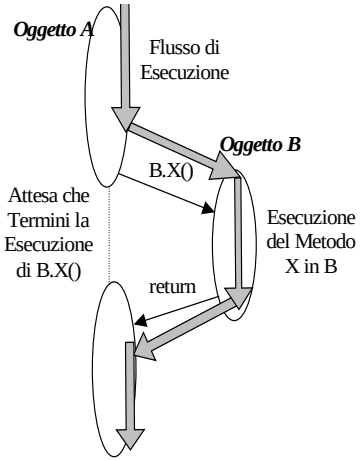
THREAD IN JAVA

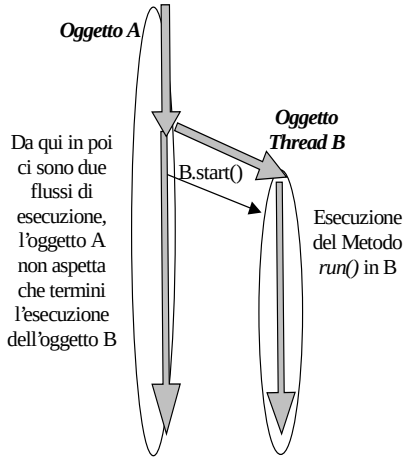
Come si può realizzare il concetto di Thread in Java?

NEL MODO PIU' NATURALE! 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!

Normale Richiesta di Servizio

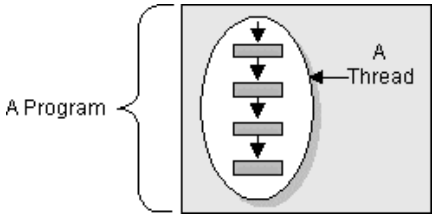


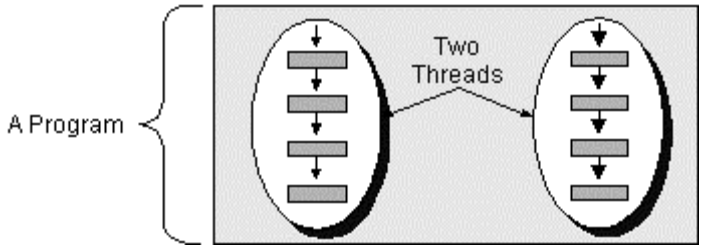
Richiesta di Servizio start() a un Thread



Thread

Un ***thread*** (*lightweight process*) è un singolo flusso sequenziale di controllo all'interno di un processo





Un ***thread***:

- esegue all'interno del contesto di esecuzione di un **unico processo/programma**
- **NON** ha uno spazio di indirizzamento riservato: tutti i thread appartenenti allo stesso processo condividono lo **stesso spazio di indirizzamento**
- ha *execution stack* e *program counter* **privati**

Java Thread

Due modalità per implementare ***thread*** in Java:

1. come sottoclasse della classe **`Thread`**
2. come classe che implementa l'interfaccia **`Runnable`**

1) come sottoclasse della classe **`Thread`**

- **`Thread`** possiede un metodo **`run()`** che la sottoclasse deve ridefinire
- si crea un'istanza della sottoclasse tramite **`new`**
- si esegue un thread chiamando il metodo **`start()`** che a sua volta richiama il metodo **`run()`**

Esempio di classe `Simplethread` che è ***sottoclasse*** di `Thread` (modalità 1):

```
public class SimpleThread extends Thread {

   public SimpleThread(String str) {
      super(str);
      }

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


public class TwoThreadsTest {

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

E se occorre definire thread che non siano **necessariamente** sottoclassi di Thread?

2) come classe che implementa l'interfaccia **`runnable`**

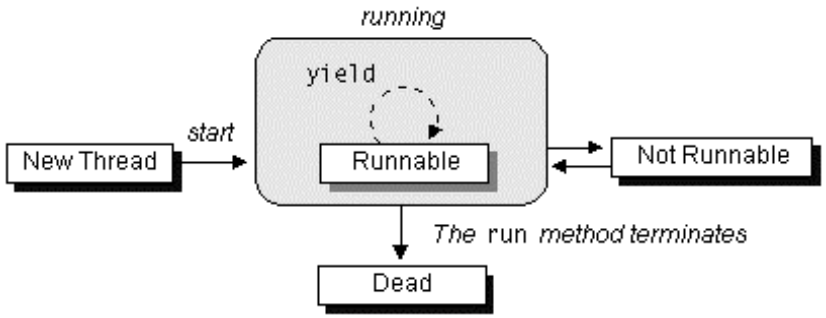
- implementare il metodo **`run()`** nella classe
- creare un'istanza della classe tramite **`new`**
- creare un'istanza della classe **`Thread`** con un'altra **`new`**, passando come parametro l'istanza della classe che si è creata
- invocare il metodo **`start()`** sul thread creato, producendo la chiamata al suo metodo **`run()`**

Interfaccia `Runnable`:
maggiore **flessibilità** derivante dal poter essere sottoclasse di qualsiasi altra classe

Esempio di classe `EsempioRunnable` che ***implementa l'interfaccia `Runnable`*** ed è sottoclasse di `MiaClasse` (modalità 2):

```
class EsempioRunnable extends MiaClasse
   implements Runnable {

   // non e' sottoclasse di Thread

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

public class Esempio {

   public static void main(String args[]){

      EsempioRunnable e =
           new EsempioRunnable();
      Thread t = new Thread (e);
      t.start();
   }
}
```

Il ciclo di vita di un thread



- **Creato**
  subito dopo l'istruzione **new**
  le variabili sono state allocate e inizializzate; il thread è in attesa di passare allo stato di eseguibile

- **Runnable**
  thread è in esecuzione, o in coda d'attesa per ottenere l'utilizzo della CPU

- **Not Runnable**
  il thread non può essere messo in esecuzione dallo scheduler. Entra in questo stato quando in **attesa** di un'operazione di I/O, o dopo l'invocazione dei metodi **`suspend()`**, **`wait()`**, **`sleep()`**

- **Dead**
  al termine "naturale" della sua esecuzione o dopo l'invocazione del suo metodo **`stop()`** da parte di un altro thread

Metodi per il controllo di thread

`start()` fa ***partire*** l'esecuzione di un thread. La macchina virtuale Java invoca il metodo `run()` del thread appena creato

`stop()` ***forza*** la ***terminazione*** dell'esecuzione di un thread. Tutte le risorse utilizzate dal thread vengono immediatamente ***liberate*** (lock inclusi), come effetto della propagazione dell'eccezione `ThreadDeath`

`suspend()` ***blocca*** l'esecuzione di un thread in attesa di una successiva operazione di ***resume***. Non libera le risorse impegnate dal thread (possibilità di ***deadlock***)

`resume()` ***riprende*** l'esecuzione di un thread precedentemente ***sospeso***. Se il thread riattivato ha una priorità maggiore di quello correntemente in esecuzione, avrà subito accesso alla CPU, altrimenti andrà in coda d'attesa

`sleep(long t)` blocca per un ***tempo specificato*** (`time`) l'esecuzione di un thread. Nessun *lock* in possesso del thread viene rilasciato.

`join()` ***blocca*** il thread chiamante in attesa della ***terminazione*** del thread di cui si invoca il metodo. Anche con ***timeout***

`yield()` ***sospende*** l'esecuzione del thread invocante, lasciando il controllo della CPU agli altri thread in ***coda d'attesa***

I metodi precedenti interagiscono ***ovviamente*** con il ***gestore della sicurezza*** della macchina virtuale Java (`SecurityManager`, `checkAccess()`, `checkPermission()`)

Altri metodi fondamentali per le operazioni di ***sincronizzazione*** fra thread Java:
`wait()`, **`notify()`**, **`notifyAll()`**

(vedi lucido 11 e seguenti)

Il problema di `stop()`e `suspend()`

`stop()` e `suspend()` rappresentano azioni "brutali" sul ciclo di vita di un thread
=> rischio di determinare situazioni di blocco critico (***deadlock***)

Infatti:

- se il ***thread sospeso*** aveva acquisito una ***risorsa*** in maniera ***esclusiva***, tale risorsa rimane ***bloccata*** e non è utilizzabile da altri, perché il thread sospeso non ha avuto modo di rilasciare il *lock* su di essa
- se il ***thread interrotto*** stava compiendo un insieme di operazioni su risorse comuni, da eseguirsi idealmente in maniera ***atomica***, l'interruzione può condurre ad uno ***stato inconsistente*** del sistema

➔ JDK 1.2, pur supportandoli ancora per ragioni di *back-compatibility*, ***sconsiglia*** l'utilizzo dei metodi `stop()`, `suspend()` e `resume()` (***metodi deprecated***)

Si consiglia invece di realizzare tutte le azioni di ***controllo*** e ***sincronizzazione*** fra thread tramite i metodi **`wait()`** e **`notify()`** su variabili condizione (astrazione di ***monitor***)

Priorità dei thread in Java

Scheduling: esecuzione di una molteplicità di thread su una singola CPU, in un qualche ordine

Macchina virtuale Java (JVM)

Fixed Priority Scheduling

algoritmo di scheduling molto semplice e deterministico

- JVM sceglie il thread in stato ***runnable*** con ***priorità*** più ***alta***
- Se più thread in attesa di eseguire hanno ***uguale priorità***, la scelta della JVM avviene con una modalità di tipo ***round-robin***.

La classe `Thread` fornisce i metodi:

- `setPriority(int num)`
- `getPriority()`

con valori di `num` compresi fra `MIN_PRIORITY` e `MAX_PRIORITY` (costanti definite anch'esse nella classe `Thread`)

Il ***thread*** messo ***in esecuzione*** dallo scheduler viene ***interrotto*** se e solo se:

- un thread con priorità ***più alta*** diventa ***runnable***;
- il metodo `run` ***termina l'esecuzione*** o il thread esegue un `yield`;
- il ***quanto*** di tempo assegnato si è ***esaurito*** (solo su sistemi che supportano *time-slicing*, come *Windows95/NT*)

```
public void run() {
   while (tick < 200000) {
         tick++;
         if ((tick % 50000) == 0)
            System.out.println("Thread #" +
              num + ", tick = " + tick);
   }
}
```

Time-Sliced System	***Non Time-Sliced System***
Thread #1, tick = 50000	Thread #0, tick = 50000
Thread #0, tick = 50000	Thread #0, tick = 100000
Thread #0, tick = 100000	Thread #0, tick = 150000
Thread #1, tick = 100000	Thread #0, tick = 200000
Thread #1, tick = 150000	Thread #1, tick = 50000
Thread #1, tick = 200000	Thread #1, tick = 100000
Thread #0, tick = 150000	Thread #1, tick = 150000
Thread #0, tick = 200000	Thread #1, tick = 200000

SINCRONIZZAZIONE di THREADS

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:
Oggetto `conto_corrente`
con metodo `versamento(importo)`

```
public void versamento(int importo)
{ int nuovo_totale; //variabile locale

nuovo_totale = totale_conto + importo - tasse;
//totale_conto è una variabile dell'oggetto
//e indica i soldi totali sul conto
//l'istruzione sopra calcola in nuovo totale del
// conto corrente mettendo il risultato nella
//variabile locale

totale_conto  = nuovo_totale;
// metto il totale calcolato nella variabile
//dell'oggetto che memorizza il conto totale
}
```

Supponiamo che due thread abbiano entrambi un riferimento all'oggetto invochino separatamente il metodo versamento per fare ognuno un versamento sul conto….

ESEMPIO DI INCONSISTENZA

Supponiamo che l'oggetto `conto_corrente` abbia nella sua variabile `totale_conto` il valore **2000**. Ci si aspetta che se qualcuno deposita **500** e qualcun altro deposita **1000**, e supponendo che le tasse per ogni versamento siano **100**, alla fine la variabile totale_conto valga **3300**.
Supponiamo che questi due depositi vengano fatti in concorrenza da due thread diversi e che durante l'esecuzione i thread si alternino sul processore, come segue

Thread A	Thread B
conto_corrente.versamento(1000) // il thread invoca il metodo e il flusso di esecuzione fa a eseguire le istruzioni di tale metodo	
	conto_corrente.versamento(500) // il thread invoca il metodo e il flusso di esecuzione fa a eseguire le istruzioni di tale metodo
nuovo_totale= totale_conto+importo-tasse; //nuovo_totale = 2000+1000-100	
	nuovo_totale= totale_conto+importo-tasse; //nuovo_totale = 2000+500-100
totale_conto =nuovo_totale; //totale_conto vale 2900	
	totale_conto = nuovo_totale; //totale_conto vale 2400

Alla fine, **`totale_conto`** ***vale 2400!!!!***

Sincronizzazione di thread

Differenti thread che fanno parte della stessa applicazione Java condividono lo ***stesso spazio*** di memoria

➔ è possibile che ***più thread*** accedano ***contemporaneamente*** allo stesso metodo o alla stessa sezione di codice di un oggetto

Servono meccanismi di sincronizzazione

JVM supporta la definizione di ***monitor*** per la sincronizzazione nell'accesso a risorse tramite la keyword **`synchronized`**

`synchronized` su:

- singolo metodo, o
- blocco di istruzioni

In pratica:

- a ogni oggetto Java è automaticamente associato un ***lock***
- per accedere a un metodo o una sezione `synchronized`, un thread deve prima ***acquisire il lock*** dell'oggetto
- il ***lock*** è automaticamente ***rilasciato*** quando il thread esce dalla sezione `synchronized`, o se viene interrotto da un'eccezione
- un thread che non riesce ad acquisire un *lock* ***rimane sospeso*** sulla richiesta della risorsa fino a che il *lock* non è disponibile

NOTA:

ad ogni oggetto contenente metodi o blocchi `synchronized` viene assegnata ***una sola variabile condizione***

➔ Due thread non possono accedere contemporaneamente a ***due sezioni `synchronized` diverse di uno stesso oggetto***

L'esistenza di *una sola variabile condizione* **per ogni oggetto rende il modello Java** *meno espressivo* **di un *vero monitor*, che presuppone la possibilità di definire più sezioni critiche per uno stesso oggetto**

ESEMPIO - Uno stack sincronizzato

```
class EmptyStackException extends Exception {
  EmptyStackException() {}
  EmptyStackException(String s) { super(s); }
}
```

```
class Stack {
  Object val[];
  int sp = 0;

  Stack(int max) { val = new Object[max]; }

  public synchronized void push(Object e) {
    val[sp++]=e;
  }

  public synchronized Object pop()
        throws EmptyStackException {
    if (sp>0) return val[--sp];
    else throw new EmptyStackException();
  }

  public boolean isEmpty() { return (sp==0); }

  public void print() {
    System.out.print("Stack content: [");
    for(int i=0; i<sp-1; i++)
        System.out.print(val[i] + ", ");
    if (sp>0) System.out.print(val[sp-1]);
    System.out.print("]");
  }
}
```

ESEMPIO - Uso dello stack

Questo codice riunisce in sé *tre attività diverse*: il pusher (che inserisce nello stack dato 20 oggetti Integer), il popper (che li estrae), e il printer (che visualizza lo stato dello stack).

Ogni istanza di `MyBody9` si comporterà quindi o come pusher, o come popper, o come printer, *a seconda del nome che le viene passato all'atto della costruzione*.

```
class MyBody9 implements Runnable {
  String chiSonoIo;
  static Stack s = new Stack(100); // condiviso
  MyBody9(String name){ chiSonoIo = name; }
  public void run(){
    if (chiSonoIo.equals("pusher")) {
      for(int i=0; i<20; i++) {
        System.out.println(
            Thread.currentThread().getName()
            + " pushing " + i);
        s.push( new Integer(i) );
      } } else
    if (chiSonoIo.equals("popper")) {
      try { Thread.sleep(500); }
      catch (InterruptedException ee) {}
      try {
         while(!(s.isEmpty()))
          System.out.println(
            Thread.currentThread().getName()
              + " popping " + s.pop());
      } catch (EmptyStackException e) {
          System.out.println(
            Thread.currentThread().getName()
            + " tried to pop an empty stack"); }
    } else /* sono il printer */
      s.print();
}}
```

ESEMPIO - Uso dello stack

```
public class Esempio9 {
  public static void main(String args[]){
    // creo tre istanze configurate diversamente
    MyBody9 b1 = new MyBody9("pusher");
    MyBody9 b2 = new MyBody9("popper");
    MyBody9 b3 = new MyBody9("printer");
    // creo tre thread, uno per task da eseguire
    Thread t1 = new Thread(b1, "Produttore");
    Thread t2 = new Thread(b2, "Consumatore");
    Thread t3 = new Thread(b3, "Visualizzatore");
    // ora attivo Produttore e Consumatore…
    t2.start();
    t1.start();
    // …e aspetto che finiscano entrambi
    try { t1.join(); }
    catch(InterruptedException e1) {}
    try { t2.join(); }
    catch(InterruptedException e2) {}
    // alla fine attivo il Visualizzatore
    t3.start();
  }
}
```

NOTA:

- a rigore, esiste ancora un rischio di inconsistenza *sugli oggetti memorizzati nello stack*, in quanto si memorizzano *riferimenti*: altri thread potrebbero fare riferimento agli stessi oggetti, e modificarli *mentre sono memorizzati* nello stack

- sarebbe più sicuro archiviare nello stack delle copie (*cloni*) degli oggetti anziché mantenere dei riferimenti

Ogni oggetto Java (istanza di una sottoclasse qualsiasi della classe `Object`) fornisce i metodi di ***sincronizzazione***:

- **`wait()`**
  blocca l'esecuzione del thread invocante in attesa che un altro thread invochi i metodi `notify()` o `notifyAll()` per quell'oggetto. Il thread invocante deve essere in possesso del *lock* sull'oggetto; il suo blocco avviene dopo aver rilasciato il *lock*. Anche varianti con specifica di ***timeout***

- **`notify()`**
  risveglia un ***unico thread*** in attesa sul monitor dell'oggetto in questione. Se più thread sono in attesa, la scelta avviene in maniera ***arbitraria***, dipendente dall'implementazione della macchina virtuale Java. Il thread risvegliato compete con ogni altro thread, come di norma, per ottenere la risorsa protetta

- **`notifyAll()`**
  esattamente come `notify()`, ma risveglia ***tutti i thread*** in attesa per l'oggetto in questione. È necessario tutte le volte in cui ***più thread*** possono essere ***sospesi su differenti sezioni*** critiche dello stesso oggetto (***unica coda d'attesa***)

Esempio (*Produttori e Consumatori*):

```
public synchronized int get() {
   while (available == false)
      try {
         // attende un dato dai Produttori
         wait();
      } catch (InterruptedException e) {}
   }
   available = false;
   // notifica i produttori del consumo
   notifyAll();
…
}

public synchronized void put(int value) {
   while (available == true)
      try {
         // attende il consumo del dato
         wait();
      } catch (InterruptedException e) {}
   }
   …
   available = true;
   // notifica i consumatori della
   // produzione di un nuovo dato
   notifyAll();
}
```