Concurrency with Java

Managing Threads

As explained in the FSO subject, the use of Threads can lead to Race Condition due to the simultaneous access to the same data or Object. So, in order to prevent that, Java provides many functions, some of them very similar to the methods provided by C language, and others that are a bit more special, let's take a look.

The methods that are almost the same as in `C language` are in the summary of Unit 1.

Interrupts

Before diving into concurrency, take into account that when using Threads, you have to handle the `InterruptedException`, which is thrown when a Thread is interrupted by any event. You can handle that with `Try-catch`.

In addition, if a Thread goes a long time without invoking any method that throws `InterruptedException`, you can check the state of the Thread using the `Thread.interrupted()` method, that will check if a Thread has been interrupted.

Synchronization

Synchronized methods

```
// synchronized keyword in the declaration of a method intern
// locks so that the access to the method is performed by onl

public synchronized void increment() { c++; }

// Using this establishes a happens-before relationship betwe
// calls this method before the other threads.
```

Synchronized statements

```
// Inside a method, you can also use synchronized for a set o

public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

By going a bit further, you can implement locks with synchronized statements

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();
    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Atomic Access

```
// Atomic actions are those executed in a single execution st
// Thread can alter the value before it is written
// This is performed with the volatile keyword
int volatile num = 1;
```

Liveness

Deadlocks, starvation and livelock

Deadlock → Situation in which two threads are blocked waiting for the other to exit a situation infinitely

Starvation → Situation in which a thread is unable to access regularly the data it wants, waiting indefinitely long times.

Livelock → Similar to deadlock, two threads are consequently responding to a trigger event from the other, this is repeated so many times that they are both stuck blocking each other.

Guarded Blocks

Sometimes, threads have to wait for other thread to perform some action this can be orchestrated by using `blocks`, which are implemented as:

```
while(!someBlock) {
    try{
        wait();
    } catch (InterruptedException ie) {}
}
```

Note that, instead of using an empty while loop, we **must** use `wait()`, as this method will not waste CPU Time.

```
// On the other hand, the thread that is being waited execute
public synchronized notifyBlock() {
    someBlock = true;
    notifyAll();
}
// notifyAll() informs all waiting threads that an important
```

High level Concurrency Objects

After reviewing the Low level API for Concurrency in Java, we come up to the features provided by the `java.util.concurrent` package.

Lock Objects

Similar to the implicit locks, but they add a crucial feature, that is the ability to back out of an attempt to acquire a lock with the structure `tryLock`. Also, the method `lockInterruptibly`, allows to back out an attempt if a thread sends an interrupt before the lock is acquired.

ReentrantLock

→ Provided the methods `isHeldByCurrentThread()` and `getHoldCount()`, we can check the ownership of a lock.

```
// This example shows how to use the ReentrantLock objects
class X {
    private final ReentrantLock lock = new ReentrantLock();

    public void m() {
        lock.lock();
        try {
            //...
        } finally {
            lock.unlock();
        }
    }
}
```

Executors

Interface used for creating new Threads and execute them, or also use an existing working thread to run it, or to place the new thread in a queue waiting for a thread to be available.

```
// Instead of
(new Thread(r)).start(); //Being r a Runnable object

// Executor
e.execute(r);
```

Thread pools

By using **worker threads**, a limit is set to the amount of existing threads, which minimizes overhead problems. The most common implementation is setting a **fixed thread pool**.

Concurrent Collections

BlockingQueue

Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

Atomic Variables

Similar to synchronized access to variables, allows to declare variables that are atomic, and so, to use them, the only change is that the need of `synchronized` is removed, and instead of writing `c++; or c--;`, you will use `c.incrementAndGet(); c.decrementAndGet(); c.get();`.

Semaphores

Synchronization that set a limit to the number of threads that can access a resource.

When used with a limit of one, it can be used as a mutual exclusion lock, but the usability comes from using higher amounts.

It provides the methods `acquire() and release()`.

Cyclic Barrier

Synchronization aid that allows sets of threads to wait for each other to reach a common barrier point, useful for programs that require a set of threads to be in the more or less the same point.

```
class Solver {
  final int N;
  final float[][] data;
  final CyclicBarrier barrier;
  class Worker implements Runnable {
     int myRow;
     Worker(int row) { myRow = row; }
     public void run() {
           while (!done()) {
              processRow(myRow);
              try {
                 barrier.await();
              } catch (InterruptedException ex) {
                 return;
              } catch (BrokenBarrierException ex) {
                 return;
              }
           }
     }
  }
  public Solver(float[][] matrix) {
    data = matrix;
    N = matrix.length;



     barrier = new CyclicBarrier(N,
        new Runnable() {
            public void run() {
                  mergeRows(...);
            }
        });
        for (int i = 0; i < N; ++i)
          new Thread(new Worker(i)).start();
        waitUntilDone();
  }
}
```

Here, `done` will return true when all threads have arrived to the same barrier.

On the other hand, if you want to select which thread is the one that releases the barrier, you can use:

```
if(barrier.await() == 0) {
    doSomething();
}
```

💡 Note: Cyclic barrier will throw exceptions for all threads, even if it is just one which failed.

Cyclic Barriers can be reset

Countdown Latches

Synchronization aid that allows a set of threads to wait until a set of operations performed by another set of threads is completed, very similar to Cyclic Barriers, but without the possibility to reset the counter.

```
class Driver{
    void main() throws InterruptedException {
        CountDownLatch startSignal = new CountDownLatch(1);
        CountDownLatch doneSignal = new ContDownLatch(N):
        for(int i = 0; i < N; ++i) {
            new Thread(new Worker(startSignal, doneSignal)).s
        }
        doSomething();
        startSignal.countDown();
        doSomething();
        doneSignal.await();
    }
}

class Worker implements Runnable{
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch start, CountDownSignal done){
        this.startSignal = start;
        this.doneSignal = done;
    }
```

```
    public void run(){
        try {
            startSignal.await();
            doWork();
            doneSignal.countDown();
        } catch(InterruptedException ie) {}
        return;
    }

    void doWork() {...}

}
```

Exchanger

Synchronization point at which threads can exchange elements within pairs.

```
class FillAndEmpty {
  Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
  DataBuffer initialEmptyBuffer = ... a made-up type
  DataBuffer initialFullBuffer = ...

  class FillingLoop implements Runnable {
    public void run() {
       DataBuffer currentBuffer = initialEmptyBuffer;
       try {
          while (currentBuffer != null) {
              addToBuffer(currentBuffer);
              if (currentBuffer.isFull())
                currentBuffer = exchanger.exchange(currentBuffer);
          }
       } catch (InterruptedException ex) { ... handle ... }
    }
}

class EmptyingLoop implements Runnable {
   public void run() {
       DataBuffer currentBuffer = initialFullBuffer;
       try {
          while (currentBuffer != null) {
              takeFromBuffer(currentBuffer);
              if (currentBuffer.isEmpty())
                 currentBuffer = exchanger.exchange(currentBuffer);
           }
       } catch (InterruptedException ex) { ... handle ...}
   }
}

void start() {
       new Thread(new FillingLoop()).start();
       new Thread(new EmptyingLoop()).start();
}
}
```

In this example, there are two buffers that are exchanged between two threads, one thread empties the buffer received while the other thread fills the buffer it receives.