Recap su programmazione concorrente in Java

Matteo Trentin
matteo.trentin2@unibo.it

Recap: Thread in Java

```
public class Worker implements Runnable {
    ...
    @Override
    public void run() {
        ...
    }
}

...
Thread t = new Thread(new Worker());
t.start();
...
t.join();
```

"implements Runnable" o "extends Thread"?

In breve: `implements Runnable` è la scelta giusta in quasi ogni caso.

Meno in breve:

- In Java non è possibile estendere più di una classe; se usiamo `extends Thread`, non possiamo estendere altro
- In generale, un Runnable può essere usato anche al di fuori di un Thread
- Estendere `Thread` offre la possibilità aggiuntiva di fare override di metodi *specifici dei Thread* (e.g. `start`, `interrupt`)
- Questa possibilità aggiuntiva è necessaria molto raramente

Thread in Java - Multi-user Info Server

- Vogliamo estendere l'esempio precedente
- Abbiamo sempre un server che fornisce informazioni su richiesta
- Vogliamo accettare più di un client e inviare l'informazione a chi la richiede
- Vogliamo che i client possano inviare e ricevere messaggi indipendentemente

Concorrenza - Problemi - Race condition

check-then-act

```
...
if (!stack.empty()) {
    stack.pop();
}
...
```

read-modify-write

```
...
counter++;
...
```

=

```
...
int tmp = counter;
tmp = tmp + 1;
counter = tmp;
...
```

Cosa succede se più thread eseguono questi frammenti di codice?

Concorrenza - Problemi - Sincronizzazione

A volte è necessario garantire un ordine per le operazioni di più Thread:

```
...
Scanner userInput = new Scanner(System.in);
String request = userInput.nextLine();
Thread printEven = new Thread(new PrintEven(request));
Thread printOdd = new Thread(new PrintOdd(request));
printEven.start();
printOdd.start();
printEven.join();
printOdd.join();
...
```

Come facciamo a stampare correttamente la stringa `request`?

Concorrenza - Altri problemi

- **Deadlock**: i thread rimangono in attesa circolare, e non proseguono nella loro esecuzione
- **Livelock**: i thread non proseguono nella loro esecuzione, ma continuano a cercare di sincronizzarsi
- **Starvation**: un thread (o più) non riesce ad acquisire il lock, e di conseguenza non procede nella sua esecuzione

Concorrenza in Java - Object Monitor

- Ogni oggetto in Java è implicitamente dotato di un monitor
- Il monitor è composto da un *lock* e da una *condition variable* (i.e. una coda in cui i thread possono rimanere in attesa o venire risvegliati)
- Il lock viene acquisito tramite il costrutto `synchronized`
- La coda viene gestita con i metodi `wait` e `notify`/`notifyAll`

Concorrenza in Java - `synchronized` - Statement

- È possibile avere mutua esclusione per specifici blocchi di codice
- `synchronized(obj)` acquisisce il lock su `obj`, rilasciandolo al termine del blocco
- Solo un thread alla volta può entrare nel blocco
- Si ha mutua esclusione se i thread cercano di acquisire *lo stesso lock*

```
Object lock = new Object();
synchronized(lock) {
    counter++;
}
```

Concorrenza in Java - `synchronized` - Metodi

- Acquisisce il lock dell'oggetto a cui appartiene il metodo
- Il corpo del metodo viene eseguito in mutua esclusione
- Si seguono le stesse regole degli statement, ma il lock è implicitamente su `this`
- L'acquisizione del lock è astratta rispetto al codice chiamante

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

=

```
public void method() {
    synchronized(this) {
        ...
    }
}
```

Concorrenza in Java - `wait()` & `notify()`

Con statement:

```
Object lock = new Object();
synchronized(lock) {
    ...
    lock.wait();
    ...
    lock.notify();
}
```

Con metodi:

```
public synchronized void method() {
    ...
    wait();
    ...
    notify();
}
```

Quando viene chiamato `wait()`, il thread rilascia il lock dell'oggetto e si mette in attesa sulla condition variable.

Quando viene chiamato `notify()`, un thread in attesa sulla condition variable dell'oggetto viene risvegliato.

Concorrenza in Java - `wait()` & `notify()` - Note

- Il thread che viene risvegliato da `notify()` è arbitrario (il criterio dipende dall'implementazione della JVM)
- Quando un thread viene risvegliato deve comunque ottenere nuovamente il lock prima di tornare nel blocco (o metodo) `synchronized`
- `notify()` risveglia un solo thread in attesa. `notifyAll()` risveglia tutti i thread in attesa. Non è possibile imporre quale thread otterrà il lock per primo
- In generale, `notifyAll()` è più semplice da utilizzare rispetto a `notify()`

Errori - `if - wait() - else - notify()`

- Immaginiamo un oggetto con un attributo `stack`, contenente uno stack di interi
- Vogliamo permettere a più thread di accedere allo stack
- Qual è il problema?

```
public synchronized void pop() {
    if (stack.empty()) {
        wait();
    }
    stack.pop();
}
```

```
public synchronized void push(int x) {
    stack.push(x);
    notify();
}
```

Errori - `if - wait() - else - notify()`

- Thread t1 esegue `pop()` → attende
- Thread t2 esegue `push()` → risveglia t1
- Thread t3 esegue `pop()` e ottiene il lock prima di t1
- t1 esce dal blocco `if` anche se la condizione non è rispettata → crash

```
public synchronized void pop() {
    if (stack.empty()) {
        wait();
    }
    stack.pop();
}
```

```
public synchronized void push(int x) {
    stack.push(x);
    notify();
}
```

Errori - Deadlock con `wait()` & `notify()`

- Immaginiamo lo stesso stack, con l'aggiunta di un attributo `readOnly`
- Lo stack può essere modificato solo se `readOnly` è falso
- `readOnly` può essere cambiato con il metodo `setReadOnly()`
- Qual è il problema?

```
public synchronized void pop() {
  while (stack.size() == 0 || readOnly){
    wait();
  }
  stack.pop();
}

public synchronized void push(int x) {
  while (readOnly) {
    wait();
  }
  stack.push(x);
  notify();
}

public synchronized void setReadOnly(
     boolean b) {
    this.readOnly = b;
    if (!this.readOnly) {
        notify();
    }
}
```

Errori - Deadlock con `wait()` & `notify()`

- Thread t1 esegue `setReadOnly(true)`
- Thread t2 esegue `pop()` → attende
- Thread t3 esegue `push()` → attende
- Thread t4 esegue `setReadOnly(false)` → sveglia t2
- t2 non può comunque proseguire perché lo stack è vuoto → attende
- Nessun thread prosegue

```
public synchronized void pop() {
  while (stack.size() == 0 || readOnly){
    wait();
  }
  stack.pop();
}

public synchronized void push(int x) {
  while (readOnly) {
    wait();
  }
  stack.push(x);
  notify();
}

public synchronized void setReadOnly(
     boolean b) {
    this.readOnly = b;
    if (!this.readOnly) {
        notify();
    }
}
```

Errori - Deadlock con `synchronized`

```
public void increaseTwoCounters(Counter c1, Counter c2) {
    synchronized(c1) {
        synchronized(c2) {
            c1.increase();
            c2.increase();
        }
    }
}
```

Qual è il problema?

Errori - Deadlock con `synchronized`

```
public void increaseTwoCounters(Counter c1, Counter c2) {
    synchronized(c1) {
        synchronized(c2) {
            c1.increase();
            c2.increase();
        }
    }
}
```

```
Counter c1 = new Counter();
Counter c2 = new Counter();
Thread t1 = new Thread(() -> {
    increaseTwoCounters(c1, c2);
});
Thread t2 = new Thread(() -> {
    increaseTwoCounters(c2, c1);
});
t1.start();
t2.start();
```

Errori - Deadlock con `synchronized` - 2

Thread t1:

```
synchronized(lock1) {
    synchronized(lock2) {
        ...
        wait();
        ...
    }
}
```

Thread t2:

```
synchronized(lock1) {
    synchronized(lock2) {
        ...
        notifyAll();
        ...
    }
}
```

Qual è il problema?

Errori - Deadlock con `synchronized` - 2

Thread t1:

```
synchronized(lock1) {
    synchronized(lock2) {
        ...
        wait();
        ...
    }
}
```

Thread t2:

```
synchronized(lock1) {
    synchronized(lock2) {
        ...
        notifyAll();
        ...
    }
}
```

`wait()` rilascia il lock solo sull'oggetto `lock2`, quindi `lock1` rimane bloccato.

Se t1 entra per primo nel blocco `synchronized` più esterno, t2 non riuscirà mai a chiamare `notifyAll()`.

Errori - Uso scorretto `synchronized` e lock

Thread t1:

```
Object lock = new Object();
for (int i = 0; i < 100; i++) {
    synchronized(lock) {
        counter++;
    }
}
```

Thread t2:

```
Object lock = new Object();
for (int i = 0; i < 100; i++) {
    synchronized(lock) {
        counter++;
    }
}
```

Supponiamo che `counter` sia condiviso tra i due thread. Qual è il problema?

Concorrenza in Java - Producer/Consumer Info Server

- Vogliamo (di nuovo) estendere l'esempio precedente
- Sempre un server, sempre multipli client
- I client ora vogliono poter inserire (o rimuovere) informazioni
- Possibili problemi?