Concurrent programming

1 Concurrency

Every program that we have written up to now consisted of one single thread. The flow of execution started at the beginning of the `main` method and continued line-by-line up the last statement of such method. Method calls were executed orderly, line-by-line, from the beginning to the end. When methods called other methods, called methods were executed to the end (or until an exception was thrown) before the calling method continued its execution. When an exception was thrown, the flow jumped downwards until an adequate `catch` block was found or until the the end of the method; if the exception found the end of the method, it continued looking for a `catch` block on the calling method, and continued moving up on the stack until it found it (maybe up to, and out of, the `main` method).

In any case, there was one and only thing to do at any point in time. Our programs have been *deterministic*. If we knew what statement had just been executed by the computer, we knew which statement was going to be executed next (and with which values, etc).

However, we know that computers can do several things at the same time. The computer you are working on now is able to show you this document while accepting input from your keyboard and mouse, browsing the internet, keeping several services active in the background, and performing several other tasks at the same time. Computers are able to do this by switching tasks on the microprocessor (some nanoseconds of reading data from the network, then a bit of reading keyboard input, then painting the screen, then a bit of network again. . .), but the switching happens so fast that it gives the impression to humans that all tasks happen simultaneously. Additionally, now that multi-processor computers and multi-core processors have become the norm, computers are able to actually do more than one task at the same time, one per microprocessor or core (plus they still switch tasks fast).

In this section we are going to learn the basics of concurrent programming, enabling our programs to do more than one thing at the same time. This is intended to be only a very basic introduction. Concurrent programming is a very complicated topic, difficult even for seasoned programmers[1].

1.1 Why concurrency?

Concurrent programming is harder than simple deterministic programming, but it bears many important benefits, including:

[1]Concurrent programming is so difficult that several programming languages have been created with the explicit goal of easing the creation of concurrent programs. Examples range from educational languages like SR to production-oriented languages like Erlang, Scala, Go, or Clojure (and arguably Java). Scala and Clojure programs run on the Java Virtual Machine and can interact easily with Java programs.

Performance. If your task can be separated into many subtasks, and each subtask can be done in parallel in different processors or machines, the whole task may be completed faster. This is a very common approach to perform complicated scientific and engineering calculations that take long to complete (e.g. weather simulations).

Responsiveness. Concurrent programs are able —by definition— to do several things at the same time. This enables concurrent programs to start long calculations in the background while still listening to user input, even if the calculation has not finished yet. Think of the Java garbage collector, for example; it is running at the same time as your programs, looking for unreachable memory to release; it does not need your program to end before memory can be recovered.

Cleaner design. Many applications require different tasks to happen at the same time: games require many enemies to move, web browsers require several images to be downloaded for visualising a web page, videoconference software encodes (outgoing) and decodes (incoming) video and sound at the same time, etc. This can be programmed as separate tasks and let the operating system (or virtual machine) to take care of the switching, which is much simpler than taking care of the switching as part of the program.

1.2 Basic concepts

1.2.1 Non-determinism

The benefits of concurrent programming come at a cost that we have hinted at already: determinism. Concurrent programs are not deterministic, we cannot tell which will be the next statement to be executed even if we know exactly which one was executed last.

This is because the machine will keep switching tasks and we *do not have any control* over that process. The machine will switch tasks depending on many different factors that we cannot control and this means that our execution flow may be stopped after one or one hundred statements have been executed, and the context may have changed when the execution is resumed. Imagine that you are going to take the last sandwich on the plate but, just before you grab it, it disappears; this means that the computer stopped you midway through your action "taking sandwich", continued executing another thread (that grabbed the sandwich and ran away), and then continued executing you... but the context had changed (the sandwich had disappeared).

1.2.2 Processes vs Threads

At this point, it is important to distinguish two basic concepts: processes and threads. A process has its own memory space (stack and heap) separated from other processes. Switching between processes is done by the operating system. A Java program runs in its own process.

Threads (sometimes called lightweight processes) share part of the same memory space. Each thread has its own stack, but they share access to the heap and can communicate inside the program[2], e.g. by accessing the same variables or objects.

Concurrent programming in Java is done by using threads. The Java Virtual Machine switches between threads, activating or deactivating them according to different factors beyond the control

[2]Modern operating systems allow for communication between processes, but that requires use of special operating system calls and is out of the scope of this section.

of the programmer. As far as the programmer is concerned, threads run in parallel at random (and variable) speeds.

1.2.3 Atomic operations

An *atomic* operation is an operation that cannot be interrupted by thread-switching. In Java, every read and every write to a 32-bit primitive type is atomic (note that this leaves out `double` and `long`). In other words, if the computer starts executing the statement `n = 0` it cannot change thread until that statement has finished.

It is important to note —especially if you have experience with C or C++— that `n++` is *not* atomic. In Java, that statement is equivalent to three statements: one read, one increment, and one write. It is possible that your thread will be interrupted after the read but before the write. If another thread interacts with `n` then, the result is unpredictable and may result in data loss.

1.3 Launching new threads

Java provides a class called `Thread` to launch and run new threads. New threads are defined by using a class that implements interface `Runnable`. This interface consists of just one simple method:

```
public void run();
```

Starting a thread with a new `Runnable` task is easy. First, we create the thread, then we start it. Assuming we have a runnable object called *myRunnable* (that implements interface `Runnable`), the code looks like:

```
Thread newThread = new Thread(myRunnable);
newThread.start();
```

From this point on, the new thread will be running in parallel with the main thread of the application (the one started by Java at the `main` method) until its `run()` method finishes[3].

1.4 The problems of concurrency

We already know that concurrent programs are not deterministic. This is a big source of bugs. A computer program is quite complex, and it is difficult to keep track of every single aspect of it. You have experienced this yourself in past weeks when your programs did not behave as expected, and you had to debug them: maybe your program was not updating a variable as you thought it was, or maybe some data was overwritten and it was not your intention. All of this can happen in multi-threaded programs, and worse. Look at the following (contrived but illustrative) code:

```
private int count = 0;
// ...
public int increment() {
   count++;
}
```

[3]There is another way of creating threads: extending class Thread instead of implementing Runnable, and then starting it directly. This approach is less flexible because Java does not allow for multiple inheritance, so it is rarely used.

```
public int decrement() {
   count--;
}
public int getCount() {
    increment();
    decrement();
    return count;
}
```

It seems obvious that any invocation of `getCount()` method will return exactly the value of `count`, right? Unfortunately, this is the kind of "evident yet false" behaviour that is common when multithreaded programs are not studied carefully.

Let's imagine that we have two threads, T1 and T2, running in parallel. (Read the following lines carefully, its gets messy).

- T1 is executing at the moment, and T2 is "dormant".
- T1 calls method `getCount()` when `count` is 0, calls `increment()` and increases the value to 1, then it is interrupted.
- T2 resumes its execution, which includes calling `getCount()` (`count` is now 1), and then calls `increment()`, and increments `count`, that is now 2.
- T2 is interrupted and T1 resumes execution. It finishes with `increment()`, returns to `getCount()` and calls `decrement()`, decreasing the value of `count` to 1, then returns to `getCount()` and returns 1 (not 0!).
- T2 continues execution and finishes `getCount()` returning a value of 0 (not 1!).

If we had more than two threads, the situation could be even more complicated and unpredictable. For example, a third thread may have called `increment()` and `decrement()` while T1 and T2 had not finished their execution of `getCount()`. Keep also in mind that interruptions and thread-switching could have happened in between the `count++` and `count--` statements because they are not atomic despite the fact that they are just one short line.

The moral of the story is: concurrent programming is difficult, and non-deterministic; you have to be very careful or you will get it wrong.

1.5 Synchronisation

Computer programs are lists of instructions to make computers process data in a reliable (and fast) way. If we want our programs to be reliable even when using non-deterministic threads we need to make sure threads are synchronised in some way. The most common way of achieving synchronisation is by means of *locks*. By acquiring and releasing locks, we can make sure that our threads do not overlap with bad results.

We can think of thread locks as locks on doors: door locks control access to rooms and thread locks control access to pieces of code. When a door is open (lock free) any person (thread) can open the door and enter (start executing some code), closing the door behind (acquiring the lock). Any other persons that tries to open the door (tries to acquire the lock) while another person has locked it (while another thread has the lock) will be blocked, and will have to wait. As soon as the person that locked the door opens it and comes out (thread releases the lock) another person can enter (another thread can acquire the lock and start executing the code).

It is important to note that threads do *not* make an ordered queue in front of a lock like good British citizens. Quite the opposite: when a lock is released, any thread that was waiting for the lock may acquire it. You can think of threads as noisy foreigners, pushing each other to be in front of the door when it opens, ready to rush in as soon as the person inside gets out.

In Java, locks are acquired (and released) easily by using one of the few keywords we have not seen yet: `synchronized`. This keyword can be used on methods or on fragments of code.

1.5.1 Synchronising fragments of code

When used on a fragment of code, it must specify the object from which to acquire the lock. Every Java object contains a lock inside, so every Java object can be used for this purpose. If the lock is available, it will be acquired and the code will execute. At the end of the scope, the lock will be released. If the lock is not available (another thread has it) the thread will block, waiting for the lock to be available.

```
1    private Object obj = new Object();
2    // ...
3    public int getCount() {
4        synchronized (obj) {
5            increment();
6            decrement();
7            return count;
8        }
9    }
```

When T1 enters the `synchronized` block (lines 4–8), it may be interrupted because the machine switches to T2. Thanks to the use of locks, this will not pose a problem when T2 executes `getCount()` because T2 will try to acquire the lock of object `obj` (line 4), realise it is already acquired by another thread, and get blocked. When the machine switches back to T1 at an undetermined time in the future, it will continue with its execution until it returns a value from `getCount()` (line 7). The lock is released when a thread reaches the end of a `synchronised` block, either explicitly (in the example, by reaching line 8) or implicitly, either by returning a value (as in this example) or by throwing an exception.

When the lock on `obj` is released, T2 (or any other thread waiting for the same lock, *maybe in other methods of the same object*) will be able to acquire it and continue with their execution.

Note that we could have used *any* object for locking this block of code. For example, if `count` was an `Integer` instead of an `int`, we could have used its lock to synchronise the block. We could have also acquired the lock of `this` object (the object where the code is being executed). Note as well that the object from which we use **the lock must be a field** and cannot be a local variable because every thread has its own stack, so the following code will not work as intended:

```
public int getCount() {
    Object obj = new Object(); // Every thread has its own copy
    synchronized (obj) {       // of object obj (and its lock),
        increment();           // so they do not block each other
```

```
            decrement();
            return count;
        }
    }
```

1.5.2 Synchronising full methods

The second way in which `synchronized` can be used is to synchronise full methods, as in this example:

```
    public synchronized int getCount() {
        increment();
        decrement();
        return count;
    }
```

When it is used in this way, the thread trying to execute the synchronised method will acquire the lock of `this` object. In a way, this is similar to using `synchronized(this)` on the full code of the method, but the effect is a bit different.

- Synchronising the whole method has two benefits: it is simpler (less lines of code) and clearer (the `synchronized` will appear in the documentation of your class, making it explicit to other programmers).
- Synchronising a fragment has two benefits: it allows you to synchronise only part of the method (finer-grained synchronisation) and it can use different objects for locks, not just `this`.

Both aspects may be important for making sure that your program does not block, depending on the application (see Section 1.5.3).

Note that `synchronized` does *not* make your block of statements *atomic*. Your thread can still be interrupted an arbitrary number of times while it is executing a synchronised block. The only thing that `synchronized` will do for you is making sure that other threads do not enter the same block of statements until the current thread has finished with it (either by executing all statements or by throwing an exception) and released the lock.

A note on terminology

Locks are sometimes called *mutex*, because they provide *mut*ually *ex*clusive access to a resource. *Semaphores* are a special type of lock that allows several threads to gain access to a resource. Semaphores can be represented as an integer number, and every thread that gets access to the resource decreases it. When it gets to zero, the next thread will need to wait until the semaphore raises again. A lock can be viewed as a semaphore of value 1.

1.5.3 Why not make everything synchronised?

Synchronising access to resources prevents harmful overlapping of threads using the same resource, but we cannot make every single method in our program `synchronized`. First of all, synchronisation comes at a performance cost. Checking, acquiring, and releasing locks takes time. Careless use of locks all over our code will make it run slower. However, this is not our main concern.

The most important problem of careless synchronisation is that our program may block. If thread A has acquired lock L1 and is waiting for lock L2, while thread B has acquired lock L2 and is waiting for L1, our program cannot continue. This is called a *deadlock*, and is one of the most common problems with concurrent programs (the other being race conditions leading to harmful overlapping of threads accessing the same resource). Remember that we said that thread locks are like door locks? Imagine that Andrew stops before the door to let Bob pass first, and the same time Bob also stops before the door to let Andrew pass first; as each of them is waiting for the other to act, none of them do: this is a deadlock.

There is no easy way to avoid deadlocks in our code: we must be really careful in our design. Careful design will prevent deadlocks and also other less common but equally bad situations: starvation and livelock.

Starvation occurs when one or more of our threads never execute and/or have access to some resource they need. This may happen for example if one thread acquires a lock but never releases it; threads waiting for that lock will remain “dormant” forever. A *livelock* is similar to a deadlock in the sense that two or more threads are blocking each other, but the difference is that the program is still running, only not making any progress. Imagine that Andrew and Bob meet in the middle of the corridor; Andrew moves to the left to let Bob pass, and Bob moves to the right for the same reason: they are blocked again; then Andrew moves to the right and Bob moves to the left, blocking each other again; and this continues forever.

Finally, a *race condition* happens when the outcome of our program is not reliable and depends on the specific order in which the threads execute: sometimes it may give the right result, sometimes it may lose data, sometimes it may block. This is always the result of a poor understanding of the interaction of the threads and/or poor synchronisation. It is extremely difficult to debug a program with race conditions, as they may manifest only with some operating systems, and/or some versions of the Java Virtual Machine, and/or in the presence of some specific data input, and/or some other apparently harmless factor in the environment. May you never find yourself debugging a race condition the day before a deadline.

1.5.4 Sleeping, waiting, and notifying

Sometimes a lock is not what we need to synchronise our threads; sometimes we want one of your threads to wait until one condition is true (i.e. set as true by another thread). For example, the thread that takes our mail should not read from the inbox until some other thread has left any mail on it. The block of code that comes after such a condition is sometimes called a *guarded block*, because it is guarded/protected by the condition. We could implement guarded blocks with a simple `while` loop, as shown below:

```
while (!conditionMet) {
    // empty loop -- no code
}
// Guarded block comes here...
```

The code above is really wasteful. The computer will spend a lot of CPU time doing a lot of empty loops. It would be better to let the computer make a short pause and then continue with the execution. We can do this by using the static method `sleep()` and providing the length of the pause in milliseconds.

```
while (!conditionMet) {
    try {
        Thread.sleep(1000); // sleep 1000ms = 1 second
    } catch (InterruptedException ex) {
        // Nothing to do in this case, just sleep less...
    }
}
// Guarded block comes here...
```

(Methods that put a thread on hold, like `Thread.sleep()`, can be interrupted. If that happens, an `InterruptedException` will be thrown. This is a checked exception and, therefore, must be caught).

This is better than the former version, but it still checks every single second. Maybe the condition will take a long time before it evaluates to `true`. Is there any way to tell the thread to wait *indefinitely* until the time comes to wake up again?

As it happens, there is. An object can call its own method `wait()` to interrupt the execution of a thread at a given point. The thread will remain stopped until notified by another thread (using method `notifyAll()`[4]).

A thread calling `wait()` or `notifyAll()` must have the lock on the object, otherwise Java will throw an `IllegalMonitorStateException`. The easiest way to acquire the monitor is by synchronising the method, as in the following code:

```
01    public synchronized void doSomething() {
02        while (!conditionMet) {
03            try {
04                wait(); // sleep until notified
05            } catch (InterruptedException ex) {
06                // Nothing to do in this case, just wait less...
07            }
08        }
09        // Guarded block comes here...
10    }
11    // ...
12    public synchronized void anotherMethod() {
13        // ...
14        notifyAll();
15        // ...
16    }
```

When a thread executes `wait()` (line 4), it releases the lock it had acquired. Other thread may acquire it then and enter the same `synchronized` block of code (or others), and maybe get blocked at the same `wait()` call. The lock can also be acquired by another thread to enter a block of code where it calls `notifyAll()` (line 14)). At this point, all the threads that were waiting on the same object are woken up —but they cannot continue executing because the lock is still with the thread that executed the notification. After this thread releases the lock (line 16), one of the awaiting

[4]There is also a method notify(). There are clear differences between notify() and notifyAll(), but their discussion goes well beyond the scope of this section.

threads will re-acquire it (without the need to re-enter the code, it is already there) and continue its execution. Note that this may or may not have a real effect, depending on whether the condition has already been met (line 2); if it is not, the thread will `wait()` again (line 4).

1.6 Immutable objects

An immutable object is an object whose state cannot be changed after creation.

Use of immutable objects is a good strategy for creating simpler and more reliable code. This is especially true in the case of multi-threaded programs, where the complexity of interactions of different parts of the program increases because of the complexity of interaction between different threads. Since immutable objects cannot change state, they cannot be corrupted by thread interference and cannot be observed in an inconsistent state.

The simplest strategy to create immutable objects is declaring all the class' fields as `final`. However, this is not enough in the general case. In order to ensure that an object is immutable, the following rules must be followed:

- Make all fields not only `final`, but `private`.
- Do not provide "setter" methods, methods that modify fields (impossible since they are final) or objects referred to by fields (possible, even if the pointer is `final`).
- Do not allow subclasses to override methods. The easiest way to achieve this is by declaring the class as `final`.
- If instance fields point to mutable objects (e.g. a String), do not share the reference to those objects.
  - Never store references to objects passed in the constructor, make copies and point to those copies.
  - Never provide references to the objects pointed to by your instance fields. Make copies and return references to the copies.

The latter point is commonly referred to as *making defensive copies* and is another good strategy in multi-threaded environments, even if immutable objects are not used. Since it can be generally assumed that other threads will make changes to the objects passed to them, it is always a good idea to pass copies of our own objects instead of references to the objects themselves. Otherwise, other threads may make changes to the objects with unpredictable effects on other threads.

Making defensive copies at construction time or in response to method calls incurs in a certain overhead, but this rarely constitutes a bottleneck. The performance cost is generally compensated by the simplicity gained by the program.