#Appunti_17-11-22

Il tentativo è alzare il livello di astrazione, quindi avere una certa condizione rende tutto più semplice, le condizioni sono collegate alle lock, perché quando qualche thread si mette in attesa, quando si sveglia vuole essere dentro a una sezione critica, e se questo non succede si avranno un sacco di race condition;

Quindi prima cosa costruiamo i lock, a questo punto andiamo a costruire per ogni lock la possibilità di creare condition, una volta che siamo possessori del lock quindi l'owner, a questo punto possiamo metterci in attesa di eventi, se la condizione è vera non ci metteremo in attesa altrimenti il contrario.

Modifiche al file LOCK.JAVA

Abbiamo un nuovo metodo newConditions() che è il metodo che permette a un locks di costruire una *condizione*, e poi abbiamo cambiato l'operazione lock perchè non lancia più eccezioni, vuol dire che può lanciare solo eccezioni RunTime, quindi nella nostra implementazione ci metteremo in wait sul nostro oggetto che descrive il nostro lucchetto, se ci sblocchiamo in modo anomalo lanceremo un illegalMonitoStateException.

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

	public interface lock{
		public void lock();
		public void unlock();
		public Condition newCondition();
	}
```

File CONDITION.JAVA

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

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

	public void signal();

	public void signalAll();
}
```

Una Conditions è un oggetto con cui potremo fare : signal() che è l'equivalente di notify, signalAll() che è l'quivalente di notifyAll, await() è l'equivalente di wait e può lanciare un

InterruptedExcaption.

Conditions sembra che non abbia l'implementazione, questo perchè vengono costruite tramite i Lokcs, questo che senso ha? è qualcosa legato alle regioni critiche, nel momento che noi facciamo await nelle conditions viene liberata la regione critica e in modo atomico viene messo in attessa il thread; nel momento che facciamo signal() cercheremo di svegliare un thread in await, e qua si sono svegliati ma non ripartono fin tanto che il thread che ha fatto signal non libera il lock, quindi non può essere conditions senza locks, se un oggetto non può esserci senza un altro evidentemente il meccanismi di costruzione devono essere legati, infatti le classi che implementano condition vengono fatte solo tramite il lock, e qua ci vengono in aiuto le inner class che andiamo a realizzare una classe che è legata alla sua classe contenitrice, talmente legata che ne vede anche lo stato così da poterci lavorare insieme.

Aggiunte al FILE REENTRANTLOCK.JAVA

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

private class InnerCondition implements Condition{
        private Object condition;

        private InnerCondition(){
                this.condition = new Object();
        }
        @Override
        public void await() thrwos InterruptedException {
        unlock();
        synchronized(condition){
                condition.wait();
        }
        lock();
        }
        @Override
        public void signal(){
                synchronized(mutex){
                        if(owner != Thread.currentThread())
                                throw new IllegalMonitorStateException("owner !=
                                        Thread.currentThread()");
                }
        synchronized (condition){
                condition.notify();
                }
        }
```

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

Logica di utilizzo

Costruiamo un lock, sul lock costruiamo una due...conditions a questo punto se vogliamo metterci in attesta su una conditions, acquisiamo il lock(quindi lock.lock()), conditions.wait(), una volta che la condition si sblocca, ritorniamo dalla wait e abbimao in mano il lock, e sotto ci sarà lock.unlock(). per la signal e la signalAll, abbiamo il lock andiamo avanti nel nostro codice e facciamo signal e poi facciamo unlock.

In *signal()* dobbiamo verificare che chi fa signal, è anche l'owner del lock e siccome lavoriamo su owner dobbiamo lavorare nella regione critica, e se siamo l'owner siamo gli unici che possiamo fare unluck, quindi non necessita che anche le righe synchronized (condition) e condition.notify() sia all'interno della regione critica, se lo si mette si crea deadlock;
Quando arriviamo a synchronized (condition) siamo l'owner del lock, dobbiamo fare notify e lo facciamo sull'oggetto che utilizziamo come conditions, siccome un lock può avere più conditions tutte le volte che facciamo newConditions ce ne viene restituita una nuova, non possiamo avere un unico oggetto per tutte le conditions, ma avremo un oggetto conditions quello privato che ci dice : questo è l'oggetto su cui adremo a fare le wait e le notify; lo facciamo dentro ad un blocco synchronized in modo che siamo sicuri le conditions.notify non lanci un eccezione, se chiamamiamo wait notify e non siamo in una sezione critica sull'oggetto su cui stiamo chiamando il metodo abbiamo un ecezzione.

signalAll() = *signal()*

In *await()* in modo atomico deve rilasciare la sezione critica e contemporaneamente in attesa sulla conditions; per prima cosa facciamo unluck() se questa va a buon fine, vuol dire che siamo gli owner del lock, e siamo gli unici che lo rilasciamo e contemporaneamente facciamo una wait e nel momento in cui esce e andiamo a riprendere il lock, e se qualcun'altro prende il lock non fa niente prima o poi lo prenderemo anche noi.

File ARRAYBLOCKINGQUEUE.JAVA

```java
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 isNotEmpty, 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.isNotEmpty = lock.newCondition();

		this.isNotFull = lock.newCondition();
	}

	@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();
        }
}

@Override
public T take() throws InterruptedException {
        try {
                lock.lock();

                while (count == 0)
                        isNotEmpty.await();

                @SuppressWarnings("unchecked")
                T result = (T) queue[out];

                queue[out] = null;

                --count;

                out = (out + 1) % size;

                isNotFull.signal();

                return result;
        } finally {
                lock.unlock();
        }
}

@Override
public void clear() {
        lock.lock();

        in = out = count = 0;
```

```
                queue = new Object[size];

                isNotFull.signalAll();

                lock.unlock();
        }

        @Override
        public int remainingCapacity() {
                lock.lock();

                int result = size - count;

                lock.unlock();

                return result;
        }

        @Override
        public boolean isEmpty() {
                lock.lock();

                boolean result = (count == 0);

                lock.unlock();

                return result;
        }
}
```

Il lock è un ReentrantLock e su questo costruiamo due condizioni *isNotEmpty e isNotFull*.

Partiamo da *RemainingCapacity()*, questo cosa fa? acquisisce il lock, che ci serve per manipolare lo stato dell'oggetto e ci serve per farlo in mutua esclusione; una volta acquisito viene calcolato result con size - count che sono equivalenti al numero di celle e al numero di celle occupate, e la differenza ci dice il numero di celle libere.

isEmpty() ci dice se la coda è vuota, quindi quando count è uguale a zero.

Clear() deve liberare gli oggetti che sono nel nostro array, per poterlo fare acquisiamo il mutex e gli diciamo che in = out = count = 0 quindi la coda è stata sicuramente svuotata, poi facciamo
queue = new Object[size] che ha l'effetto di liberare l'array che contiene i riferimenti agli oggetti, poi facciamo unlock ma prima segnaliamo che la coda non è più piena.

Put() acquisisce il lock, se count = size si mette in attesa e aspetta che qualcuno faccia signal sulla condizione *isNotFull* e la signal ad esempio viene fatta dalla *clear()* , a questo punto mettendo dentro al while questa await() rimanimao in attesa che le cose vadano a buon fine; a questo punto se entriamo e quindi siamo nella condizione che c'è spazio nella coda, andiamo a scrivere nella prima posizione libera che è identificata dall'indice in (che parte da zero) a riferimento al nostro object che ci è stato passato, incrementiamo count perchè la coda si è allungata di uno, poi spostiamo int di uno modulo size, cioè facciamo in modo di andare avanti e quando arriviamo in fondo ritorniamo all'inizio, a questo punto facciamo signal() e svegliera tutti quelli che aspettano che la coda non sia più vuota.

Take() acquisisce il lock, facciamo un await() in modo che siamo sicuri di metterci in attesa nel momento che count è zero, nel momento in cui è diverso da zero abbiamo qualcosa da prendere e andiamo a tirare fuori l'oggetto che è in posizione out, questo oggetto ci viene ritornato come riferimento a object, mettiamo il riferimento alla coda a nullo cosi lo svincoliamo, decrementiamo out perchè c'è un elemento in meno e postiamo out avanti di uno in modulo size, quindi in è più avanti di out e questo rincorre in e lo raggiunge quando count è uguale a zero, poi faccimao una singal() che sveglierà qualcuno che aspetta che la coda non sia più piena.