programmazione concorrente

- ***single-tasking***
  - primi sistemi: un job alla volta, modalità batch
- ***multi-tasking***
  - più processi in esecuzione, contemporaneamente
  - sia applicazioni utente (word-processor, browser, mail …)
  - .. che processi di sistema
- ***multi-threading***
  - sistemi più recenti: più flussi di esecuzione nel contesto di uno stesso processo
  - un'applicazione può eseguire più compiti
    - un browser scarica diversi file, mentre stampa una pagina …
    - un'applicazione server può gestire più richieste in parallelo

- sistemi con ***un solo core***
  - il parallelismo di processi e thread viene simulato
    - time slicing
  - il tempo di elaborazione è suddiviso tra processi e/o thread
  - allo scadere di ogni unità di tempo, il s.o. opera un cambio di contesto (*context switch*)
- sistemi con ***più processori*** o con ***più core***
  - in ogni istante di tempo, ci possono essere più processi o thread fisicamente in esecuzione

- **processo**
  - per processo si intende un'istanza di un programma in esecuzione
- **thread**
  - un thread è una suddivisione di un processo in due o più filoni o sottoprocessi, che vengono eseguiti ***concorrentemente***
  - ogni processo ha almeno un thread
  - i thread ***condividono le risorse*** del processo (il suo contesto), compresa la memoria e i file aperti
  - ma hanno il ***proprio stack***, il ***proprio program counter***…

thread in Java

- ogni applicazione ha diversi thread “di sistema”
  - gestione della memoria e gestione degli eventi
- il lavoro inizia sempre con un solo thread
  - thread principale
- il programmatore ha la possibilità di ***creare altri thread***

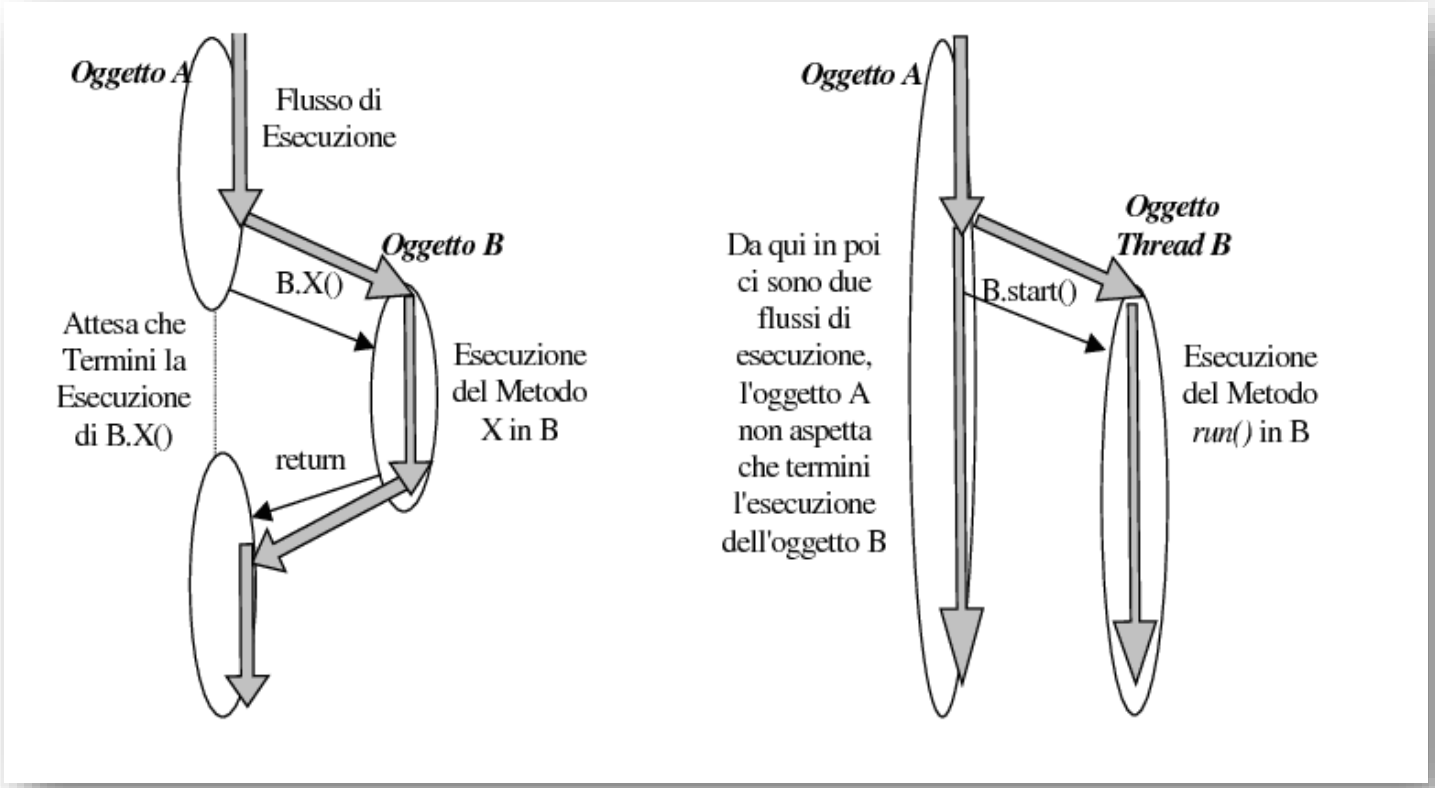
- la gestione dei thread da parte di Java è ***indipendente*** dalla piattaforma sottostante
- un thread è caratterizzato da:
  - un corpo (***thread body***)
  - uno stato (***thread state***)
  - un gruppo di appartenenza (***thread group***)
  - una priorità (***thread priority***)

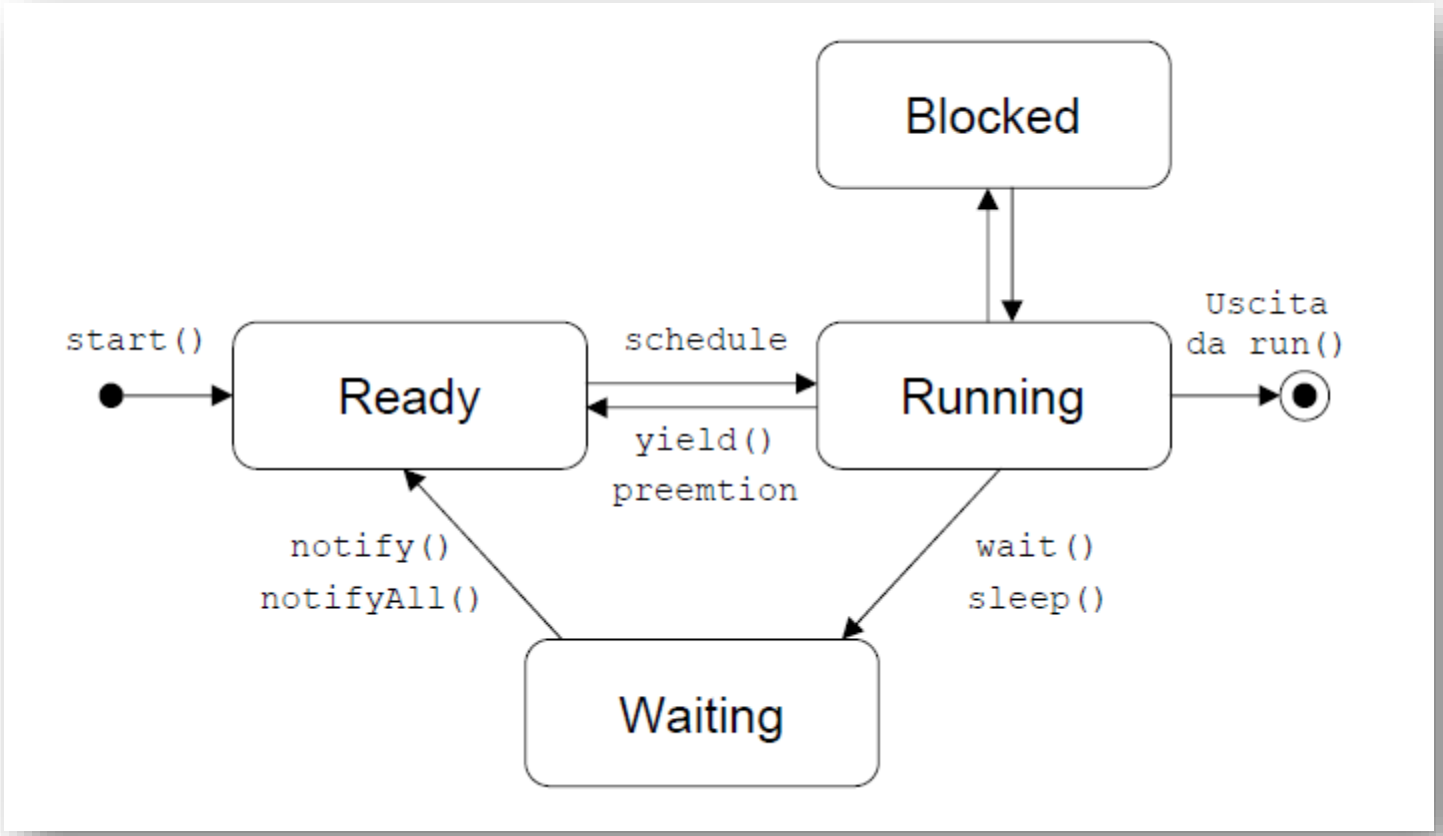
avvio di un thread

- per specificare il ***corpo*** di un thread, bisogna scrivere l'implementazione del suo metodo ***run***
- un thread viene ***avviato*** invocando il metodo ***start***
- in seguito, la macchina virtuale genera la ***biforcazione del flusso***
- a un certo punto lo scheduler eseguirà il metodo run
- non bisogna invocare direttamente run

metodo - thread

A. Ferrari




Blocked
start()
Ready
schedule
yield()
preemtion
Running
Uscita
da run()
notify()
notifyAll()
wait()
sleep()
Waiting

avviare un thread (primo metodo)

- realizzare un oggetto che ***implementa l'interfaccia Runnable***
- nell'interfaccia Runnable è presente il solo metodo ***run***, che deve essere implementato e contenere il codice specifico che deve essere eseguito
- l'oggetto Runnable viene passato al costruttore di Thread

esempio 1

A. Ferrari

```
public class EsempioThread implements Runnable {
    public void run() {
      System.out.println("Saluti dal thread!");
    }

    public static void main(String args[]) {
      Runnable saluto = new EsempioThread();
      Thread t = new Thread(saluto);
      t.start();
    }
}
```

secondo metodo

- realizzare una ***sottoclasse*** di **Thread**
  - Thread implementa Runnable, ma il metodo run non fa niente
  - bisogna fornire l'implementazione di run (ridefinire il metodo run)
- nota:
  - entrambi gli esempi invocano Thread.start per avviare il nuovo thread

esempio 2

A. Ferrari

```
public class EsempioThread extends Thread {
   public void run() {
      System.out.println("Saluti dal thread!");
   }

   public static void main(String args[]) {
      EsempioThread t = new EsempioThread();
      t.start();
   }
}
```

quale modo usare?

- il secondo modo è più facile da usare in applicazioni semplici
  - limitato dal fatto che la classe del task deve discendere da Thread
- l'uso di un oggetto Runnable è più generale e flessibile: l'oggetto può estendere qualsiasi classe
- la classe Thread definisce vari metodi utili per la gestione di thread
- metodi statici: forniscono informazioni sul thread chiamante, o ne modificano lo stato
- altri metodi invocati per gestire l'oggetto thread chiamato

alcuni metodi di Thread

A. Ferrari

```
public void start()                              //lancia il thread

public void run()                                //esegue il codice

public final void stop()                         //distrugge il thread

public final void suspend()                      //sospende il thread

public final void resume()                       //riattiva il thread sospeso

public static void sleep(long n)                 //sospende il thread per n ms

public final void setPriority(int priority)  //modifica la priorita'

public final int getPriority()                   //ottiene la priorità corrente

public static void yield()                       //rischedula

public final native boolean isAlive()            //true se il thread è vivo
```

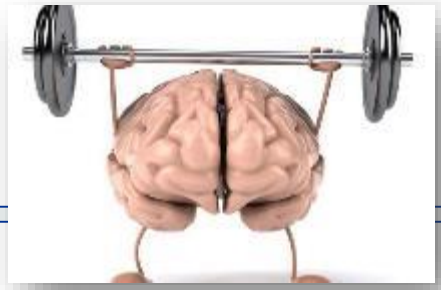
sleep

- **Thread.sleep** provoca la sospensione dell'esecuzione del thread corrente per un periodo specificato
- rende il processore disponibile ad altri thread dell'applicazione, o di altre applicazioni che girano sul computer

join

- il metodo **join** permette a un thread di aspettare il completamento di un altro thread
- Se t è un Thread, t.join() mette in attesa del completamento di t
- è possibile specificare anche il tempo massimo di attesa

- realizzare la classe Albero che cresce fa frutti e muore con intervalli casuali. Il Main lancia 3 thread della classe albero
- modificare l'esempio precedente lanciando 10 thread e bloccandone uno casualmente
- scrivere un'applicazione che carica valori casuali in un array (molto grande) poi lancia due thread che ricercano sequenzialmente la presenza di un valore, il primo nella prima metà degli elementi dell'array e l'altro nella seconda metà



problemi di esecuzione concorrente di un metodo

A. Ferrari

- più thread potrebbero eseguire in modo ***concorrente*** un metodo di una classe e causare errori logici
- *esempio*: movimenti di prelievo e versamento su un conto corrente.
- metodo prelievo
  - valore = getsaldo();
  - valore = valore – importo;
  - setsaldo(valore)
- metodo versamento
  - valore = getsaldo();
  - valore = valore + importo;
  - setsaldo(valore)
- si ipotizzi l'esecuzione concorrente partendo da un saldo iniziale di 1000 euro di un versamento di 100 euro e un prelievo di 50 euro

- Thread1 versamento

```
1.  val = getSaldo();
2.  val = val + importo;
3.  .
4.  .
5.  setSaldo(val)
6.  .
```

- Tread2 prelievo

```
1.  .
2.  .
3.  val = getsaldo();
4.  val = val - importo;
5.  .
6.  setSaldo(val)
```

soluzione

- una soluzione possibile è quella di rendere ***atomica*** (*indivisibile*) l'esecuzione di un intero metodo, nel nostro caso prelievo e versamento
- in java è necessario definire **synchronized** il metodo che deve essere indivisibile
- `public synchronized void preleva(double importo)` …
- `public synchronized void versa(double importo)` …

metodi sincronizzati

- non è possibile che due invocazioni di metodi sincronizzati si intreccino
  - ***se un thread sta eseguendo un metodo sincronizzato su un oggetto***
  - ***tutti i thread che invocano metodi sincronizzati sullo stesso oggetto si bloccano in attesa della terminazione del precedente metodo***

- la sincronizzazione dei thread in Java è effettuata dai ***monitor***
- un monitor è un oggetto i cui metodi possono essere eseguiti solo in modalità mutuamente esclusiva da parte di più thread concorrenti

- realizzare la classe monitor contoCorrente
- lanciare due thread che effettuano operazioni di prelievo e versamento dal conto corrente