Multithreading in Java

Flusso (thread) di esecuzione

- In un programma sequenziale esiste un solo flusso di esecuzione attivo
- In un programma concorrente esistono più flussi di esecuzione attivi (task)
- Scopo?
  - Applicazioni interattive
  - Server
- Thread vs Processi:
  - Thread sono più leggeri dei processi
  - Sfruttano una comunicazione a memoria condivisa

Thread in Java—metodo “Thread”

1. Definire una classe che eredita da Thread e contiene il metodo run()

```
class ListSorter extends Thread {
    …
    public void run() {
     gli statements del thread che si vuol definire
    }
}
```

2. Creare istanza della classe

```
ListSorter concSorter = new ListSorter(list1)
```

3. Invocare il metodo start(), che a sua volta chiama run()

```
concSorter.start();
```

lancia il thread e ritorna immediatamente

Thread in Java—metodo “Runnable”

1. Definire una classe che implementa l'interfaccia Runnable, eventualmente estende altra classe e che contiene il metodo run()

```
class ListSorter implements Runnable {
        …
        public void run() {
                gli statements del thread che si vuol definire
        }
}
```

2. Creare istanza

```
ListSorter concSorter = new ListSorter(list1);
```

3. Creare thread

```
Thread myThread = new Thread(concSorter);
```

4. Attivare metodo run, attraverso metodo start()

```
myThread.start()
```

Quale metodo utilizzare?

- **Metodo Thread**
  - Vantaggi: più intuitivo, permette di istanziare un solo oggetto.
  - Svantaggi: poco flessibile.
- **Metodo Runnable**
  - Vantaggi: permette di avviare classi che estendono altre classi come Thread. Vedremo più avanti che esistono meccanismi più avanzati per la schedulazione di Thread che preferiscono oggetti di tipo Runnable.
  - Svantaggi: è un metodo più complicato.

Non-Determinismo

```
class MyThread implements Runnable {
    private String message;
    public MyThread(String m) {
        message = m;
    }
    public void run() {
        for (int r = 0; r < 90000; r++)
            System.out.println(message);
    }
}
class ProvaThread {
    public static void main(String[] args) {
        Thread t1, t2;
        MyThread r1, r2;
        r1 = new MyThread("primo thread");
        r2 = new MyThread("secondo thread");
        t1 = new Thread(r1);
        t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

Quale sarà l'output?

Dati condivisi

- Può essere necessario imporre che certe sequenze di operazioni che accedono a dati condivisi vengano eseguite dai task in **mutua esclusione**

```
class ContoCorrente {
        private float saldo;
        public ContoCorrente (float saldoIniz)
                { saldo = saldoIniz; }
        public void deposito (float soldi)
                { saldo += soldi; }
        public void prelievo (float soldi)
                { saldo -= soldi; }
        ...
}
```

che succede se due Thread concorrenti cercano l'uno di depositare e l'altro di prelevare?

Operazioni non atomiche

Saldo iniziale = 100

Thread 1	Thread 2
deposito(50)	**prelievo(50)**
read(saldo) -> 100	
	read(saldo) -> 100
sum(50+100) -> 150	
	sub(100-50) -> 50
	write(saldo) -> 50
write(saldo) -> 150	

Saldo finale = 150

Operazioni atomiche

Saldo iniziale = 100

Thread 1	Thread 2
deposito(50)	**prelievo(50)**
read(saldo) -> 100	
sum(50+100) -> 150	
write(saldo) -> 150	
	read(saldo) -> 150
	sub(150-50) -> 100
	write(saldo) -> 100

Saldo finale = 100

Come rendere i metodi "atomici"

- La parola chiave "synchronized"

```
class ContoCorrente {
        private float saldo;
        public ContoCorrente (float saldoIniz)
                { saldo = saldoIniz; }
        public synchronized void deposito (float soldi)
                { saldo += soldi; }
        public synchronized void prelievo (float soldi)
                { saldo -= soldi; }
        ...
}
```

Metodi synchronized

- Java associa un *lock* (*monitor*) a ciascun oggetto
  - Solo un thread alla volta può eseguire il codice dell'oggetto
- Quando il metodo synchronized viene invocato
  - se nessun metodo synchronized è in esecuzione, l'oggetto viene bloccato (*locked*) e quindi il metodo viene eseguito
  - se l'oggetto è bloccato, il task chiamante viene sospeso e messo in **coda** fino a quando il task bloccante libera il lock

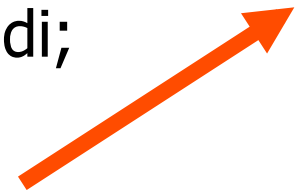
Metodi synchronized (2)

- Quando un metodo **synchronized** viene invocato da un altro metodo **synchronized** appartenente al medesimo oggetto, il thread chiamante non deve competere per il **monitor**, in quanto quest'ultimo è già stato acquisito durante l'invocazione del primo metodo (*reentrant lock*)
- L'accesso mutuamente esclusivo vale solo per i metodi dichiarati **synchronized**: l'accesso attraverso gli altri metodi non è mutuamente esclusivo, cioè può avvenire anche mentre un thread ha acquisito il **monitor**

Precondizioni per metodi synchronized

- Come evitare il prelievo se il conto va "in rosso"?

```
class ContoCorrente {
        private float saldo;
        ...
        synchronized public void prelievo (float soldi) {
                while (saldo-soldi<0) wait();
                saldo -= soldi;
        }
        ...
}
```



rilascia il lock sull'oggetto e sospende il task

Come risvegliare un task in wait?

```
class ContoCorrente {
        private float saldo;
        public ContoCorrente (float saldoIniz)
                { saldo = saldoIniz; }
        synchronized public void deposito (float soldi) {
                saldo += soldi;
                notify();
        }
         synchronized public void prelievo (float soldi) {
                while (saldo-soldi<0) wait();
                saldo -= soldi;
        }

        ...
}
```

risveglia un task sospeso in wait, se esiste *nondeterminismo*

Attenzione!
if (saldo-soldi<0) wait();
Non è sufficiente

Primitive di sincronizzazione

- Le primitive di sincronizzazione **wait**, **notify** e **notifyAll** sono associate a ogni oggetto, in quanto definite nella classe **Object**.
- Consentono a un thread di sospendersi all'interno di un monitor (**wait**), e di risvegliare uno (**notify**) o tutti (**notifyAll**) i thread sospesi.
- Tali primitive operano sul monitor associato all'oggetto, pertanto possono essere invocate all'interno di un thread solo dopo che esso ha acquisito il monitor:
  - Cioè solo se il thread sta eseguento all'interno di un blocco o metodo **synchronized**
  - Se si invoca una di queste primitive su un oggetto per cui non si è acquisito il monitor si ottiene una **IllegalMonitorStateException**

Il blocco `synchronized`

- Talvolta risulta necessario controllare l'accesso concorrente a porzioni di codice con una granularità più fine del metodo
  - Più è grande la porzione di codice sincronizzata, minore il parallelismo
- In questi casi è possibile impiegare il blocco **`synchronized`**

```
synchronized (obj) {
    ... codice critico ...
}
```

- La semantica del blocco **`synchronized`** è simile a quella dei metodi synchronized, con la differenza che il monitor viene acquisito sull'oggetto `obj` anzichè su `this`.

```
void synchronized m() {
... codice critico ...
}
```

```
void m() {
    synchronized(this) {
     ... codice critico ...
    }
}
```

Esempio: una coda fifo condivisa

- Operazione di inserimento di elemento:
  - sospende task se coda piena
    - while (codaPiena()) wait();
  - al termine
    - notify();
- Operazione di estrazione di elemento:
  - sospende task se coda vuota
    - while (codaVuota()) wait();
  - al termine
    - notify();

invece notifyAll risveglia tutti...
MA uno solo guadagna il lock

Priorità e scheduling

- Assegnabile priorità (1-10) ai task (**setPriority**); default 5
- Alcune piattaforme supportano il "time slicing"
- In assenza, un thread viene eseguito fino al completamento, a meno che non diventi "blocked", "waiting" o "dead"
- Compito dello scheduler è mandare in esecuzione il thread con priorità più alta

Alcuni metodi della classe Thread

void start() – Avvia il thread (eseguendo il metodo **run**())

void join() – Aspetta fino a quando il thread non termina

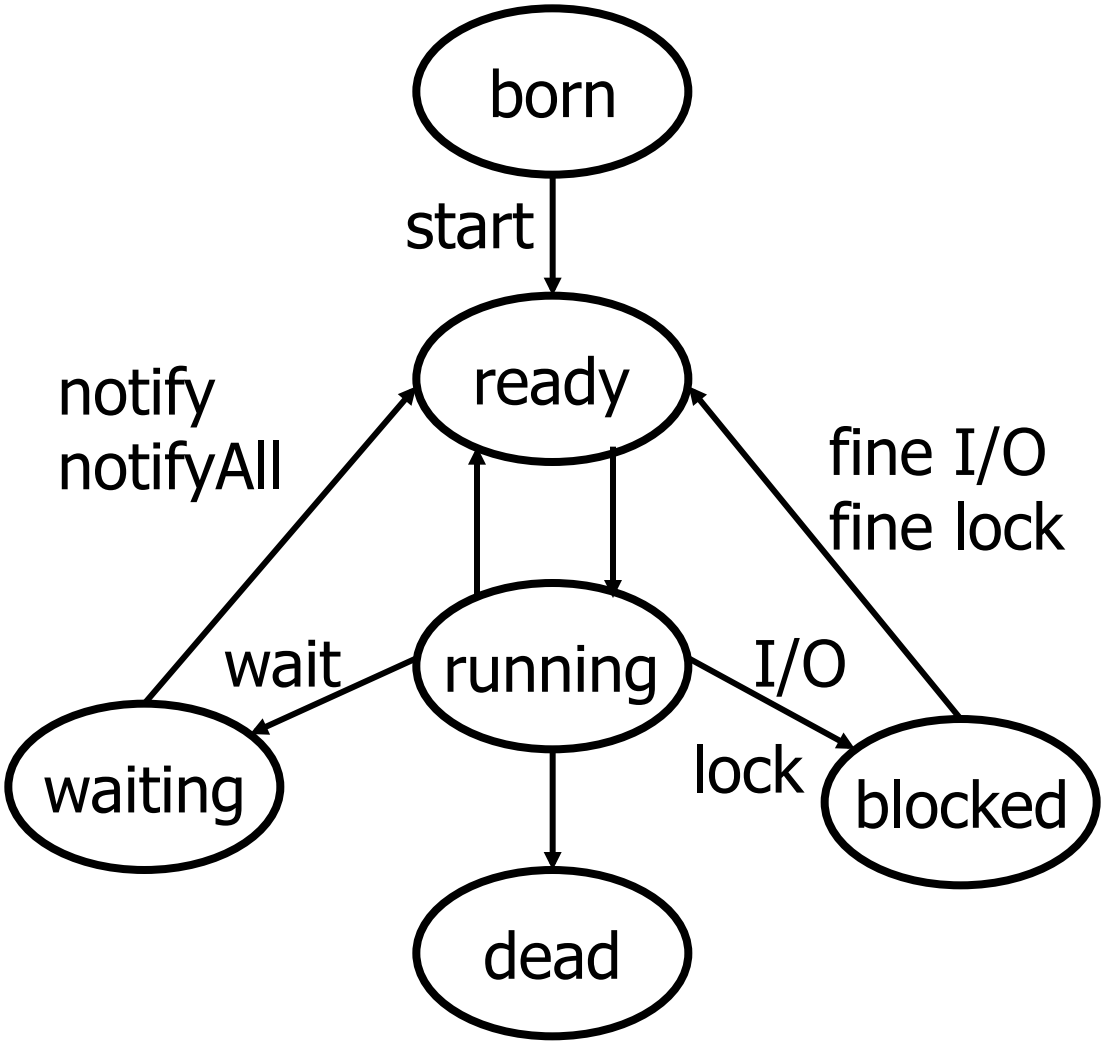
boolean isAlive() – Controlla se il thread è in esecuzione

static void sleep(int ms) – Aspetta **ms** millisecondi

static boolean holdsLock(Object obj) – Restituisce true se il thread corrente ha acquisito un lock del monitor di **obj**

static void yield() – Sospende temporaneamente il thread corrente per permettere l'esecuzione degli altri thread in esecuzione

Ciclo di vita di un thread



Esecutori

- Sono una famiglia di classi che implementano l'interfaccia **ExecutorService**
- Si utilizzano come metodo alternativo per eseguire delle classi che implementano **Runnable** (definite **Task**)

void submit(Runnable task) – Schedula l'esecuzione del task
void shutdown() – Impedisce all'esecutore di accettare nuovi task
boolean awaitTermination(long timeout, TimeUnit unit) – Attende la fine dell'esecuzione di tutti i task oppure il tempo specificato come timeout
…

Esempio uso Esecutori

```
public class EsempioEsecutori {
    public static void main(String[] args) {
        ExecutorService esecutore = Executors.newFixedThreadPool(2);
        esecutore.submit(new TaskCheImpiegaTreSecondi("TaskA"));
        esecutore.submit(new TaskCheImpiegaTreSecondi("TaskB"));
        esecutore.submit(new TaskCheImpiegaTreSecondi("TaskC"));
        esecutore.shutdown();
    }
}

class TaskCheImpiegaTreSecondi implements Runnable {

    private final String nome;

    public TaskCheImpiegaTreSecondi(String nome) {
        this.nome = nome;
    }

    @Override
    public void run() {
        System.out.println(nome + " avviato alle: " + new Date());
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {}
        System.out.println(nome + " completato alle: " + new Date());
    }
}
```

Esempio uso Esecutori (2)

```
TaskA avviato alle: Sun Mar 25 08:30:52 CEST 2012
TaskB avviato alle: Sun Mar 25 08:30:52 CEST 2012
TaskB completato alle: Sun Mar 25 08:30:55 CEST 2012
TaskA completato alle: Sun Mar 25 08:30:55 CEST 2012
TaskC avviato alle: Sun Mar 25 08:30:55 CEST 2012
TaskC completato alle: Sun Mar 25 08:30:58 CEST 2012
```

Java offre degli **ExecutorService** predefiniti all'interno della classe **Executors**.

`Executors.`	
newSingleThreadExecutor()	Esegue al massimo un Task contemporaneamente
newFixedThreadPool(int n)	Esegue al massimo n Task contemporaneamente
newCachedThreadPool()	Esegue un numero illimitato di Task contemporaneamente

Vantaggi uso Esecutori

- Permettono di disaccoppiare il concetto di **Thread** dal concetto di **Task**.
  - Uno stesso thread può eseguire più task in sequenza senza dover essere distrutto e ricreato → aumenta le prestazioni
- Permettono di avere controllo sul numero di Thread in esecuzione dando la possibilità di accodare i Task in eccesso
- Esistono **Esecutori** più avanzati che permettono di schedulare l'esecuzione di task ripetuti o dopo un intervallo di tempo specificato. Esempio:
  - Interfaccia: `ScheduledExecutorService`
  - Implementazione: `Executors.newScheduledThreadPool(int n)`

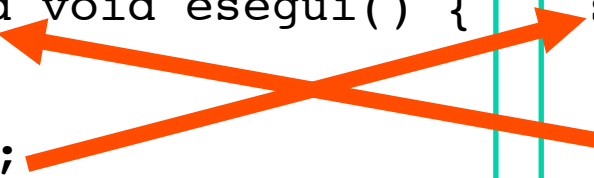
Deadlock

- Il deadlock è una situazione di stallo in cui due (o più) processi (o azioni) si bloccano a vicenda aspettando che uno esegua una certa azione (es. rilasciare il controllo su una risorsa come un file, una porta input/output ecc.) che serve all'altro e viceversa.
- Per evitare il deadlock è necessario non avere dipendenze circolari per acquisire le risorse (lock)
  - Il grafo delle risorse richieste non deve avere cicli!
- Esempio:
  - Il thread1 ha il lock su A e vuole acquisire anche il lock su B
  - Il thread2 ha il lock su B e vuole acquisire il lock su A
  - Deadlock: entrambi i thread restano bloccati nella speranza di acquisire il lock

Esempio Deadlock

```
class Oggetto A {
 OggettoB b;
 ...
 synchronized void esegui() {
     ...
     b.test();
     ...
  }
 }
```

```
class Oggetto B {
 OggettoA a;
 ...
 synchronized void test() {
     ...
     a.esegui();
     ...
  }
 }
```



Bug Concorrenti

- Quando due thread diversi accedono a un oggetto contemporaneamente, possono verificarsi dei problemi come la comparsa di **ConcurrentModificationException**.
- Alcune di queste anomalie sono dette **heisenbug**, poiché tendono a non essere riproducibili in un ambiente controllato (es. debugging).
  - Heisenbug is a computer programming jargon term for a software bug that seems to disappear or alter its behavior when one attempts to study it. The term is a pun on the name of Werner Heisenberg, the physicist who first asserted the observer effect of quantum mechanics, which states that the act of observing a system inevitably alters its state. (da Wikipedia)

Oggetti Thread-Safe

Un oggetto si definisce **thread-safe** quando permette l'accesso contemporaneo a più thread senza anomalie (cioè l'oggetto offre soltanto operazioni **atomiche**).

Collezioni e Mappe Thread-Safe

- La maggior parte delle collezioni e mappe del package java.util NON sono thread-safe!
- Nel package java.util.concurrent è possibile trovare delle versioni thread-safe di tutte le collezioni e mappe che useremo. Ovviamente il costo di questa possibilità si paga in termini di prestazioni inferiori.

Non thread-safe	Thread-safe (concurrent)	Thread-safe (synchronized)
ArrayList	CopyOnWriteArrayList	Collections.synchronizedList(new ArrayList())
LinkedList	-	Collections.synchronizedList(new LinkedList())
HashSet	CopyOnWriteArraySet	Collections.synchronizedSet(new HashSet())
TreeSet	ConcurrentSkipListSet	Collections.synchronizedSet(new TreeSet())
HashMap	ConcurrentHashMap	Collections.synchronizedMap(new HashMap())
TreeMap	ConcurrentSkipListMap	Collections.synchronizedMap(new TreeMap())

Esercizio 1

- Si scriva un programma che esegua 2 thread.
- Il primo thread fa le seguenti operazioni 10 volte:
  - Incrementa una variabile della classe che lo ha lanciato
  - Stampa il valore di tale variabile
- Il secondo thread fa le seguenti operazioni 10 volte:
  - Decrementa la variabile della classe che lo ha lanciato
  - Stampa il valore di tale variabile
- Al termine dei due thread viene stampato “Fatto!”.

Esercizio 2

- Un call center possiede 5 centralinisti. Nel momento di maggior affluenza ci sono 100 clienti che chiamano. Una conversazione dura mediamente tra 4 e 6 secondi (per semplicità ...).
- Se tutti i centralinisti sono occupati il cliente resta in attesa.
- Si simuli questa situazione con un programma Java in cui si ha un Thread per ogni cliente.
- Al termine di ogni chiamata il cliente stampa su video il tempo totale della sua chiamata (attesa più conversazione).

Esercizio 2 (estensioni)

- **Estensione 1**: si modifichi la soluzione dell'esercizio 2 utilizzando l'Esecutore **Executors.newCachedThreadPool**

- **Estensione 2**: si modifichi la soluzione dell'esercizio 2 facendo in modo che i clienti messi in attesa per primi siano i primi ad essere serviti.