JAVA CONCURRENCY

- Pablo Pastor Martín
- Jorge Sierra Acosta

Github Repo

INDEX

- **Threads**
  - Process & Threads
  - Runnables
  - Interruptions & pauses
- **Synchronization**
  - Synchronized
  - Liveness
- **Concurrency Objects**
  - Locks
  - Executors
  - Atomic

Threads

How to use them and why.

Concurrent Software

- Run several programs or parts of a program in parallel.
- Java includes several tools to develop concurrent software.

Processes vs Threads

- Basic units of execution:
    - Processes
    - Threads
- Present even on monocore systems

We are going to be focus on threads, but first, we’re going to introduce processes.

Processes

- Independent & isolated.
- Own set of basic runtime resources.
- Programs use one or more.

Threads

- Lightweight process.
- Shared data (same process).
- Own call stack.
- Default is one thread.

Threads advantages & disadvantages

- Parallel processing.
- Asynchronous behaviour.
- Faster execution if we subdivide the task in subtasks.

- Dividing the tasks may be complicated.
- Thread interference.
- Memory consistency.
- Thread contention.

Examples:
When should we use threads.

Threads (Java Object)

- Each thread is an instance of the class `Thread`.
- Threads can have different priorities.
- Override the `Thread.run()` method.

Runnables (Java interfaces)

- Implements the `run()` method.
- The `Thread` constructor can receive a `Runnable` object.
- Runnables use the composition relationship.

```java
public class ExThread extends Thread {
    public void run() {
        /* … */
    }
}

public class ExRunnable implements Runnable {
    public void run() {
        /* … */
    }
}

(new ExThread()).start();
(new Thread(new ExRunnable())).start();
```

Runnables & lambda expressions

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};
```

Examples:
Runnable & Thread.

Pausing a thread

- `sleep()`.
- Not precise.
- Only a interrupt can stop the sleep.

Interrupts

- Indication to stop and something else.
- Launched with the `Thread.interrupt()` method.

Supporting the interruption

- Invoking methods that throw `InterruptedException`.
- Checking the interrupt flag `Thread.interrupted()`.

Interrupt flag

- Setting the flag: `Thread.interrupt()`.
- Checking the flag: `Thread.isInterrupted()`.
- Clearing the flag:
  - `Thread.interrupted()`.
  - Any method that exits by throwing an `InterruptedException`.

Examples: Interrupts.

Joins

- Blocks the current Thread until another Thread ends.
- It’s possible to specify a maximum waiting time.
- Not precise.

```
Thread t1();
Thread t2();

t1.start();
t2.start();

t1.join();
t2.join(2000);
```

```java
public class ExampleMessages {
    public static void main(String args[]) {
        String importantInfo[] = {
            "Mares eat oats", "Does eat oats",
            "Little lambs eat ivy", "A kid will eat ivy too"
        };

        for (int i = 0; i < importantInfo.length; ++i) {
            try {
                Thread.sleep(4000);
            } catch (InterruptedException e) {
                return;
            }
            System.out.println(importantInfo[i]);
        }
    }
}
```

Examples:
Threads on the first example.

Synchronization

Thread Interference

- Two operations from different threads on the same data interleave.
- Causes non deterministic behavior.
- Difficult to detect and fix.

Memory inconsistency

- When different threads have inconsistent views of what should be the same data.
- Solved with the *happens-before* relationship.
  - Assure that writes by one statement are visible to another.
  - `Thread.start()` & `Thread.join()` create this relationship.

Synchronized Methods

- It's not possible for two invocations of synchronized methods of the same object to interleave.
- It automatically creates a *happens-before* relationship.

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

Synchronized Methods

- It's not possible for two invocations of synchronized methods of the same object to interleave.
- It automatically creates a *happens-before* relationship.

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

If an object is shared, all reads or writes to that object's variables must use synchronized methods

(**Exception**: Constants (`final`))

Intrinsic Locks and Synchronization

Locks enforce exclusive access to an object's state and establish happens-before relationships. Every object has its own intrinsic lock:

- Instances
- Class objects
- ...

If a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns.

Intrinsic Locks and Synchronization

Synchronized statements

- Improve concurrency with fine-grained synchronization.
- When we use them, we must specify the lock's object.

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

Examples:
Adder problem.

Liveness

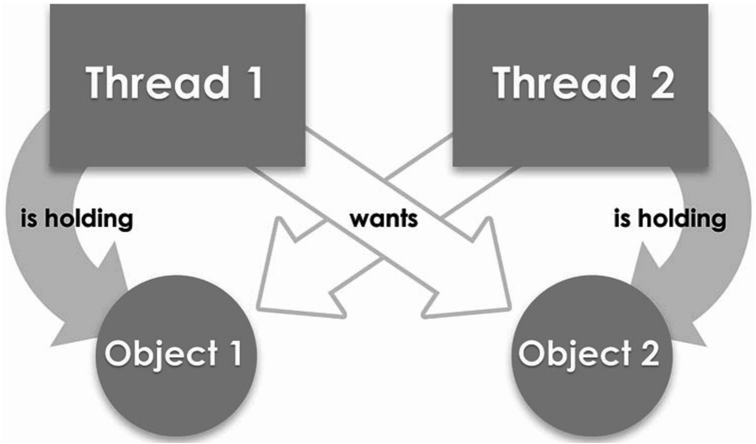
- Ability of a concurrent application to be executed in a timely manner.

- It’s most common problems are:
  - Deadlock
  - Starvation
  - Livelock

Liveness

Deadlock

- Situation where two or more threads are blocked forever, waiting for each other.
- Under some circumstances, this kind of error are extremely likely to happen.



Examples:
Deadlock problem.

Liveness

Starvation & Livelock

- Less common.
- **Starvation:**
  - A thread is unable to gain regular access to shared resources because they are locked by another thread.
- **Livelock:**
  - Threads are scheduled but are not making forward progress because they are continuously reacting to each other's state changes.

Guarded Blocks

- Most common coordination idiom between threads.
- Such a block begins by polling a condition that must be true before the block can proceed.

- Its way of work are based on:
  - A waiting thread.
  - A thread that notifies that thread.

Guarded Blocks

```
public synchronized void guarded() {
    while(!condition) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
}
```

```
public synchronized notifyingMethod() {
    condition = true;
    notifyAll();
}
```

`Volatile` variables

- Its use reduces the risk of memory consistency errors.
- Any write to a `volatile` variable establishes a happens-before relationship with subsequent reads of that same variable.
- Changes to a `volatile` variable are **always** visible to other threads.

Examples:
Guarded Blocks.

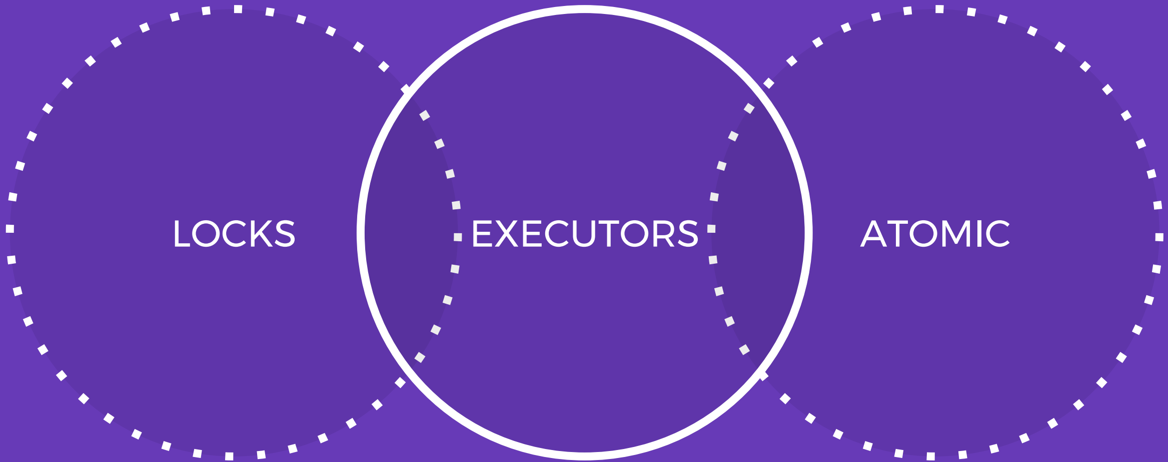
Immutable Objects

- Its state cannot change after its creation.
- Extremely useful in concurrent applications
  - They cannot be corrupted by thread interference.
  - They cannot be observed in an inconsistent state.
  - The need of creating new objects instead of updating a previous one doesn’t have any impact on performance.

Immutable Objects

- Don't provide *setter* methods
- Make all fields `final` and `private`.
- Don't allow subclasses to override methods (`final` classes).
- If the instances fields include references to mutable objects:
  - Do not prive methods to modify them.
  - Don't share references to them.

HIGH LEVEL **CONCURRENCY OBJECTS**



`java.util.concurrent`
(Java 5.0)

Lock Objects

- Based on the Lock interface.
- Similar to the implicit locks used by `synchronized`.
- Only one thread can own a Lock object at a time.
- Ability to back out of an attempt to acquire a lock.

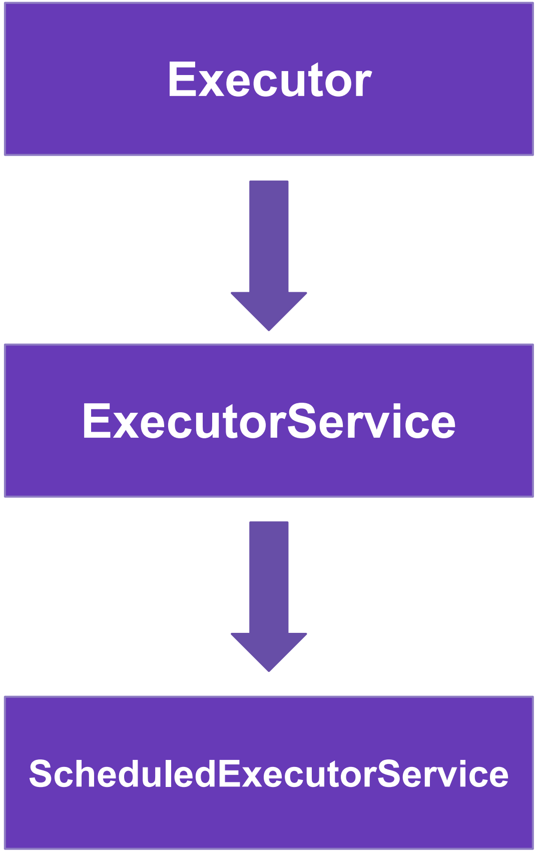
Lock Objects

Interface

`void lock()`	Acquires the lock
`void lockInterruptibly()`	Acquires the lock unless the thread is interrupted
`Condition newCondition()`	Returns a new condition, bounded to this lock
`boolean tryLock()`	Acquires the lock only if it is free at the time of invocation.
`boolean tryLock(long time, TimeUnit unit)`	Acquires the lock if it is free within the given waiting time and not interrupted.
`void unlock()`	Releases the lock.

Executors (Objects)

- Separate thread management and creation from the rest of the application.
- Interfaces:
  - **Executor**: Supports launching new tasks.
  - **ExecutorService**: Adds new features to the previous one, that help manage the lifecycle, of the individual and of the executor itself.
  - **ScheduledExecutorService**: Supports future and/or periodic execution of tasks.



Atomic Access

- An *atomic action* is one that happens all at one. It cannot stop in the middle.
- Some actions that are atomic:
  - Reads and writes for reference and primitive variables.
  - Reads and writes for all variables declared `volatile`.

- Atomic actions can be used without fear of thread interference.
- Memory inconsistency errors are possible if variables are not `volatile`.

Atomic Variables

- They use *Atomic Actions*.
- No need for the `synchronized` keyword or locks.
- Faster than locks.
- Atomic objects can be accessed with *getters* and *setters*.
  - `get()`
  - `set(int value)`
  - `getAndSet(int value)`
  - `incrementAndGet()`
  - `lazySet(int value)`
  - `compareAndSet(int expected, int value)`

LongAdder (Atomic)

- Alternative to AtomicLong.
- Preferable over atomic numbers types when **updates from multiple threads are more common than reads**.
- Higher memory consumption.
- Reduce contention over threads.

LongAccumulator (Atomic)

- Generalized version of LongAdder
- Given a function to combine values

Concurrent Collections

Collections that help avoid Memory Consistency Errors by defining *happens-before* relationships.

- BlockingQueue
- ConcurrentMap
- ConcurrentNavigableMap

BIBLIOGRAPHY

- Java Concurrency tutorial.
- Winterbe java Concurrency.
- fahd.bloc Concurrency
- Java Concurrency / Multithreading Tutorial
- Java concurrency (multi-threading)

THANKS!

Any questions?

You can find us at

alu0100896282@ull.edu.es &

alu0100890839@ull.edu.es