

# Example03

## Table of contents

- [Locks e conditions](#)
- [`it.unipr.informatica.concurrent.locks`](#)
  - [`Lock.java`](#)
  - [`Condition.java`](#)
  - [`ReentrantLock.java`](#)
- [`it.unipr.informatica.concurrent`](#)
  - [`ArrayBlockingQueue.java`](#)
- [Example03](#)

## Locks e conditions

Una sezione critica in JAVA è identificata sempre da un blocco synchronized, dandoci il vantaggio che tutto è molto più vincolato.

Il concetto di **lock** è come fare un passo indietro rispetto al blocco di sincronizzazione, fintanto che lo facciamo esplicito:

- **acquisizione** del lock e **sblocco** del lock per l'accesso alla risorsa;
- hanno interfaccia;
- sono come un blocco synchronized ma la probabilità d'introdurre bug è esageratamente più alta.

Le **condizioni** sono strettamente legate ai lock, per evitare **race conditions**.

Fissato un lock sarà possibile fornire condizioni legate allo stesso.

Il lock viene costruito chiamando `lock()` e nel momento in cui non ci serve più facciamo `unlock()`. Per implementare la nostra interfaccia, andiamo nel solito package `concurrent.locks`.

### `it.unipr.informatica.concurrent.locks`

#### `Lock.java`

```
package it.unipr.informatica.concurrent.locks;

public interface Lock {
    public void lock();

    public void unlock();

    public Condition newCondition();
}
```

Se il `lock` è già stato acquisito, non mi fermo: chiamiamo questo `ReentrantLock`.

La differenza è che il `lock` normale non rientra nella sezione critica e non si rimettono in campo

fintanto che il thread non ha tempo di attesa determinato.  
Sicuramente più delicato da maneggiare.

## Condition.java

Se la condizione è vera, viene generata la segnalazione e quelli in attesa sulla condizione si sbloccano e solo 1 continua la sua esecuzione.

Lo stesso modo della sezione critica ma usiamo invece gli oggetti per farlo.

```
package it.unipr.informatica.concurrent.locks;

public interface Condition {
    public void await() throws InterruptedException;

    public void signal();

    public void signalAll();
}
```

## ReentrantLock.java

Contiamo quante volte il lock viene acquisito e possiamo fare `unlock()` sul mutex, ovvero sbloccare risorse, soltanto quando il numero di `lock()` corrispondenti è uguale. Seguendo il ragionamento, il thread `owner` che gestisce il lock è `this`, quello originale.

```
package it.unipr.informatica.concurrent.locks;

public final class ReentrantLock implements Lock {

    // quale thread è owner del lock
    private Thread owner;

    // contiamo gli accessi a lock e unlock
    private int counter;

    // per fare effettivamente la sincronizzazione
    private Object mutex;

    // costruttore
    public ReentrantLock() {
        this.owner = null;
        this.mutex = new Object();
        // non deve mai scendere sotto 0
        this.counter = 0;
    }

    // ...
}
```

Nella funzione `lock()` creiamo una sezione critica.

Anche solo il verificare se siamo owner o meno del lock, necessita l'ingresso in sezione critica (altrimenti verificherebbero che l'owner ormai cambiato).

Verifichiamo se possiamo andare avanti e se possiamo rilasciamo la sezione critica:

- se esiste un owner del thread che non è `currentThread()` ci blocchiamo;
- se `owner == null` oppure `owner == currentThread()` allora procediamo dicendo che il thread corrente è owner.

La `mutex.wait()` verrà notificata dalla `notify()` della `unlock()`.

Una `lock()` è entrata e quindi facciamo `counter++`.

```
// ...
@Override
public void lock() {
    // riferimento all'esecutore attuale
    // oggetto in memoria
    Thread currentThread = Thread.currentThread();

    synchronized (mutex) {
        if (counter < 0)
            throw new IllegalMonitorStateException("counter < 0");

        while (owner != null && owner != currentThread)
            try {
                mutex.wait();
            } catch (InterruptedException interruptedException) {
                throw new
IllegalMonitorStateException("interrupted");
            }

        if (owner == null)
            owner = currentThread;

        counter++;
    }
}
// ...
```

La `unlock()` apre una sezione critica, in modo da verificare col `mutex` se possiamo rilasciare il `lock`.

Se non siamo gli `owner` del thread con cui stiamo avendo a che fare, dobbiamo bloccarci e lanciare una `IllegalMonitorStateException` (usata per i mutex).

Se per qualche strano motivo il counter va sotto 0, lanciamo una `IllegalMonitorStateException` (se accadesse overflow per esempio, counter diventerebbe negativo ma noi lo noteremmo).

Se lo stato attuale del `lock` è coerente con le aspettative, ovvero il contatore è a +0, lo decrementiamo con `counter--`.

Controlliamo: se il contatore è a 0, allora effettivamente il `lock` viene rilasciato e non ne siamo più i padroni e possiamo fare il risveglio.

```

// ...
@Override
public void unlock() {
    synchronized (mutex) {
        if (owner != Thread.currentThread())
            throw new IllegalMonitorStateException
                ("owner != Thread.currentThread()");

        if (counter ≤ 0)
            throw new IllegalMonitorStateException
                ("counter ≤ 0");

        counter--;

        if (counter = 0) {
            owner = null;

            mutex.notify();
        }
    }
}
// ...

```

Costruiamo un `lock`, sul quale costruiamo  $n$  `condition` e se vogliamo metterci in attesa della stessa, facciamo `condition.wait` e una volta sbloccata e lock riacquisito, avremo `unlock`. Per costruire le condizioni, usiamo una inner class in grado di accedere allo stato del contenitore.

```

// ...
@Override
public Condition newCondition() {
    return new InnerCondition();
}

private class InnerCondition implements Condition {
    private Object condition;

    private InnerCondition() {
        this.condition = new Object();
    }
}
// ...

```

In modo atomico, nella sezione critica, deve rilasciare la sezione critica e mettersi in attesa. La `wait()` di `Object` funziona pressapoco nello stesso modo se non fosse per il fatto che lo fa sull'unico oggetto (quello su cui invochiamo), se vogliamo 2 oggetti, uno sulla sezione critica e uno sulla condizione, non possiamo limitarci a `wait()` siccome lo può fare su uno solo.

Se la `unlock()` va a buon fine, vuol dire che eravamo gli `owner` del `lock`, altrimenti lanciamo eccezione.

```

// ...
@Override
public void await() throws InterruptedException {
    unlock();

    synchronized (condition) {
        condition.wait();
    }

    lock();
}
// ...

```

Siccome dobbiamo lavorare su `owner`, apriamo sezione critica.

Lanciamo la solita eccezione dopo la verifica, nel caso non fossimo gli `owner` (gli unici che possono fare `unlock`).

```

// ...
@Override
public void signal() {
    synchronized (mutex) {
        if (owner != Thread.currentThread())
            throw new IllegalMonitorStateException
                ("owner != Thread.currentThread()");
    }
    // fatto su oggetto private per ogni condition
    // ci dice su chi fare wait e notify
    synchronized (condition) {
        condition.notify();
    }
}
// ...

```

L'interfaccia `Condition` richiede d'implementare `signalAll()`, anche se è la stessa identica cosa della `signal()` fatta eccezione per la `notifyAll()`.

```

// ...
@Override
public void signalAll() {
    synchronized (mutex) {
        if (owner != Thread.currentThread())
            throw new IllegalMonitorStateException
                ("owner != Thread.currentThread()");
    }

    synchronized (condition) {
        condition.notifyAll();
    }
}
}
}

```

# it.unipr.informatica.concurrent

Costruiamo una seconda implementazione della `BlockingQueue.java` vista in [Example02](#).

## ArrayBlockingQueue.java

Costruiamo un array partendo dalla lunghezza che ci viene fornita.

Nel costruttore controlliamo che sia valida la lunghezza sia nella norma, memorizziamo `size` perché prima o poi ci servirà (verifica blocco della `put()`), costruiamo l'array e inizializziamo `count` per memorizzare il numero di elementi in coda. Due indici e posizioni:

- `in`, su cui faremo la prossima `put()`;
- `out`, da cui andremo a fare la prossima `take()`.

Gestiamo la sincronizzazione con:

- `isNotEmpty` se la coda non è vuota e quindi possiamo fare `take()`;
- `isNotFull` se la coda non è piena e quindi possiamo fare `put()`.

```
package it.unipr.informatica.concurrent;

import it.unipr.informatica.concurrent.locks.Condition;
import it.unipr.informatica.concurrent.locks.Lock;
import it.unipr.informatica.concurrent.locks.ReentrantLock;

public class ArrayBlockingQueue<T> implements BlockingQueue<T> {
    private Object[] queue;
    private int in, out;
    private int count, size;
    private Lock lock;
    private Condition isEmpty, isNotFull;

    public ArrayBlockingQueue(int size) {
        if (size < 1)
            throw new IllegalArgumentException("size < 1");
        this.size = size;
        this.queue = new Object[size];
        this.in = 0;
        this.out = 0;
        this.count = 0;
        this.lock = new ReentrantLock();
        this.isEmpty = lock.newCondition();
        this.isNotFull = lock.newCondition();
    }
    // ...
}
```

La `put` acquisisce il `lock` e se `count == size` si mette in attesa che qualcuno faccia `signal()` sulla condizione `isNotFull`. Con la `while` rimaniamo in attesa che le cose vadano a buon fine prima di procedere.

Se entriamo (spazio nella coda), andiamo nella prima posizione libera, scriviamo il nostro

riferimento a `Object`, incrementiamo di 1, spostiamo in avanti l'indice della fine.  
Facciamo `signal()` svegliando tutti.

```
// ...
@Override
public void put(T object) throws InterruptedException {
    if (object == null)
        throw new NullPointerException("object = null");

    try {
        lock.lock();
        while (count == size)
            isNotFull.await();
        queue[in] = object;
        ++count;
        in = (in + 1) % size;
        isNotEmpty.signal();
    } finally {
        lock.unlock();
    }
}
// ...
```

Acquisisce il `lock` facendo `await` per essere sicuri di entrare in attesa.

Prendiamo con cast a `T` il nostro oggetto in posizione finale e svincoliamo il riferimento, altrimenti rimane vincolato all'array, decrementiamo `count`.

Ad `out` incrementato, facciamo `signal()` che sveglierà uno di quelli in attesa che la coda non sia piena.

```
// ...
@Override
public T take() throws InterruptedException {
    try {
        lock.lock();
        while (count == 0)
            isNotEmpty.await();
        // suppress per evitare warnings di JAVA
        // sul tipo generico
        @SuppressWarnings("unchecked")
        T result = (T) queue[out];
        queue[out] = null;
        --count;
        out = (out + 1) % size;
        isNotFull.signal();
        return result;
    } finally {
        lock.unlock();
    }
}
// ...
```

Liberiamo gli oggetti che ci sono nel nostro array: per farlo andiamo ad assicurarci di portare in stato reset il nostro array. Segnaliamo tutti i thread per sicurezza.

```
// ...
@Override
public void clear() {
    lock.lock();
    in = out = count = 0;
    queue = new Object[size];
    // segnaliamo che la coda non è più piena
    // sicuramente non lo è ma lo facciamo lo stesso
    isNotFull.signalAll();
    lock.unlock();
}
// ...
```

Acquisisce il `lock` per la mutua esclusione e calcolato il numero di celle libere, lo ritorna. Ci dice quanto spazio c'è.

```
// ...
@Override
public int remainingCapacity() {
    lock.lock();
    int result = size - count;
    lock.unlock();
    return result;
}
// ...
...

---
La coda è vuota quando count == 0, se lo è quindi ritorniamo true.
```java
// ...
@Override
public boolean isEmpty() {
    lock.lock();
    boolean result = (count == 0);
    lock.unlock();
    return result;
}
}
```

## Example03

Facciamo un `ArrayBlockingQueue` invece che `BlockingQueue` di [Example02](#).

La costruiamo con 3 elementi per verificare la condizione di blocco per la coda piena e vuota.

```
package it.unipr.informatica.examples;

import it.unipr.informatica.concurrent.ArrayBlockingQueue;
import it.unipr.informatica.concurrent.BlockingQueue;
```



```

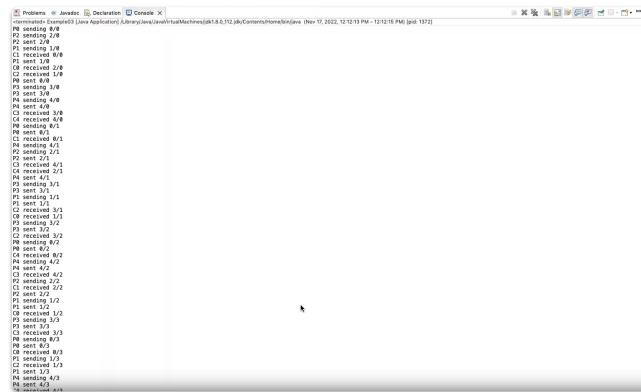
public class Example03 {
    private void go() {
        BlockingQueue<String> queue = new ArrayBlockingQueue<>(3);

        for (int i = 0; i < 5; ++i) {
            Consumer consumer = new Consumer(i, queue);
            new Thread(consumer).start();
        }

        for (int i = 0; i < 5; ++i) {
            Producer producer = new Producer(i, queue);
            new Thread(producer).start();
        }
    }

    public static void main(String[] args) {
        new Example03().go();
    }
}

```



```

Example03 [Java Application] Library\java\bin\java.exe (Nov 17, 2022, 12:12:13 PM - 12:12:16 PM) [pid: 13712]
P0 sending 0/0
P0 sending 2/0
P1 sending 1/0
C1 received 0/0
P0 sent 1/0
C0 received 2/0
C2 received 1/0
P0 sent 0/0
P0 sending 3/0
P0 sent 3/0
P0 sending 4/0
C2 received 3/0
C4 received 4/0
P0 sending 0/1
P0 sent 0/1
P0 sending 1/1
C1 received 0/1
C3 received 1/1
C4 received 2/1
P0 sent 1/1
P0 sending 2/1
P0 sent 2/1
C2 received 2/1
C0 received 3/1
P0 sent 3/1
P0 sending 3/1
P0 sent 3/1
C2 received 3/1
C0 received 4/1
P0 sent 4/1
P0 sending 4/2
P0 sent 4/2
C2 received 4/2
C0 received 5/2
P0 sent 5/2
C1 received 5/2
P0 sending 0/2
P0 sent 0/2
P0 sending 1/2
P0 sent 1/2
C2 received 1/2
C0 received 2/2
P0 sent 2/2
P0 sending 2/3
P0 sent 2/3
C2 received 2/3
P0 sending 3/3
P0 sent 3/3
C2 received 3/3
P0 sending 4/3
P0 sent 4/3
C2 received 4/3
P0 sent 4/3
P0 sending 5/3
P0 sent 5/3

```