il figlio può:
  - o continuare a eseguire lo stesso codice del padre,
  - o eseguirne un altro tramite la primitiva **exec()**

Thread

Attività autonoma che "vive" all'interno di un processo

Porzione di processo che esegue contemporaneamente insieme ad altri thread dello stesso processo

- Ha un proprio stack riservato (e quindi le sue variabili locali sono private e non condivise da altri thread), propiro program counter, propri registri
- ...ma non ha uno spazio di indirizzamento riservato: tutti i thread appartenenti allo stesso processo condividono il medesimo spazio di indirizzamento → lo stesso codice, gli stessi dati, le stesse risorse, la stessa memoria
- La comunicazione fra thread può avvenire mediante competizione sullo spazio di memoria condiviso: se ci sono variabili globali o riferimenti ad aree di memoria od oggetti comuni, diversi thread possono interagire
- Occorre però disciplinare l'accesso a tale area comune
  - necessità di opportuni **meccanismi di sincronizzazione**
- Modello dei processi/thread ad **ambiente globale**

Comunicazione in OO

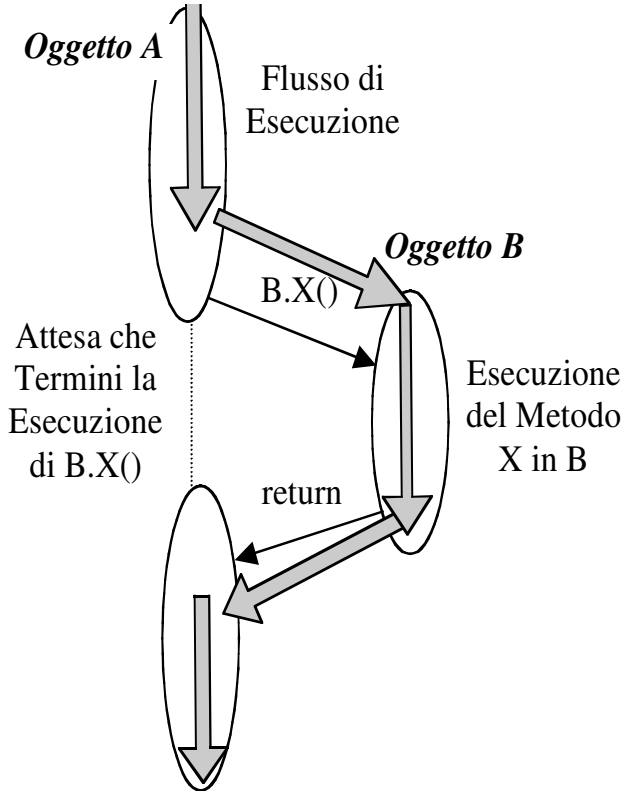
- In un linguaggio ad oggetti, ovviamente, il concetto di variabile globale non ha senso
- Però, più thread possono condividere riferimenti allo stesso oggetto, e interagire tramite tale oggetto

Thread in Java

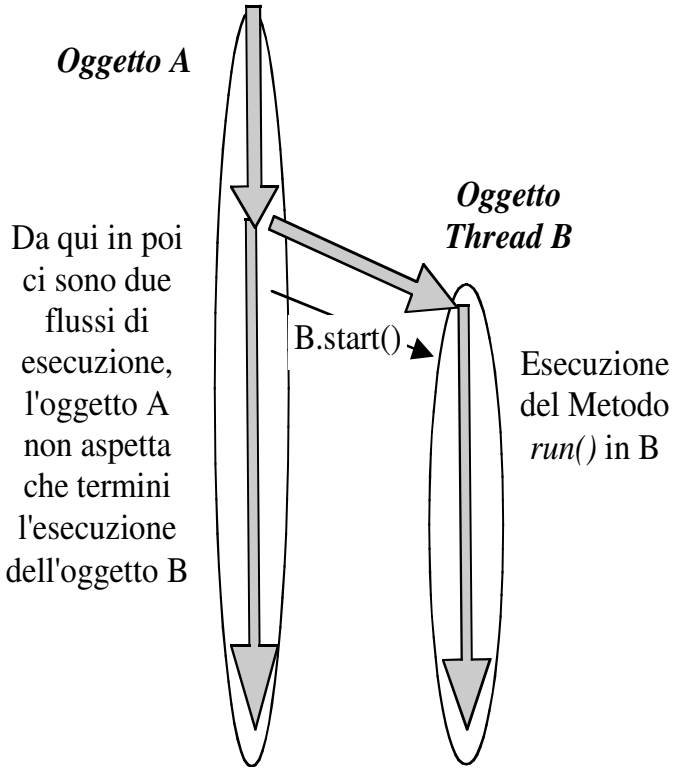
- Come si può realizzare il concetto di Thread in Java?
- Filosofia OO: sono oggetti particolari ai quali si richiede un servizio (chiamato start()) corrispondente al lancio di una attività, di un thread
- MA: non si aspetta che il servizio termini, esso procede in concorrenza a chi lo ha richiesto
- Permettono di eseguire diversi task in modo concorrente (multithreading)

Richiesta di servizio

- Normale richiesta di servizio



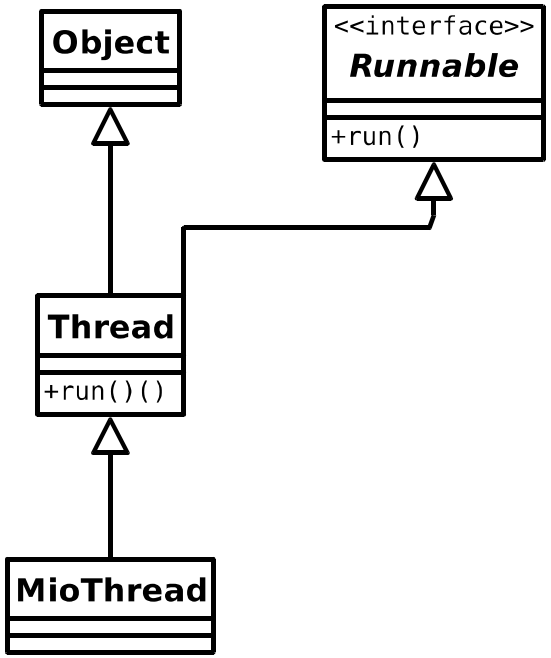
- Richiesta di servizio start() a un thread



La classe Thread

- La **classe Thread** è la base per la gestione dei thread in Java
- Essa comprende metodi per attivare, fermare, sospendere, riprendere, attendere thread, per controllarne la priorità, lo scheduling, etc.
- Creazione di un thread → normale *creazione di un oggetto di classe Thread o derivate*
- Generalmente la classe **java.lang** contiene tutti i metodi per la gestione dei Thread

java.lang.Thread



Definire ed eseguire un thread (A)

- Estendere la classe **java.lang.Thread**
- Il corpo del thread è costituito dal metodo **run()**
- Quindi, per definire una nuova classe di thread si deve:
  - definire una propria sottoclasse di Thread che *ridefinisca* opportunamente il metodo run()
- Per creare un nuovo oggetto thread si dovrà quindi:
  - creare una *istanza* della sottoclasse di Thread
- Per far partire il nuovo thread (cioè l'esecuzione del suo metodo run()) basta invocare su di esso il metodo **start()**

Esempio di thread (A) - 1

```
public class SimpleThread extends Thread
{
  public SimpleThread(String str)
  {
   super(str); //si definisce il nome del thread
  }

  public void run()  //metodo opportunamente ridefinito
  {
    for (int i = 0; i < 10; i++)
    {
     System.out.println(i + " " + getName());
      try {
        sleep((int)(Math.random() * 1000));
      } catch (InterruptedException e)
              {System.out.println(”Eccezione”);}
    }
    System.out.println("DONE! " + getName());
  }
}
```

Esempio di thread (A) - 2


```
public class Esempio1
{
  public static void main (String[] args)
  {
    new SimpleThread("Jamaica").start();
    new SimpleThread("Fiji").start();
  }
}
```

Risultato di 2 esecuzioni

```
sirio$ java Esempio1
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Fiji
2 Jamaica
3 Fiji
4 Fiji
5 Fiji
3 Jamaica
4 Jamaica
6 Fiji
7 Fiji
5 Jamaica
8 Fiji
6 Jamaica
7 Jamaica
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica
```

```
sirio$ java Esempio1
0 Jamaica
0 Fiji
1 Jamaica
1 Fiji
2 Jamaica
3 Jamaica
4 Jamaica
2 Fiji
5 Jamaica
3 Fiji
4 Fiji
5 Fiji
6 Jamaica
7 Jamaica
6 Fiji
8 Jamaica
7 Fiji
9 Jamaica
DONE! Jamaica
8 Fiji
9 Fiji
DONE! Fiji
```

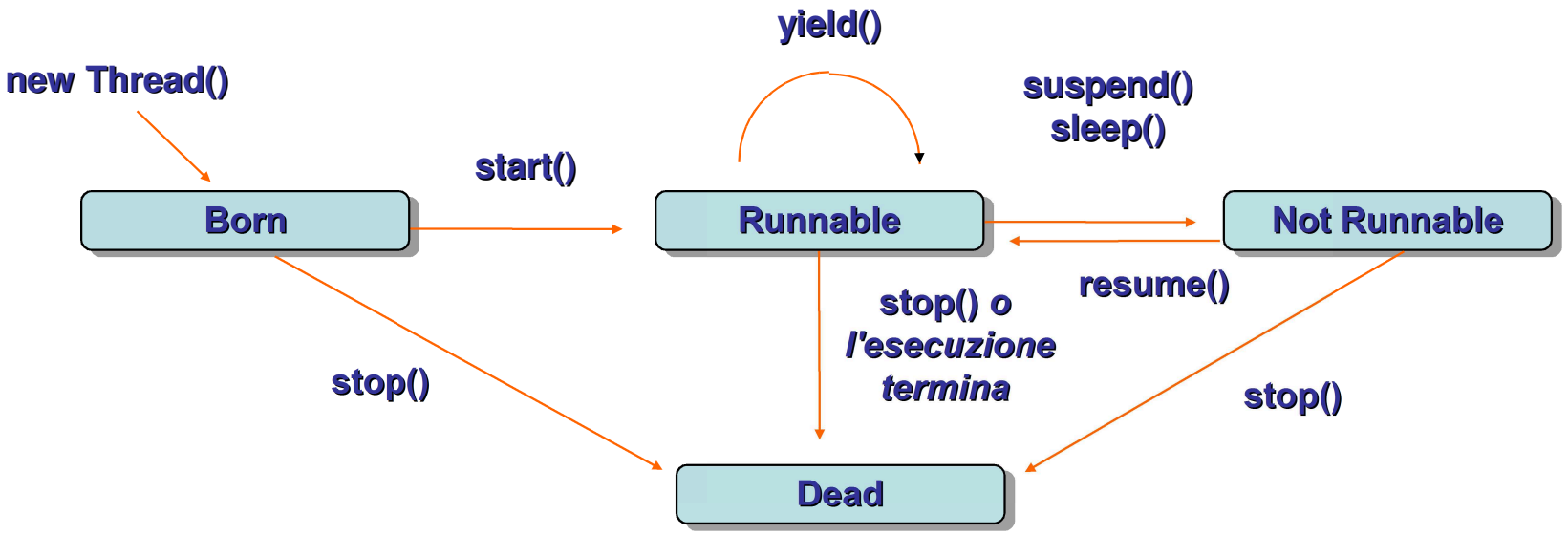
Definire ed eseguire un thread (B)

- Il corpo del thread è costituito dal metodo **run()**
- Per definire una nuova classe di thread si deve:
  - definire una propria classe che *implementa* l'interfaccia **Runnable**
  - ridefinire il metodo run() che è dichiarato anche nell'interfaccia Runnable
- Per creare un nuovo oggetto thread si dovrà quindi:
  - creare una *istanza* della sottoclasse di Thread alla quale si passa l'elemento Runnable creato
- Per far partire il nuovo thread (cioè l'esecuzione del suo metodo run()) basta invocare su di esso il metodo **start()**

Esempio di thread (B)


```
public class SimpleThread implements Runnable
{
 public static void main (String[] args)
 {
   System.out.println("Thread corrente" + thread.currentThread());
   SimpleThread st = new SimpleThread();
   Thread t = new Thread(st);
   t.start();
 }
  public void run()  //metodo opportunamente ridefinito
  {
    // write your method here
  }
}
```

Stato di un thread



Metodi

Metodi di Thread

- public void start()
- public void run()
- public void stop()
- public String getName()
- public void setName(String)
- public void sleep(long)
- public void suspend()
- public void resume()
- public void yield()
- public void join()
- public void setPriority(int num)
- public int getPriority()

start(), run() e stop()

- start() inizia l'esecuzione dopo la necessaria inizializzazione
  - invoca run() (o meglio la JVM invoca run() in un altro thread Java, facendo partire effettivamente la vita del thread)
- stop() termina l'esecuzione del thread (lancia una eccezione ThreadDeath() alla vittima!!) se si ha il privilegio per farlo.

getName() e setName()

- Gestione del nome del thread
- Ogni Thread ha sempre associato un nome (non necessariamente univoco!), corrispondente a una stringa, che o viene assegnato di default dal sistema o viene assegnato dall'utente in fase di inizializzazione
- Per ottenere il nome del thread corrente:

```
Thread.currentThread().getName();
```

```
setName(String newName)
```

Un metodo alternativo per settare i nome di un thred è invocare su di esso il metodo super() – v. esempio precedente) -

sleep(), suspend() e resume()

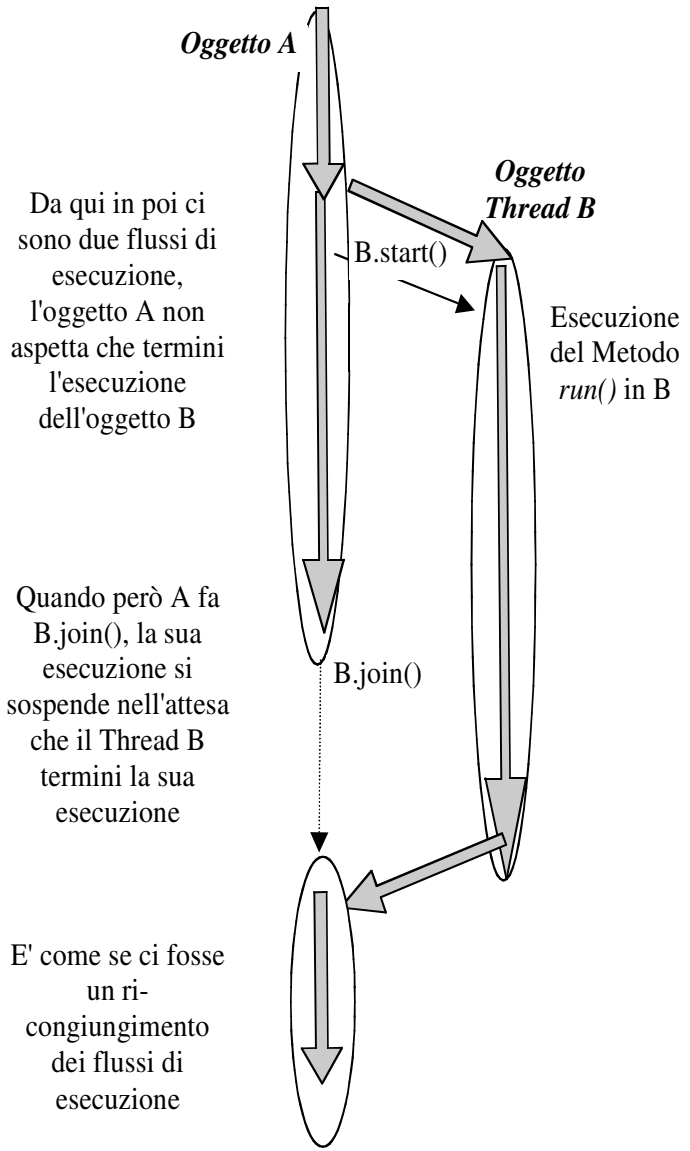
- I primi due metodi portano il thread nello stato "not runnable"
- sleep() per un periodo di tempo prefissato
  - Non utilizza cicli del processore
  - è un metodo statico e mette in pausa il thread corrente
  - mentre un thread è in sleep può essere interrotto da un altro thread:
    - viene sollevata un'eccezione InterruptedException → sleep() va eseguito in un blocco try catch (oppure il metodo che lo esegue deve dichiarare di sollevare tale eccezione)
- suspend() sospende un thread
- resume() risveglia un thread sospeso da suspend(), portandolo nello stato runnable

yield()

- Cede il controllo del processore → Permette ad un thread di lasciare **volontariamente** il processore ad un altro thread
- E' come una sorta di "Preemption" volontaria → re-scheduling

join()

- Si può decidere di aspettare che termini l'esecuzione di un thread → viene invocato su un thread specifico
- Per fare questo, bisogna invocare l'operazione **join()** sul thread che si intende aspettare → rimane bloccato in attesa della terminazione dell'altro thread
- Simile alla wait() di UNIX



Esempio di join()

- Il programma principale genera un thread e gli passa un riferimento ad un oggetto StringBuffer
- Il thread scrive nello StringBuffer la data corrente
- Il programma principale aspetta che il thread finisca e poi stampa il valore dello StringBuffer

Esempio di join (2)


```
import java.util.Date;

public class StringThread extends Thread
{
  private StringBuffer s;

  public StringThread(StringBuffer s)
  {
       this.s = s;
   }
  public void run()
  {
   s = s.append(new Date());
  }
}
```

Esempio di join (3)

```
public class Esempio3
{
  public static void main(String args[])
  {
    StringBuffer s = new StringBuffer();
    StringThread t = new StringThread(s);
    t.start();
    try {
       t.join(); }
    catch(InterruptedException e) {}
    System.out.println("s vale " + s);
  }
}
```

Note sull'esempio

- L'attesa di join() potrebbe essere interrotta da un'eccezione
  - necessità di catturare l'eccezione tramite il costrutto **try/catch** (si veda più avanti)
  - necessità analoghe per il metodo sleep()
- Se non avessimo usato join(), il risultato dell'esecuzione sarebbe stato probabilmente:

```
c:\myprog\java>java UsaStringThread
                  s vale
```

- Ma dipende dallo scheduler!

Vita di un thread

- Un thread continua la propria esecuzione:
  - fino alla sua terminazione naturale, o
  - fino a che non viene espressamente fermato da un altro thread che invochi su di esso il metodo stop()

Sospensione di un thread

- Un thread può anche essere sospeso temporaneamente dalla sua esecuzione:
  - perché un operazione blocca l'esecuzione, come una operazione di input che implica di attendere dei dati; il Thread si blocca finchè i dati non sono disponibili
  - perché invoca la funzione Thread.sleep()
  - da un altro thread che invochi su di esso il metodo suspend(), per riprendere poi la propria esecuzione quando qualcuno invoca su di esso il metodo resume()

Problemi

- stop(), suspend() e resume() sono deprecati in Java2, in quanto non sicuri!

- Stop():
  - Il thread terminato non ha il tempo di rilasciare eventuali risorse
  - Si possono così avere dati corrotti
  - I lock acquisiti non vengono rilasciati → deadlock

- Questi tre metodi agiscono in modo brutale su un thread
  - che non può "difendersi" in nessun modo

    → possibili rischi!

Possibili rischi

- Se il thread sospeso / fermato stava facendo un'operazione critica, essa rimane a metà e l'applicazione viene a trovarsi in uno *stato inconsistente*

- Se il thread sospeso / fermato aveva acquisito una risorsa (è nel monitor di questa risorsa!), essa rimane bloccata da quel thread e nessun altro può conquistarla

➔ Se un altro thread si pone in attesa di tale risorsa, si crea una situazione di **deadlock**

Soluzione

- La soluzione suggerita da Java 2 consiste nell'adottare un diverso approccio:
  - non agire d'imperio su un thread
  - ma segnalargli l'opportunità di sospendersi / terminare
  - lasciando al thread stesso il compito di sospendersi / terminare, in modo che possa farlo in un momento "non critico"
  - la funzione yield() permette a un thread di cedere l'esecuzione ad un altro

Scheduling

- In un *sistema monoprocessore* i thread devono condividere l'esecuzione su un unico processore
- In un certo istante, ci sarà un solo thread realmente in esecuzione, e altri in attesa di *acquisire il controllo* del processore

Rilascio del processore

- Quando si invocano dei metodi di controllo, come suspend(), resume(), stop(), che permettono di gestire l'uso del processore e fanno in modo che un thread, bloccandosi o sospendendosi, ceda il processore

- Quando un processo invoca una operazione che implica l'attesa di qualche evento, come un'operazione di input, e quindi essendo in attesa non ha bisogno del processore

Altrimenti…

- ***scheduling nonpreemptive*** se il controllo del microprocessore non viene mai forzatamente tolto al thread corrente

- ***scheduling preemptive*** se il controllo del microprocessore può forzatamente essere tolto al thread corrente in un qualsiasi momento dell'esecuzione

Attenzione!!!

- Java non precisa quale tipo di scheduling debba essere adottato dalla macchina virtuale!
- Dipende dal Sistema Operativo!
- Per determinare se il SO è preemptive o meno, è possibile eseguire un semplice test

Test


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

class Esempio4 {
  public static void main(String args[]){
   System.out.println("Main - inizio");
   MyRun r = new MyRun();
   new Thread(r, "cip ").start();
   new Thread(r, "ciop").start();
   // NB: se è non-preemptive, va solo cip
   // se è preemptive, vanno entrambi
  }
}
```

Priorità

- Ogni thread ha una priorità
- intero compreso tra MIN_PRIORITY (1) e MAX_PRIORITY (10)
- Di default, NORM_PRIORITY (5)
- Metodi **getPriority()** e **setPriority()** della classe Thread

Sincronizzazione

- Quando due o più thread eseguono concorrentemente, è in generale impossibile prevedere l'ordine in cui le loro istruzioni verranno eseguite
- Problemi nel caso in cui i thread invocano metodi sullo stesso oggetto di cui condividono il riferimento
- SONO POSSIBILI INCONSISTENZE!

Esempio di sincronizzazione


```
public class Contatore
{
  private int valore;

  public Contatore(int val) { valore = val; }

  public int conta()
  {
        int tmp = valore + 1;
        valore = tmp;
        return tmp;
  }
}
```

Esempio di sincronizzazione (2)

```
public class ThreadCheConta extends Thread
{
  private Contatore contatore;

  public ThreadCheConta(String str, Contatore cont)
  {
       super(str);
       contatore = cont;
   }

  public void run()
  {
    for (int i = 0; i < 5; i++)
       System.out.println(getName() + " ha incrementato a "
   + contatore.conta());
  }
}
```

Esempio di sincronizzazione (3)


```
public class Esempio5
{
  public static void main(String[] args)
  {
    Contatore c = new Contatore(0);
    new ThreadCheConta("T1", c).start();
    new ThreadCheConta("T2", c).start();
  }
}
```

Risultato

```
c:\myprog\java>java Esempio5
T1 ha incrementato a 1
T2 ha incrementato a 2
T1 ha incrementato a 3
T2 ha incrementato a 4
T2 ha incrementato a 5
T2 ha incrementato a 6
T1 ha incrementato a 7
T2 ha incrementato a 8
T1 ha incrementato a 8
T1 ha incrementato a 9
```

Flusso di esecuzione

- Thread T1

```
contatore.conta()

tmp = valore + 1;
// tmp = 7 + 1


valore = tmp;
// valore = 8
```

- Thread T2

```
contatore.conta()


tmp = valore + 1;
// tmp = 7 + 1


valore = tmp;
// valore = 8
```

Errore!

- I due thread eseguono concorrentemente e leggono *lo stesso* valore, poi lo modificano

LETTURA SPORCA
(dirty read)

Sezione critica

- Le sezioni di codice in cui si accede a risorse e oggetti comuni (condivise) sono *critiche*
- Per evitare inconsistenze, è necessario assicurare che, prima di entrarvi, i thread si **sincronizzino**, in modo da garantire che solo un thread per volta possa eseguire la sezione critica
- Nell'esempio precedente, bisogna evitare che due thread eseguano in concorrenza il metodo *conta*

Costrutto synchronized

- In Java si può garantire l'accesso in **mutua esclusione** a una istanza tramite il costrutto dei mutex:
- Si protegge il metodo o la sezione di codice critica tramite la keyword **synchronized**:
  - a livello di metodo
  - a livello di blocco di codice

- È come avere:
  - **Un monitor per ogni oggetto**
  - **Una (sola) variabile condition usata implicitamente**

- Invocare un metodo sincronizzato o eseguire un blocco sincronizzato corrisponde a entrare in una procedure entry di un monitor concurrent pascal

Metodi

1. Metodo sincronizzato (sincronizzazione implicita):

```
public synchronized void conta() {...}
```

- Solo un thread alla volta può eseguire questo metodo sullo stesso oggetto

2. Blocco di codice sincronizzato (sincronizzazione esplicita):

```
synchronized(object) {...}
```

- Solo un thread alla volta può eseguire la parte di codice protetta sull'oggetto object (che può essere this)

Modifiche all'esempio

- Il metodo *conta* deve essere synchronized
- Mentre il flusso di esecuzione di un thread esegue tale metodo, è garantito che nessun'altro thread potrà eseguirlo
- Se T1 sta già eseguendo il metodo, quando T2 tenta di eseguirlo viene sospeso, in attesa che T1 termini
- Quando T1 termina, T2 riprende l'esecuzione a può eseguire il metodo *conta*

Contatore corretto


```
public class ContatoreCorretto
{
  private int valore;
  public ContatoreCorretto(int val)
  {
       valore = val;
  }

  public synchronized int conta()
  {
       int tmp = valore + 1;
       valore = tmp;
       return tmp;
  }
}
```

wait() e notify()

- Java mette a disposizione due metodi (della classe Object) per la sospensione e il risveglio dei thread

- **wait()** (→ *wait* in concurrent pascal)
  - Sospende il thread che lo invoca su una coda associata all'oggetto sul quale il metodo è invocato
  - Il thread che lo invoca rilascia il mutex associato all'istanza

- **notify()** (→ *signal* in concurrent pascal) e **notifyAll()**
  - Risveglia il primo thread (o tutti se notifyAll()) sospeso sulla coda dell'oggetto su cui viene invocato il metodo
  - Il metodo che lo invoca deve aver acquisito il mutex→ il thread attende il rilascio

- **NB: wait() e notify() possono essere invocati soltanto all'interno di un metodo synchronized**

Esempio produttore – consumatore

- Contenitore
  - ha capacità unitaria
- Produttore
  - inserisce un intero alla volta nel contenitore
  - solo se il contenitore non è già pieno!
- Consumatore
  - estrae un intero dal contenitore
  - solo se c'è!

Esempio P-C (1)


```
public class Produttore extends Thread
{
  private Contenitore contenitore;
  private int id;

  public Produttore(Contenitore c, int id) {
    contenitore = c;
    this.id = id;
  }

  public void run()
  {
    for (int i = 0; i < 5; i++)
    {
      contenitore.DEPOSITA(i);
      System.out.println("Pr#"+ id +" mette: " + i);
    }
  }
}
```

Esempio P-C (2)

```
public class Consumatore extends Thread
{
  private Contenitore contenitore;
  private int id;

  public Consumatore(Contenitore c, int id) {
    contenitore = c;
    this.id = id;
  }

  public void run()
  {
    int valore = 0;
    for (int i = 0; i < 5; i++)
    {
      valore = contenitore.PRELEVA();
      System.out.println("Co#" + id + " prende: " + valore);
    }
  }
}
```

Esempio P-C (3)

```
public class Contenitore
{
     // un monitor è associato ad ogni istanza
     private int buff;
     // un buffer di dimensione unitaria
     private boolean disponibile = false;

public synchronized void DEPOSITA(int valore)
{
     while (disponibile == true)
     {
          try { wait(); }
          catch(InterruptedException e)
                  {System.out.println("Eccezione!");}
     }

     buff = valore;
     disponibile = true;
     notifyAll();
}
```

Esempio P-C (4)

```
public synchronized int PRELEVA()
    {
        while (disponibile == false)
        {
           try { wait(); }
          catch(InterruptedException e)
                        {System.out.println("Eccezione!");}
        }
        disponibile = false;
        int tmp = buff;
        notifyAll();
        return tmp;
    }
}
```

Esempio P-C (5)


```
public class Esempio7
{
  public static void main(String args[])
  {
    Contenitore contenitore = new Contenitore();
    new Produttore(contenitore, 1).start();
    new Consumatore(contenitore, 1).start();
  }
}
```

Esempio P-C (6)


```
Risultato dell'esecuzione
sirio$ java Esempio8
Pr#1 mette: 0
Co#1 prende: 0
Pr#1 mette: 1
Co#1 prende: 1
Pr#1 mette: 2
Co#1 prende: 2
Pr#1 mette: 3
Co#1 prende: 3
Pr#1 mette: 4
Co#1 prende: 4
```

Note su wait() e notify()

- Una **notify** effettuata su un oggetto su cui nessun thread è in wait viene persa (non è un semaforo)
- Usciti dallo stato di **wait** sarebbe meglio ricontrollare le condizioni
- **notifyAll** risveglia tutti i therad in attesa, ma solo uno alla volta riprenderà il lock
- Prima che un thread in attesa riprenda l'esecuzione, il thread che ha notificato deve rilasciare il monitor (uscire dal blocco synchronised)

La gestione degli errori

- In Java la gestione degli errori avviene attraverso le *eccezioni*
- **Eccezione**: evento che capita durante l'esecuzione e sconvolge il normale flusso di esecuzione
- Esistono classi per rappresentare le eccezioni

Sollevare un'eccezione

- Ogni metodo può sollevare una o più **eccezioni**
  - Costrutto **throws**: indica quali eccezioni può sollevare
    - `nomeMetodo(…) throws tipoEccezione`
  - Costrutto **throw**: solleva un'eccezione
    - `throw new tipoEccezione();`

Catturare un'eccezione

- Un'eccezione sollevata viene catturata e gestita
  - Costrutto **try-catch**

```
try
{
... // blocco di istruzioni
}
catch (tipo_di_eccezione_da_catturare variabile_eccezione)
{
... // istruzioni da eseguire in caso di eccezione
}
```

Monitor ed eccezioni

- Cosa succede se in un blocco synchronized viene sollevata una eccezione?
  Java garantisce il rilascio del lock "sicuro"

1. Metodo synchronized:
   La JVM si occupa di rilasciare il lock di mutua esclusione sul monitor dell'oggetto
2. Blocco synchronized
   Il compilatore inserisce l'istruzione monitorexit in un blocco finally

Deamon therad

- Sono eseguiti in background (nei momenti in cui il processore non viene utilizzato)
  - Es: garbage collector
- Il programma può terminare ugualmente
  - Es: quando ci sono in esecuzione sono deamon thread
- Devono essere settati come Deamon prima di chiamare start
  - setDeamon(true)

The end

- Questo seminario (versione aggiornata 2014)

  http://mars.ing.unimo.it/wiki/index.php/Principi_di_Sistemi_Operativi_-_LM

- Per ulteriori informazioni e dubbi:

  http://mars.ing.unimo.it/wiki/index.php/User:Mariachiara