

Java

concorrenza

G. Prencipe

prencipe@di.unipi.it

Introduzione

- Un *processo* è un programma in esecuzione con un proprio spazio di indirizzi
- Un *Sistema Operativo multitasking* è in grado di eseguire diversi processi *concorrentemente* assegnando periodicamente la CPU ad ognuno di essi
- Spesso si ha la necessità di dividere un processo in sottoprocessi da eseguire indipendentemente
 - Ognuno di queste sottoparti viene detta *thread*, ed è eseguita come se avesse a sua disposizione una propria CPU

Threads

- Un *thread* è un singolo flusso di controllo all'interno di un processo
- Un processo può dunque avere più thread in esecuzione concorrentemente
- Ci sono molti utilizzi per i thread
 - In generale lo scopo è quello di avere solo una parte del programma (thread) a occuparsi di un particolare evento o legata a una particolare risorsa, e non l'intero processo
 - Quindi si crea un thread che si occupa di quell'evento indipendentemente da resto del programma

Threads

- È semplice comprendere il meccanismo di base dei thread
- Purtroppo però, comprendere *a fondo* i thread non è semplice
 - Come per il polimorfismo, richiede un diverso modo di pensare

Threads

- Una delle ragioni principali per introdurre la concorrenza è legata ad avere interfacce utente interattive
 - Un programma deve porre sia attenzione ai calcoli della CPU che a cosa dice l'utente
 - È quindi meglio separare questi due "processi", che possono essere parte dello stesso programma

Threads

- Con architetture multiprocessore, i thread sono distribuiti fra i vari processori
 - Questo incrementa drasticamente le performance
- Bisogna tenere a mente però che un programma con molti thread deve essere in grado di essere eseguito anche su macchine con una sola CPU
 - In altre parole, lo stesso programma deve poter essere scritto anche senza l'utilizzo dei thread
 - Essi forniscono un modo per organizzare il codice e chiedere alla CPU di trattare alcune parti di un processo come parti distinte

Thread

- Il modo più semplice per creare un thread è di ereditare da **java.lang.Thread**
- Il metodo più importante in **Thread** è **run()**
 - Bisogna riscriverlo (override) affinché il thread faccia il lavoro che vogliamo
 - **run()** è il codice che sarà eseguito concorrentemente agli altri thread nel programma

Thread

Id di ogni thread in una variabile **statica**

Si assegna un nome al thread

Si avvia il thread. Il metodo **start()** esegue delle speciali inizializzazioni per il thread e invoca **run()**

Si recupera il nome

Sono creati 5 thread

```
public class SimpleThread extends Thread {
    private int countDown = 5;
    private static int threadCount = 0;
    public SimpleThread() {
        super("" + ++threadCount);
        start();
    }
    public String toString() {return "#" + getName() +
        ":" + countDown;}
    public void run() {
        while(true) {
            System.out.println(this);
            if(--countDown == 0) return;
        }
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new SimpleThread();
    }
}
```

Thread

- Quindi i passi sono
 - Invocare il costruttore per la nostra classe che estende **Thread**
 - Invocare **start()** che configura il thread e invoca **run()**
- Se non si invoca **start()** il thread non parte in alcun modo
 - **run()** non è un metodo che possiamo invocare direttamente

Thread

- L'output dei programmi che utilizzano thread non è sempre lo stesso
- Dipendono infatti dal meccanismo di scheduling dei thread del Sistema Operativo

Cedere il passo

- Se sappiamo che il nostro thread ha già fatto abbastanza nel **run()**, con il metodo **yield()** è possibile *suggerire* allo scheduler di cedere la CPU a qualche altro thread
 - Si tratta solo di un *suggerimento*
- ```
public void run() {
 while(true) {
 System.out.println(this);
 if(--countDown == 0) return;
 yield();
 }
}
```

## Riposarsi un po'....

- Un altro modo per controllare il comportamento di un thread è invocando **sleep()** per cessare l'esecuzione per un certo numero di millisecondi
- ```
public void run() {  
    while(true) {  
        System.out.println(this);  
        if(--countDown == 0) return;  
        try{sleep(100);}catch(InterruptedException e){}  
    }  
}
```
- Vediamo la differenza con **sleep()** in **SimpleThread.java**

Priorità

- La priorità di un thread comunica allo scheduler quanto è importante questo thread
- Sebbene l'ordine con cui la CPU esegue i thread non è determinato, se esiste un certo numero di thread bloccati e in attesa di essere eseguiti, lo scheduler tenderà a scegliere prima quello con priorità maggiore
 - Questo non vuol dire che i thread a priorità bassa non verranno mai eseguiti
 - Non si ha *starvation*

Priorità

- Le priorità si impostano con **setPriority()** e si leggono con **getPriority()**
- I campi statici **MAX_PRIORITY**, **MIN_PRIORITY** e **NORM_PRIORITY** in **Thread** conservano la massima priorità (10), la minima priorità (1), e la priorità di default (5) per un thread

Thread dèmoni

- I *thread dèmoni* sono thread che hanno il compito di fornire un servizio generale in background fino a che il programma è in esecuzione, ma non sono intesi come parte fondamentale del programma
 - Cioè, quando tutti i thread non-demoni sono completati, il programma termina (anche se ci sono thread demoni in esecuzione)
 - Se ci sono thread non-demoni ancora in esecuzione, il programma non termina
- Un esempio di thread non-demone è il **main()**

Thread dèmoni

- Si dichiara un thread come demone invocando **setDaemon(boolean b)** prima di avviarlo
- Per scoprire se un thread è in demone si invoca **isDaemon()**
- Se un thread è un demone, allora tutti i thread che egli crea saranno automaticamente dèmoni
- Vediamo un esempio

Thread dèmoni

```
public class SimpleDaemons extends Thread {
    public SimpleDaemons() {
        setDaemon(true);
        start();
    }
    public void run() {
        while(true) {
            try {
                sleep(100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            System.out.println(this);
        }
    }
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new SimpleDaemons();
    }
}
```

Ogni demone viene
posto in **sleep(100)**

Il **main()** lancia 10
dèmoni e poi termina

Come risultato, nessun
dèmone riesce a
effettuare la stampa

Attendere il completamento

- Un thread invoca **join()** su un altro thread per aspettare il completamento di quest'ultimo prima di proseguire
- In altre parole, se un thread invoca **t.join()** su un altro thread **t**, allora il thread invocante viene sospeso fino a che **t** non finisce (cioè, quando **t.isAlive()** è **false**)

Attendere il completamento

- Si può invocare **join()** con un argomento che indica il tempo d'attesa (indicato in millisecondi)
 - Se il thread da attendere non termina entro questo tempo, la chiamata a **join()** ritorna e il thread invocante prosegue
- La chiamata a **join()** può lanciare un'eccezione, e quindi va inserita in un blocco **try-catch**

Interrompere un thread

- Abbiamo visto che sia **sleep()** che **join()** necessitano di gestire una eccezione **InterruptedException**
- Questa eccezione viene lanciata nel momento in cui un thread interrompe la propria esecuzione, invocando il metodo **interrupt()**
- Il metodo **isInterrupted()** controlla se questo thread è stato interrotto

L'interfaccia Runnable

- Abbiamo visto che per ottenere un thread bisogna estendere **Thread**
- Se però la classe in cui vogliamo implementare un **Thread** già eredita da un'altra classe, non possiamo estendere **Thread**
 - Non c'è ereditarietà multipla in Java
- Abbiamo un'alternativa: implementare l'interfaccia **Runnable**

L'interfaccia Runnable

- L'interfaccia **Runnable** specifica solo che bisogna implementare un metodo **run()**
 - La classe **Thread** implementa a sua volta **Runnable**
- Una volta implementato **Runnable**, per creare un thread si invoca il costruttore di **Thread** che prende come argomento un **Runnable**
- Con questo thread si invoca **start()**

L'interfaccia Runnable

```
public class RunnableThread implements Runnable {
    private int countDown = 5;
    public String toString() {
        return "#" + Thread.currentThread().getName() +
            ": " + countDown;
    }
    public void run() {
        while(true) {
            System.out.println(this);
            if(--countDown == 0) return;
        }
    }
    public static void main(String[] args) {
        for(int i = 1; i <= 5; i++)
            new Thread(new RunnableThread(), "" + i).start();
    } //~
```

Per ottenere un riferimento al thread si invoca il metodo statico **Thread.currentThread()**

Esempio

```
public class AlwaysEven {
    private int i;
    public void next() { i++; i++; }
    public int getValue() { return i; }
    public static void main(String[] args) {
        final AlwaysEven ae = new AlwaysEven();
        new Thread("Watcher") {
            public void run() {
                while(true) {
                    int val = ae.getValue();
                    if(val % 2 != 0) {
                        System.out.println(val);
                        System.exit(0);
                    }
                }
            }
        }.start();
    } //~
```

next() mantiene i pari

Il compito del thread **Watcher** è di assicurarsi che i sia sempre pari

Se non lo è blocca tutto

Se eseguiamo il codice vedremo che *si blocca!!!!*

Domanda: perché??

Risorse condivise

- Quello che capita è che
 - **i** è una *risorsa condivisa*
 - Viene scritta da **AlwaysEven** e letta da **Watcher**
 - Può capitare che **Watcher** acceda a **i** quando il doppio incremento non sia stato ancora completato
 - Cioè che **Watcher** venga eseguito dallo scheduler tra un **i++** e il successivo (quindi **Watcher** legge **i** quando è dispari)

Risorse condivise

- Il problema con la programmazione multithread è che è possibile avere più thread che accedono alla stessa risorsa (*risorse condivise*)
- Fino a quando l'accesso a queste risorse è di sola lettura, non si hanno grossi problemi
 - Come nell'esempio visto: lo stato di **AlwaysEven** può essere momentaneamente inconsistente, ma poi sappiamo che tornerà consistente con **i** pari
- Quando però più thread possono accedere a una risorsa e modificarla in qualche modo, si possono avere effetti collaterali non desiderati se opportune precauzioni non vengono prese

Risorse condivise

- Per controllare gli accessi alle risorse condivise si possono ad esempio adottare i *semafori*
- Un *semaforo* è un oggetto che controlla l'accesso a una risorsa condivisa o a una sezione di codice in cui si accedono risorse condivise (sezione critica)
- Se il suo valore è 0 allora l'accesso alla risorsa condivisa è possibile
 - Il thread vi accede, incrementando il valore del semaforo
- Altrimenti l'accesso non è possibile e il thread che chiede l'accesso deve attendere
 - Quando un thread termina l'utilizzo della risorsa condivisa, decrementa il valore del semaforo

Semafori

- Dati che incremento e decremento sono operazioni atomiche (cioè che non possono essere interrotte), il semaforo evita che due thread possano accedere alla stessa risorsa critica contemporaneamente
 - Il problema con **AlwaysEven** visto prima è che avevamo un doppio incremento di **i**, e questa operazione non è intrinsecamente atomica
 - La si può rendere comunque sicura adottando un semaforo

Semafori

```
public class Semaphore {  
    private int semaphore = 0;  
    public void acquire() {  
        while(semaphore!=0){;}  
        ++semaphore; }  
    public void release() { --semaphore; }  
}  
} ///:~
```

- L'idea è che un thread esegua il ciclo in **acquire()** fino a che non restituisce **true**
- A questo punto acquisisce il semaforo e altri thread devono aspettare (nel ciclo di **acquire()**) prima di poter accedere la risorsa condivisa

Esempio

```
public void next() {  
    this.semaphore.acquire();  
    i++; i++;  
    this.semaphore.release();  
}  
  
public int getValue() {  
    int result;  
    this.semaphore.acquire();  
    result=i;  
    this.semaphore.release();  
    return result;}  
}
```

Proviamo a eseguire questo codice (**AlwaysEvenSemaphore** nel progetto **AlwaysEven**)

Ci sono ancora problemi....

Come mai??

Semafori

- Nell'esempio abbiamo due thread
 - Quali sono??

Semafori

- Nell'esempio abbiamo due thread
 - **Watcher** e **main()**
 - Se lo *scheduler* effettua le seguenti operazioni
 - Esegue il ciclo della **acquire()** per **getValue()** in **Watcher**; viene ottenuto l'accesso a **i**
 - Poi esegue il ciclo della **acquire()** per **next()** nel **main**, prima che la **getValue()** di prima abbia incrementato il semaforo; viene ottenuto l'accesso a **i**
 - Entrambi i thread accedono a **i**!!

Risorse condivise

- Per risolvere questo problema è necessario *garantire* che solo *un thread alla volta* acceda a una risorsa condivisa
- In Java ogni oggetto (istanza di **Object**) ha associato un *mutual exclusion lock* che consente di accedere all'oggetto in mutua esclusione
 - Cioè un solo thread alla volta
- Non si può accedere direttamente a questo lock, ma viene gestito automaticamente quando si dichiara un metodo o un blocco di codice come **synchronized**

La parola chiave **synchronized**

- Quando un metodo è **synchronized** lo si può invocare su un oggetto solo se si è acquistato il lock su tale oggetto
- Quindi i metodi **synchronized** hanno accesso esclusivo ai dati incapsulati nell'oggetto (se a tali dati si accede solo con metodi **synchronized**)

La parola chiave **synchronized**

- Quindi, per controllare l'accesso a una risorsa condivisa, bisogna prima metterla in un oggetto
- Poi, qualsiasi metodo che voglia accedere la risorsa deve essere dichiarato **synchronized**
`synchronized void f(){...}`
- I metodi non **synchronized** non richiedono l'accesso al lock e quindi si possono richiamare in qualsiasi momento

La parola chiave **synchronized**

- Dato che tipicamente i dati (variabili) di una classe si dichiarano **private**, si accede ad essi solo tramite metodi
- Quindi, per garantire l'accesso esclusivo a questi dati è sufficiente dichiarare **synchronized** questi metodi

La parola chiave **synchronized**

- Se un thread è in un metodo **synchronized**, tutti gli altri thread non possono eseguire nessuno dei metodi **synchronized** della classe fino a quando il primo thread non ritorni dalla chiamata al metodo
 - Possono eseguire i metodi non **synchronized**
- Il thread che sta eseguendo un metodo **synchronized** può, invece, eseguire altri metodi **synchronized** sullo stesso oggetto
 - È l'unico a cui è permesso questo, visto che ha già acquisito il lock sull'oggetto

Esempio

- Nel nostro esempio è sufficiente dichiarare **next()** e **getValue()** come **synchronized**

```
public synchronized void next() { i++; i++; }  
public synchronized int getValue() { return i; }
```

- Nota: *entrambe* vanno dichiarate **synchronized**, altrimenti quella non dichiarata **synchronized** può violare il non accesso simultaneo a **i**
- Vediamo l'esecuzione in **AlwaysEvenSynch**, che è infinita

La parola chiave **synchronized**

- In dettaglio, quando un thread deve eseguire un metodo **synchronized** su un oggetto, si blocca finché non riesce ad ottenere il lock sull'oggetto
- Quando lo ottiene può eseguire il metodo (e tutti gli altri **synchronized**)
- Gli altri thread rimarranno bloccati finché il lock non viene rilasciato
- Quando il thread esce dal metodo **synchronized** rilascia automaticamente il lock

Operazioni atomiche

- Una serie di operazioni in Java non ha bisogno di essere sincronizzata, in quanto *atomiche*
 - Cioè operazioni non interrompibili dallo scheduler
- Semplice assegnamento e restituzione di un valore quando la variabile in questione è primitiva, ma non **long** o **double**
 - Per ottenere atomicità anche con **long** e **double** bisogna dichiararle usando la parola chiave **volatile**
- Notare che l'incremento *non* è un'operazione atomica

Operazioni atomiche

- Comunque una buona regola generale da seguire con le risorse condivise è
 - Se bisogna *sincronizzare* un metodo in una classe, allora è più sicuro *sincronizzarli* tutti
 - Infatti non è sempre prevedibile (quando si usano i thread) cosa succede quando uno dei metodi non viene *sincronizzato*

Sezioni critiche

- A volte vogliamo solo prevenire l'accesso simultaneo solo a una *porzione* di un metodo
 - La sezione di codice che vogliamo isolare prende il nome di *sezione critica*
- In Java singoli blocchi di codice possono essere dichiarati **synchronized** su un certo oggetto
- Un solo thread alla volta può eseguire un tale blocco su uno stesso oggetto
- In questo modo si minimizzano le parti di codice da serializzare

Blocchi **synchronized**

- I blocchi **synchronized** hanno la seguente forma

```
synchronized(syncObject) { /*sezione critica*/ }
```
- **syncObject** è il riferimento all'oggetto di cui bisogna ottenere il lock prima di poter eseguire la sezione critica
 - Con i metodi **synchronized** il lock è automaticamente richiesto sull'oggetto che contiene il metodo

Blocchi **synchronized**

- Per ottenere il lock sull'oggetto che contiene il blocco **synchronized** (come avviene per i metodi **synchronized**) è sufficiente usare **this**

```
synchronized(this) { /*sezione critica*/ }
```
- I blocchi **synchronized** consentono comunque di indicare un oggetto diverso da **this** per la sincronizzazione
 - In questo caso la correttezza è affidata a chi usa l'oggetto condiviso e non all'oggetto stesso
- Spesso questa è l'unica alternativa quando non si può/vuole modificare la classe dell'oggetto condiviso (introducendo metodi **synchronized**)

Ereditarietà

- La specifica **synchronized** non fa parte vera e propria della segnatura di un metodo
- Quindi una classe derivata può ridefinire un metodo **synchronized** come non **synchronized** e viceversa

Variabili statiche

- Metodi e blocchi **synchronized** *non assicurano* l'accesso mutuamente esclusivo ai dati **statici**
 - Essi sono infatti condivisi da tutti gli oggetti della stessa classe
- Per accedere in modo sincronizzato ai dati statici si deve ottenere il lock su questo oggetto **Class**
 - Dichiarando un metodo statico come **synchronized**
 - Dichiarando un blocco come **synchronized** sull'oggetto **Class**

```
synchronized(NomeClasse.class){...}
```

Stati dei thread

- Un thread può essere in uno dei seguenti stati
 - *New*
 - *Runnable*
 - *Dead*
 - *Blocked*

Stati -- *New*

- In *New* l'oggetto thread è stato creato ma non è stato ancora avviato, quindi non può ancora essere eseguito

Stati -- *Runnable*

- In *Runnable* un thread può essere eseguito non appena lo scheduler lo seleziona
- Quindi non c'è niente che blocca l'esecuzione del thread, dipende solo dalle decisioni dello scheduler

Stati -- *Dead*

- Il modo normale di terminare per un thread è di ritornare dal **run()**
- È possibile bloccare un thread con **stop()** (comunque deprecato in Java2)

Stati -- *Blocked*

- L'esecuzione del thread è bloccata
 - In questo stato lo scheduler non lo considera tra i thread eseguibili
- Tipicamente un thread si può bloccare per vari motivi
 - Chiamata a **sleep()**
 - L'esecuzione è stata sospesa con **wait()**. Verrà ripresa con l'esecuzione di **notify()** o **notifyAll()** che vedremo
 - Il thread è in attesa di qualche operazione di I/O
 - Il thread sta cercando di accedere a un metodo **synchronized** o a una sezione critica

Cooperazione tra thread

- Oltre a possibili collisioni che possono accadere quando si usa multithreading, è possibile ottenere cooperazione tra i thread
 - Ad esempio un thread modifica dei valori e vuole comunicare a qualche altro thread che ha effettuato le modifiche
- Questa cooperazione si può ottenere tramite i metodi **wait()** e **notify()** della classe **Object**

Attesa e notifica

- Ad ogni oggetto Java è associato un *wait-set*: l'insieme dei thread che sono in attesa per l'oggetto
- Un thread viene inserito nel *wait-set* quando esegue **wait()**
- I thread sono rimossi dal *wait-set* attraverso le notifiche
 - **notify()**, che ne rimuove uno, e **notifyAll()**, che li rimuove tutti

Il metodo **wait()**

- Il metodo **wait()** pone in attesa un thread su un oggetto
- Un thread può invocare **wait()** su un oggetto sul quale ha il lock
 - Quindi **wait()** può essere invocato all'interno di un metodo o blocco **synchronized**
 - Altrimenti si ottiene una **IllegalMonitorStateException**
- Quando si invoca **wait()** su un oggetto
 - Il lock sull'oggetto viene rilasciato
 - Il thread va in stato *blocked*

Il metodo **wait()**

- **wait()** e **sleep()** pongono entrambe in attesa il thread invocante
- È importante notare però che **sleep()** *non rilascia* il lock sull'oggetto quando invocato
 - In altre parole, altri thread non possono accedere a metodi sincronizzati se il thread in **sleep()** ne ha acceduto uno

Il metodo **wait()**

- Ci sono due forme di **wait()**
 - La prima prende un argomento in millisecondi (come **sleep()**), che specifica la durata dell'attesa
 - La seconda, senza argomenti, continua l'attesa indefinitamente

I metodi **notify()** e **notifyAll()**

- Il metodo **notify()** invocato su un oggetto risveglia un singolo thread in attesa (*blocked*) su quell'oggetto
- Il metodo **notifyAll()** risveglia *tutti* i thread in attesa su quell'oggetto
- I thread risvegliati passano in stato *Runnable*, e devono comunque riacquisire il lock per riaccedere all'oggetto
- Come per **wait()**, un thread può invocare questi metodi su un oggetto solo se ha un lock per quell'oggetto

I metodi **notify()** e **notifyAll()**

- Un **notify()** effettuata su un oggetto su cui nessun thread è in **wait()** viene perso
- Se ci possono essere più thread in attesa usare **notifyAll()** (più sicuro, ma più inefficiente)
 - Risveglia tutti i thread in attesa
 - Tutti si rimettono in coda per riacquisire il lock
 - Solo uno alla volta riprenderà il lock

wait(), **notify()** e **notifyAll()**

- Questi metodi sono nella classe **Object**
- Il motivo risiede nel fatto che essi manipolano il *mutual exclusion lock* dell'oggetto su cui sono invocati
- Possono essere invocati solo da metodi o blocchi **synchronized**
 - Nota: **sleep()** non deve essere invocato da metodi o blocchi **synchronized** in quanto non manipola il lock

Esempio

- Come esempio della cooperazione tra thread consideriamo una istanza del problema produttore-consumatore
- Consideriamo un ristorante con un cuoco e un cameriere
- Il cameriere deve attendere che il cuoco prepari il cibo
- Quando il cibo è pronto, il cuoco notifica il cameriere che prende il cibo e torna ad aspettare
- Vediamo il codice

Esempio -- ordine

```
class Order {  
    private static int i = 0;  
    private int count = i++;  
    public Order() {  
        if(count == 10) {  
            System.out.println("Out of food, closing");  
            System.exit(0);  
        }  
    }  
    public String toString() { return "Order " + count; }  
}
```

È una classe che semplicemente conta il numero di oggetti creati

Termina il programma al decimo ordine

Esempio -- cameriere

```
class WaitPerson extends Thread {  
    private Restaurant restaurant;  
    public WaitPerson(Restaurant r) {  
        restaurant = r;  
        start();  
    }  
    public void run() {  
        while(true) {  
            while(restaurant.order == null)  
                synchronized(this) {  
                    try {  
                        wait();  
                    } catch (InterruptedException e) {  
                        throw new RuntimeException(e);  
                    }  
                }  
            System.out.println(  
                "Waitperson got " + restaurant.order;  
            );  
            restaurant.order = null;  
        }  
    }  
}
```

Con la **wait()** viene rilasciato il lock

Esempio -- cuoco

```
class Chef extends Thread {  
    private Restaurant restaurant;  
    private WaitPerson waitPerson;  
    public Chef(Restaurant r, WaitPerson w) {  
        restaurant = r;  
        waitPerson = w;  
        start();  
    }  
    public void run() {  
        while(true) {  
            if(restaurant.order == null) {  
                restaurant.order = new Order();  
                System.out.println("Order up!");  
                synchronized(waitPerson) {  
                    waitPerson.notify();  
                }  
                try { sleep(100); }  
            } catch (InterruptedException e) {  
                throw new RuntimeException(e);  
            }  
        }  
    }  
}
```

Ottiene il lock sul cameriere prima della notifica

Lo ottiene perché il cameriere lo rilascia con la **wait()**

Se l'ordine non è **null** aspetta 100 millisecondi e riprova

Esempio -- ristorante

```
public class Restaurant {  
    Order order;  
    public static void main(String[] args) {  
        Restaurant restaurant = new Restaurant();  
        WaitPerson waitPerson = new WaitPerson(restaurant);  
        Chef chef = new Chef(restaurant, waitPerson);  
    }  
}
```


