Software Engineering Concepts (COMP3003)

2. Object Oriented Multithreading

Concurrency is the practice of having parts of your software run concurrently (at the same time). While there are several ways to achieve this, including multiple processes and distributed computing, we'll focus for the time being on **multithreading**.

Concurrency has two important advantages:

1. It can give us access to more computing resources. Hardware engineers are running out of ways to make individual CPU cores faster than they already are. They increasingly focus on providing more *parallel* computing resources (e.g. multiple CPU cores). The catch is: to actually *use* these resources, software must be multithreaded.

   Can the compiler magically make your code multithreaded? This is an active and highly ambitious area of computer science research. However, the best gains are still made by manually *designing* multithreading into your software from the start. This makes it a software engineering problem!

2. We can more easily implement separate performance requirements for particular parts of the software; e.g. the UI, network communication, data processing.

   Single-threading does not stop you implementing your functional requirements, no matter how complex they are. If a single-threaded system needs to schedule many different tasks, generated by different parts of the system, it can simply queue them up and execute them one-by-one. However, any particularly slow task will delay other tasks. In effect, there is *performance coupling* throughout the system, where the performance of any one part can affect every other part.

   For instance, the GUI might become unresponsive if some unrelated computational task happens to take a long time. You might be able to break that computation down into a set of micro-tasks, each taking a more reasonable amount of time, and allowing GUI tasks to be done in between. However, this is really a short-term hack. It complicates the algorithm (for a purpose that has nothing to do with the algorithm itself), and the same problem could arise again in the future.

   In a multithreaded system, task scheduling is all handled by the operating system, and tasks can happen simultaneously. If you start a long-running algorithm in a separate thread, then, provided there are no locking issues, other threads (including the GUI) do not need to wait for it. You can write the algorithm without worrying about how to break it down into tiny pieces.

1 Revision

You may have already encountered threads at some level, but we'll introduce some of the basic concepts again here just in case.

When your program starts a new thread, it creates another, simultaneous point of execution. Having multiple threads means that your code is executing at multiple points concurrently.

If you use separate threads to execute completely separate tasks, then your life is simple. You start all the threads and let them run. But often those tasks are *not* entirely separate.

Whenever threads need to communicate or share information, you must be very careful to avoid **race conditions**. A race condition happens when two threads are allowed to modify the same **shared resource** at the same time, or if one reads it while another modifies it. The effects of this are essentially random and unpredictable, because the threads are busy violating each other's assumptions.

A shared resource can be anything accessible by multiple threads, including any memory or files on disk. In practice, we'll concern ourselves with the most likely candidate in Java: class fields.

Say our resource is a single integer field `i`. Even a simple operation like `i++` can be stuffed up by a race condition. Say two threads attempt to perform `i++` at the same time, on the very same copy of `i` (which would therefore have to be a class field, if we're using Java). The overall effect *might* be `i += 2`, but not necessarily. If each thread reads the original value *before* the other thread has updated it, then they'll both attempt to set `i` to its original value plus one, which only adds one overall. So sometimes you add two, and sometimes only one, and you can't predict which.

As a slightly more complex example, consider adding and removing nodes from a shared linked list. These operations involves several steps, which, if done out of order, could destroy the list. And doing things out of order is precisely what happens in a race condition.

(To take a silly analogy: it's like having two cooks trying to bake the same cake at the same time with the same ingredients, but somehow unaware of each other's existence. You *won't* get twice the cake in half the time. You'll get scrambled eggs, flour all over the floor, and probably something on fire.)

To avoid race conditions, access to shared resources should be controlled by a locking mechanism: a **mutex** (mutual exclusion lock), or its slightly more complicated cousin, a **semaphor**. Before accessing a resource, a thread must lock it, and then unlock it afterwards. If the resource is already locked (by another thread), then thread waits until it is unlocked, and then locks it.

But although locking helps avoid race conditions, it comes with its own risks. A **deadlock** happens if two or more threads are each locking a resource, and (at the same time) each require another resource currently locked by another thread. Nothing can be done, because each thread is waiting for another one.

(A related condition called a **livelock** can also occur. You might think of a livelock as a complicated and very subtle infinite loop, where one task is re-tried endlessly. If the task requires a resource that isn't available, then instead of simply waiting for it (which could cause a deadlock), you might instead attempt to "roll-back" any changes made so far and re-schedule the task. However, that doesn't always solve the problem, and instead of a deadlock you may end up with a livelock. The code *is* performing work – it's not simply waiting – but no progress is being made.)

2 Threads and Languages

Multithreading is implemented by the operating system itself, but different languages have different levels of support for it, as summarised in Table 1.

The threading arrangement in Python and Ruby deserves its own special note. The "global interpreter lock" (GIL) in *some* implementations of Python and Ruby (CPython, PyPy, Ruby MRI) is a complex arrangement that prevents the interpreter running more than one thread at

Table 1: Basic support for multithreading in different languages.

Language	Multithreading support
C and C++ (prior to 2011)	No in-built threading support, other than the volatile keyword. Use an external library, like pthreads.[1]
C and C++ (2011 onwards)	The C11[4] and C++11[5] specifications add threading support to their respective languages, although actual C/C++ compilers do not necessarily implement the latest standards.
Java and JVM-based languages	Built-in threading support.[10]
C# and .NET-based languages	Built-in threading support.[9]
Python and Ruby	Built-in threading support, but some implementations impose a "global interpreter lock" (GIL), whereas others do not.[12]
JavaScript	True multithreading is not supported, although "web workers" provide an alternate form of concurrency.[3]

a time.[2] The GIL does not prevent threading, but does essentially prevent any performance gain from multiple CPU cores. (In fact, in the GIL's original implementation in Python, performance under multiple cores was actually worse than with just one core.)

Given multiple threads, the GIL it will still try to allocate time to all of them, though. In particular, even in an interpreted language, threads don't necessarily need the interpreter all the time, if they are performing IO operations like file reading/writing.

In other implementations of Python and Ruby, there is no GIL. Jython and JRuby compile their respective languages into JVM code, while IronPython and IronRuby compile to .NET CLR code. In these cases, there is no interpreter to lock.

3 Starting Threads and Tasks

Java's object oriented threading is one of its strengths (although the language designers did make some initial mistakes, as we'll see).

To explain how it works, we'll first distinguish between a **thread** and a **task**. A thread is a mechanism for executing code in parallel. A task is the code that you want to be executed (along with any necessary data). This distinction will become more important later on.

To create a task (one that can be executed in a thread), you first implement the `Runnable` interface, and override the `run()` method:[11]

Listing 1: Defining a task as a standalone class.

```java
public class MyTask implements Runnable
{
    @Override
    public void run()
    {
        ... // Do some work in the new thread.
    }
}
```

Then, to run the task in a new thread:

Listing 2: Starting a new thread, given a task.

```java
MyTask myTask = new MyTask();
Thread myThread = new Thread(myTask, "my-thread");
myThread.start();
```

The string parameter gives the thread a name. Thread names don't really mean anything, and need not be unique, and you can omit the name parameter. However, when designing, profiling and debugging a multithreaded application, having names for your threads can be useful.

Further, some developers choose to extend `Thread` itself and override *its* `run()` method, instead of implementing `Runnable`. However, we won't do that here, as it's helpful to keep the distinction between tasks and threads. (FYI, implementing `Runnable` is a case of the Strategy pattern, whereas extending `Thread` is a case of the Template Method pattern.)

We'll get back to *stopping* threads in Section 8.

There are also a couple of alternatives here. We might prefer to encapsulate the task and thread within a single class, so that the details can remain hidden from the rest of the code:

Listing 3: A class that defines its own task internally, and runs it in a new thread.

```java
public class ThingDoer
{
    private Thread myThread; // We generally want to keep a field reference
                             // to the thread (for other purposes later on).
    public void start()
    {
        Runnable myTask = new Runnable() // Define the task as an anonymous/
        {                                // inner class.
            @Override
            public void run()
            {
                ... // Do some work in the new thread.
            }
        };

        myThread = new Thread(myTask, "my-thread"); // Start thread.
        myThread.start();
    }
}
```

Also, since Java 8, we can alternatively use lambdas or method references to make things slightly cleaner:

Listing 4: Defining tasks with Java's lambda syntax.

```java
// Instead of 'Runnable myTask = new Runnable() ...'
Runnable myTask = () ->
{
    ... // Do some work in the new thread.
};
myThread = new Thread(myTask, "my-thread");
myThread.start();
```

Listing 5:
Defining a task using a method reference.

```java
myThread = new Thread(this::myTask, "my-thread");
myThread.start();
...
private void myTask()
{
    ... // Do some work in the new thread.
}
```

(Note that Java's method references are a syntactic short-cut, not a new datatype. Writing "`this::myTask`" causes the compiler to create a new `Runnable` object whose `run()` method calls `myTask()`, so it's not *really* a direct reference to `myTask()` itself.)

4 Shared Resources

It's **shared resources** that cause all the real threading problems: race conditions, deadlocks, and (as we'll see) memory synchronisation issues.

Note 1:
Shared ≠ public.

Sharing a resource between threads is a completely different idea to sharing it between classes. Resource sharing happens at runtime, and can (and in fact usually does) happen *within* a single class on its *private* fields. The resource is shared between threads, but there's usually only one physical section of code involved.

If you have an **immutable** (un-modifiable) field or object, you can breathe a sigh of relief. For a race condition to happen, at least one of the threads involved must be attempting to modify a resource; hence, an immutable resource is immune to race conditions. Anything immune to race conditions doesn't need to be locked, in which case it's not going to cause deadlocks or livelocks either. Hurrah!

Therefore, immutability is a highly desirable characteristic in multithreaded programming. You should generally try to design classes to be immutable where practical.

Java's `String`, `Integer`, `Double`, `Character` and similar classes are all immutable. Container classes (lists, sets and maps) are not, but you can use immutable *wrappers* around them; e.g. using `Collections.unmodifiableList()`, `Collections.unmodifiableMap()`, etc.

However, it can be very difficult to *only* use immutable objects if your threads need to communicate, or record their results in a shared location. So we also need **thread-safe** *mutable* resources.

A thread-safe mutable resource must have **atomic** operations. "Atomic" literally means "indivisible"[a], although we really mean operations that *appear* to happen in just one step. That means that two such operations cannot happen at the same time, and in this way we avoid race conditions.

In practice, this means the resource locks itself when accessed. Java's `AtomicInteger` class does this, for instance[b]. `Collections.synchronizedList()` will create a self-locking wrapper object around an existing list (and there are equivalents for sets and maps). If you attempt to add something, you're guaranteed that no other thread will be able to read or modify the list at the same time.

[a] Physical atoms were originally thought to be indivisible, hence their name.

[b] There are a few other "`Atomic...`" classes in the `java.util.concurrent.atomic` package.

It may interest and puzzle you to know that `LinkedList`, `HashMap` and their like *are not* thread-safe by default. Why is this? After all, they *could* be made thread-safe without breaking backwards compatibility (since you wouldn't need to modify their public interfaces).

Many of the standard Java classes (those that are mutable) are *deliberately* not thread-safe for one reason: efficiency. A mutex is not a complicated object, but locking and unlocking does incur a cost in processing time, especially if done frequently. So, if a single thread is using a linked list purely for its own internal purposes, and not sharing it, there's no point in locking and unlocking it. Moreover, several objects may form part of a single combined resource. In that case, they need to be locked collectively, and then there's no point locking them individually as well.

5 Creating Thread-Safe Classes

We must also be able to make our own code thread-safe, of course, and this is where we start locking things directly.

5.1 The synchronized Statement

Java has a language construct for locking resources: the `synchronized` statement:

Listing 6

```
synchronized(mutex)
{
    ... /* Do something with the locked resource */
}
```

Only one thread at a time can execute the contents of this `synchronized` statement. If there are any other `synchronized` statements that share the same `mutex`, only one thread can execute *any* of them at a time.

The `mutex` is an object that works as a mutual exclusion lock. It keeps track of whether the resource is currently in use[c]. Curiously, Java's `Object` class implements mutex functionality[d], so *any* object at all can be used as a mutex. However, for the sake of clarity (amongst other reasons), it's best to make a dedicated mutex object[e]:

Listing 7

```
Object mutex = new Object();
```

Now for a small but complete example:

[c] A mutex does not know what the resource *actually is* – it's your job as a developer to keep track of that, and this tends to be done implicitly.

[d] It actually implements something slightly more complex, called a **monitor**. We'll get back to that.

[e] Fun fact: this may be the only useful thing you can do with a pure-`Object` object.

Listing 8: Using the `synchronized` statement to create a thread-safe class.

```java
public class Counter
{
    private Object mutex = new Object();
    private int i = 0;

    public int get()
    {
        synchronized(mutex)
        {
            return i;
        }
    }

    public void increment()
    {
        synchronized(mutex)
        {
            i++;
        }
    }
}
```

Here, `i` is the resource that we want to share between threads, and `synchronized(mutex)` is the locking mechanism that lets us do that safely.

If multiple threads call the `increment()` method simultaneously, only one at a time will be able to execute the `synchronized` statement. The same goes for `get()`, or if one calls `get()` while another calls `increment()`. This effectively makes these operations atomic – one operation must end before any other can begin.

Java also lets you use the `synchronized` keyword as a method modifier:

Listing 9: The `synchronized` *modifier*.

```java
public synchronized void increment()
{
    i++;
}
```

This is functionally equivalent[f] to:

Listing 10: Using `this` as a mutex.

```java
public void increment()
{
    synchronized(this)
    {
        i++;
    }
}
```

Like any object, “`this`” can serve as a mutex.

[f] The compiled code is actually different, but ultimately has the same effect.

This usage is (slightly) discouraged because it technically violates encapsulation. Synchronisation is an implementation detail, but "`this`" is effectively public. Anything else that has a reference to this object can (in theory) use it as a mutex. If the object is also using itself as a mutex internally, it might end up waiting when it doesn't strictly need to, and possibly deadlocking as a result.

For the same reason, it's better to stick to pure-`Object` mutexes. For a more complex object, you can't be sure (from its public interface alone) whether it has a `synchronized(this)` buried inside it somewhere. If it does, and if you use it as a mutex, the same problem can occur.

Here are some other potential issues in using `synchronized` as a method modifier:

- It will lock the mutex for the entire duration of the method, but in some cases you might only need the shared resource for part of the method. Locking for longer than necessary is inefficient, because you will end up blocking other threads for longer than necessary.
- It isn't applicable if you need to lock multiple, independent resources. A single mutex can only lock one resource. If there are several that must be locked separately from each other, you must use nested `synchronized` statements.

5.2 Encapsulation

As alluded to in Note 1, there can be some confusion over how to design OO structures for thread communication. (The `synchronized` statement itself only lets us perform very basic thread communication – more on that in Section 7 – but it's enough to illustrate the point.)

A good way to think about thread communication is as the responsibility of *one* class. Often a class will do the following:

- Start one thread (let's call it "thread-1"), and provide much (or all) of the code that this thread executes.
- Have a set of public accessors and mutators intended to be called by a different thread ("thread-2").
- Contain the mutex and shared resources, as *private* fields, that will allow these two threads to communicate.

Multiple parts of the same class will sometimes run simulaneously in both thread-1 and thread-2. This way, you can handle all the threading issues in one place (depending on what else the newly-started thread gets up to if it is allowed to call methods in other classes).

Note 2: Method calls and threads.

Remember that method calls only happen *within* a single thread. When `Class1.a()` calls `Class2.b()`, both methods run in the same thread. Communication *between* threads is what mutexes are for.

Figure 1 shows broadly how encapsulation should work with multiple threads.

Avoid having multiple classes be aware of the locking mechanisms needed to access a shared resource. You might be tempted to make `ClassA` "responsible" for thread-1, and `ClassB` "responsible" for thread-2, and to have objects of both classes access the same mutex object. This is not a good idea, as it creates complex and unnecessary coupling between the classes. Thread communication *is not* class communication. The resource belongs to one object only, and so should the mutex.

Consider Listing 11 as a more detailed example. The `RunningCounter` class is a `Runnable`, whose task (as you can see from the `run()` method) is to print out the value of a counter once per second. From time to time, another thread will call the `increment()` method to change the counter's value. We have a mutex in each method to avoid a race condition.

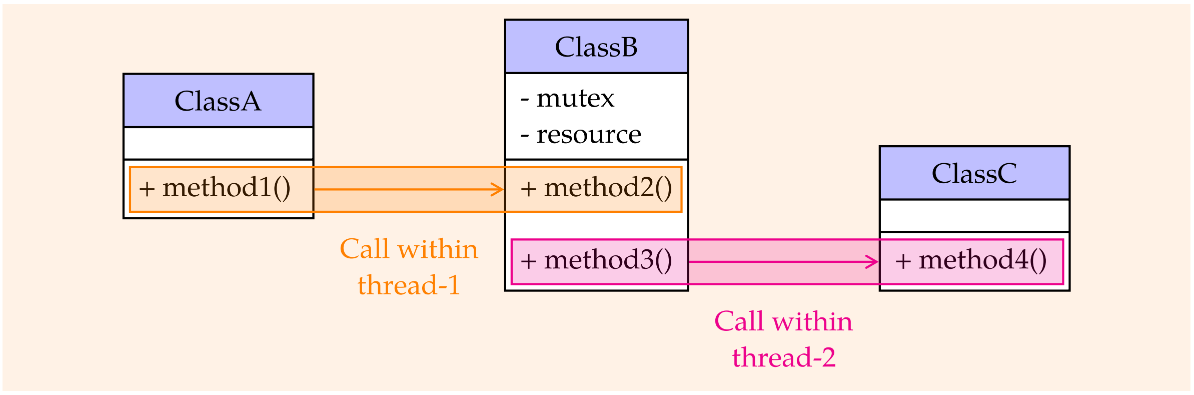

Figure 1: Threading and separation of concerns. ClassB handles thread interaction, so ClassA and ClassC don't have to.

This class encapsulates the interaction between two threads. From the outside, we can simply call `increment()` without worrying at all about how the thread interaction works.

We could also shift the actual thread creation code into `RunningCounter`. Rather than having the class implement `Runnable`, we could give it a `go()` method that contains and instantiates an anonymous nested `Runnable` class, and starts it in a new thread. The outside world would then only know about `go()` and `increment()`, and all the threading issues would be handled "behind the scenes" inside `RunningCounter`.

In any case, notice that Listing 11 is really just a more complex version of Listing 8.

6 Memory Synchronisation

Memory synchronisation is an "under-the-hood" technical issue that you should be aware of, alongside race conditions and deadlocks. This section won't give you much additional practical advice, but it will hopefully contribute towards a deep understanding of multithreading. (Having a deep understanding will ultimately make a difference to how you approach the design process.)

6.1 What `synchronized` Really Does, and Why

The `synchronized` statement (and modifier) does more than just locking. It also instructs the compiler to *synchronise the memory* in which fields are stored (those fields used within the statement, anyway).

What does that mean? First, you must understand something wonderful and horrible: compilers don't really do what you tell them! Rather, they do something functionally equivalent to what you tell them, given certain assumptions. They have a lot of scope to make changes, in order to produce faster code.

Some of their tricks include re-arranging the order of instructions, removing operations entirely if they have no effect, and caching field values in a CPU register[g]. Registers are much faster than main memory, where the field is "officially" stored[h]. If a class field is frequently

[g] The JVM doesn't have any registers itself. However, prior to being executed, JVM code is usually *re-compiled* into native code by a "just-in-time" (JIT) compiler. The JIT compiler is completely separate from the `javac` command, and does its work transparently while your program is actually running. It's the JIT compiler that tends to perform the more interesting optimisations.

[h] This because the speed of light, which limits the speed of electrical signals, is a mere 1.08 billion kilometres per hour. This is *too slow* to get data to/from RAM (a few centimetres from the CPU) in a reasonable amount of time. Needless to say, we have become quite fussy about what a "reasonable amount of time" is these days. However, CPU registers are physically much closer to where the actual instructions are processed, and so can be accessed much faster.

Listing 11: Encapsulating thread interaction within a single class.

```java
public class RunningCounter implements Runnable
{
    private Object mutex = new Object();
    private int resource = 0;

    @Override
    public void run()
    {
        // Runs in thread-1.
        try
        {
            while(...) // Some exit condition
            {
                synchronized(mutex)
                {
                    System.out.println(resource);
                }
                Thread.sleep(1000); // Do nothing for 1 second.
            }
        }
        catch(InterruptedException e) { ... /* See Section 8 */ }
    }

    public void increment()
    {
        // Runs in thread-2.
        synchronized(mutex)
        {
            resource++;
        }
    }
}
```

read and/or updated, the compiler may choose to work with the value in the register, and only occasionally re-read or update the value in memory.

This is valuable work, but there is a catch: by default, the compiler assumes that there is only one thread, or at least that no other thread needs to know what this one is doing. This assumption allows it to make more aggressive optimisations. To allow threads to communicate – that is, for one thread to see field values set by another thread – you must give the compiler a set of **synchronisation points** in the code. At each synchronisation point, the compiler must ensure that the thread's work becomes visible to other threads. There is a synchronisation point at the start of each `synchronized` statement, and another at the end.

To see why this matters, take the `Counter` example in Listing 8, and say one thread executes the following code:

Listing 12

```
public void printCounter(Counter counter, int times)
{
    for(int i = 0; i < times; i++)
    {
        int counterValue = counter.get();
        System.out.println(counterValue);
    }
}
```

Here, we simply print out the counter a lot of times, and expect that the value may change at some point if/when another thread calls `increment()`. However, if we took away the `synchronized` statement from inside `get()`, the compiler would stop thinking that another thread may be involved. As a result, it could decide that `counterValue` *cannot* change, and so, to save time, it may choose to call `get()` *only once*! Or, even if it does call `get()` multiple times, it may choose to return a cached value, rather than the up-to-date one. This would obviously ruin our plans.

Something similar happens with `increment()` itself, in fact. Without the `synchronized` statement there, even if we ignore the race condition, the compiler might choose to cache the new value of `i` for a while before actually writing it to memory. It doesn't take into account that another thread might need to read it.

A `synchronized` statement (apart from locking a resource) ensures that one thread sees changes made by another. If the code in a `synchronized` statement reads a particular field, the *compiled* code will actually read it from memory *within* the statement (and not rely on any value read and cached before the statement). Similarly, if the code inside updates a field, the compiler ensures that the value is written to memory before the statement ends.

(The compiler *may* still cache values for the duration of the statement itself, after it starts and before it ends. That's okay because the contents of the statement are only executed by one thread at a time, of course.)

6.2 Volatile

You might occasionally hear about the `volatile` keyword. It appears in C, C++, Java, C# and elsewhere (although not in Python). In all these languages, `volatile` performs memory synchronisation, but *without* locking anything.

Since we almost always want to do both these things at the same time, the usefulness of `volatile` can be limited. Nonetheless, you should be aware of it.

In Java, only fields can be declared `volatile` (as it makes no sense for a local variable, whose value can only be accessed by one thread):

Listing 13

```
private volatile int theField;
```

Each read or update of `theField` is considered a synchronisation point (just like the start and end of a `synchronized` statement, but without the locking). The value of `theField` is never cached, so it is always "up-to-date". Moreover, in Java and C# (but not in C or C++) it controls the synchronisation of other non-volatile fields around it.

If a thread reads `theField` and then afterwards reads other non-volatile fields, they are guaranteed to be at least as up-to-date as the value of `theField`. Similarly, if a thread writes to theField, any preceeding writes to non-volatile variables must cease being cached and actually take effect in memory.

Note that we're only talking about the *direct* value of theField. If the field is a reference to an object, volatile only applies to the reference itself, not to the object.

(Incidentally, in C and perhaps C++, volatile is also used in very low-level code, particularly device drivers, to talk directly to hardware devices. Usually this happens by "mapping" specific memory locations onto the hardware device's own internal registers, transforming those memory addresses into a communication channel between hardware and software. These must be flagged as volatile to prevent the C compiler assuming, as most compilers would, that only the code will modify the memory. Java, C# and other higher-level languages are not used for this, in practice.)

Listing 14: A contrived example of the volatile keyword.

```
public class Counter
{
    private volatile int i = 0;

    public int get()
    {
        return i;
    }

    public void increment()
    {
        i++; // Race condition, because we're not locking anything.
    }
}
```

Listing 14 is a slight modification of Listing 8. We've removed the synchronized statements and added volatile to i. Each call to get() will still retrieve an up-to-date (synchronised) value. *However*, there's nothing to prevent a race condition if two threads call increment() simultaneously.

7 Thread Communication: Monitors

Threads communicate using shared resources. So far, we can get multiple threads to safely access the same resource, which they could use to pass information to each other. However, actual communication relies on careful timing. The threads must be able to signal each other, so that one can tell another that information is actually available.

To explain: say thread-1 produces some information that thread-2 needs, and this information will be stored in a shared object. We *could* have a shared boolean flag to indicate when the information is available. Thread A would set this to true at the right time, and thread B would continually check it. However, *polling* (continual checking) is extremely inefficient. As a means of waiting for another thread, the empty loop construct "while(!updated) {}" is a terrible waste of CPU time[i]. It would be better if thread-2 could simply go to sleep and wake up when the information arrives.

[i] It also potentially has other issues. The "updated" flag must be volatile or inside a synchronized statement. If you have multiple threads waiting, with one or more waiting to *write* to the resource, and you lock the resource separately from the updated flag, you will create a race condition.

Enter **monitors**. A monitor is a mutex plus a couple of additional operations[j]. In total, a monitor has *four* operations: lock, wait, notify and unlock. We've been using a pure `Object` object as a mutex so far, but it actually has complete monitor functionality.

7.1 Scenario A: Thread-2 Waits and Thread-1 Notifies

In the simplest case, the arrangement works as follows, and as shown in Listing 15:

1. Thread-2 **locks** the resource (that it will eventually need) using the monitor, and then **waits** on the monitor. Thread-2 goes into suspended animation. (This is a "deeper sleep" than when a thread waits for a lock to be released.)

   The wait operation also temporarily *unlocks* the monitor, so the original locking may seem redundant. However, in general, thread-2 might want to do something with the resource before waiting.

2. With the monitor temporarily unlocked by thread-2, thread-1 now **locks** it for itself.

   (If there are more than two threads involved, there's nothing guaranteeing which particular thread will get to the resource immediately after thread-2 has released it, but eventually it will be the turn of thread-1.)

3. Thread-1 writes its information to the resource, and then **notifies** the monitor.

4. The monitor picks one of the threads waiting on it (thread-2 in this case, but in other situations there could be several), and wakes it up.

   However, thread-2 can't do anything right away, because the resource is still locked by thread-1. This blocks thread-2 for a very short period of time.

5. Thread-1 now **unlocks** the resource, but the monitor knows that it must give the lock back to thread-2 specifically. Other threads cannot intervene.

6. Thread-2 begins executing again, and can retrieve thread-1's information from the resource.

7. Finally, of course, thread-2 **unlocks** the resource.

[j] A monitor is a mutex plus a **condition object**, but it's usually simpler to think of it as just *one* thing.

Listing 15: Using a monitor to notify another thread (simplistic).

```
public class FireworksFactoryA implements Runnable
{
    private Object monitor = new Object();
    private Firework firework;

    @Override
    public void run()
    {                    // We'll discuss the try-catch, while(true) and
        try              // InterruptedException later on in more detail.
        {                // For the moment, just note that we're making
            while(true) // new objects inside a loop.
            {
                synchronized(monitor)                    // Step 2
                {
                    // Create resource
                    firework = new Firework();           // Step 3
                    f.addGunpowder();
                    f.addRocket();
                    f.addPackaging();

                    // Tell the other thread we've done it.
                    monitor.notify();                    // Steps 3 and 4
                }                                        // Step 5
            }
        }
        catch(InterruptedException e) {...} // Finished
    }

    public Firework getNextFirework() throws InterruptedException
    {
        synchronized(monitor)
        {
            // Wait for it to be created, then return it.
            monitor.wait();                              // Step 1
            return firework;                             // Step 6
        }                                                // Step 7
    }
}
```

7.2 Scenario B: Either Thread Waits For The Other

Scenario A only tells half the story, and there's an important problem with it: the loop in run() will keep producing new fireworks even if getNextFirework() is not being called (or not called often enough). The new objects will just be garbage collected again.

In the more general case, we *probably* don't want to update a resource until we know that previous updates have been dealt with. To make this happen, we need to make *both* sides wait and notify.

Listing 16 shows this in action.

Listing 16: Using a monitor to coordinate two threads (better but not perfect).

```
public class FireworksFactoryB implements Runnable
{
    private Object monitor = new Object();
    private Firework firework = null;

    @Override
    public void run()
    {
        try
        {                          // More efficient to lock outside the loop,
            synchronized(monitor) // and stay locked except during wait().
            {                      // Unlocking-and-immediately-relocking is
                while(true)        // redundant and inefficient.
                {
                    if(firework != null)
                    {
                        monitor.wait(); // First wait until the existing
                    }                   // resource is consumed.
                    firework = new Firework(); // Now make a new one.
                    f.addGunpowder();
                    f.addRocket();
                    f.addPackaging();
                    monitor.notify(); // Tell the other thread to go ahead.
                }
            }
        }
        catch(InterruptedException e) {...} // Finished
    }

    public Firework getNextFirework() throws InterruptedException
    {
        synchronized(monitor)
        {
            if(firework == null)
            {
                monitor.wait(); // First wait until a new resource
                                // has been created.
            }
            Firework ret = firework; // Consume the resource, and tell the
            firework = null;         // other thread it's been done.
            monitor.notify();
            return ret;
        }
    }
}
```

Notice we've also placed the loop *inside* the `synchronized` statement, for efficiency. This is still perfectly workable, since `wait()` temporarily unlocks the monitor.

7.3 More than Two Threads (Also, Spurious Wakeups)

What if we have *three* or more threads? For instance, we could have multiple threads simultaneously calling `getNextFirework()`. In Java, if a monitor is notified and it has *multiple* waiting threads, *one waiting thread is chosen arbitrarily.* You can't predict which thread will actually wake up.

This is a problem, because if there are two simultaneous calls to `getNextFirework()` going on, then one could end up notifying the other. That is, one call will consume the resource, call `notify()`, and then the other waiting call will wake up *before* `run()` has had a chance to create the next firework.

The solution to this has two parts. First, we replace `if` with `while`; e.g.:

Listing 17: Proper convention for waiting.

```
~~if(firework == null)~~
while(firework == null)
{
    monitor.wait();
}
```

We do this for the `firework != null` case too. In general, we do it for *every* `wait()` call. If the wrong thread wakes up, it must re-check its condition, and it goes back to sleep if it's not yet met.

This actually solves *another* problem too, which could happen with only two threads: **spurious wakeups**. Technically there's no guarantee that `wait()` won't just wake up for no apparent reason at all, due to the underlying OS thread implementation. This is the subject of some debate, but is a real (if rare) phenomenon.[6,7]

Back to the three-thread case: even if we can deal with the wrong thread waking up, we must *still* ensure that the right one does too. Otherwise, *every* thread will be waiting and we'll have a kind of trivial deadlock.

So, we also replace `notify()` with `notifyAll()`:

Listing 18: Notifying all waiting threads.

```
~~monitor.notify();~~
monitor.notifyAll();
```

This wakes up *all* the waiting threads at once (for that monitor). They will each then re-examine their conditions, and the right one will proceed to do its job.

8 Interrupting Threads

What if the user wants to press a big red "Cancel" button half-way through the task? How do you *stop* a thread?

Any thread can stop *itself* using normal control statements as needed: `break`, `return`, `throw`, etc. When control reaches the end of the `run()` method, the task ends, and therefore so does the thread[k].

But how can one thread stop a *different* thread?

[k] Later, we'll see that we can actually reuse the same thread for multiple tasks, so this isn't technically true, but it's close enough for now.

8.1 Don't Call `stop()`

`Thread` objects have a `stop()` method. *Don't* call it! It does indeed stop a thread, but its design is fatally flawed, and it has long been deprecated[l].

Just so you know, `stop()` works by creating a `ThreadDeath` exception[m] inside the thread. Like any exception, if not caught, it will cause all currently executing methods to end, including the task's main `run()` method.

This is a very blunt weapon, and can leave behind a trail of destruction from the unfinished remnants of whatever the thread was doing. You can end up with objects in invalid states, when a constructor or mutator abruptly stops after setting *only half* of the object's field values. This can lead to these objects doing things that, according to the code, ought to be impossible.

You might think a `try-catch-finally` could help solve this. You could catch `ThreadDeath`, or use `finally` to clean up when it happens. But the problem is that `ThreadDeath` could happen *literally anywhere*, unpredictably. It's unreasonable to have to put a `try-catch-finally` around every line of code, and there's nothing to stop `ThreadDeath` happening inside the clean-up code itself.

8.2 Do Call `interrupt()`

`Thread` objects *also* have an `interrupt()` method, which is generally the correct way to stop a thread from another thread. Here's how you might use it:

Listing 19: Starting and interrupting a thread.

```java
public class Controller
{
    private Thread t = null;

    public void begin()
    {
        // Verify that 't' doesn't exist yet.
        if(t != null) { throw new IllegalStateException(...); }

        // Start a new thread
        t = new Thread(new FireworksFactory(), "fireworks-factory");
        t.start();
    }

    public void end()
    {
        // Verify that 't' DOES exist.
        if(t == null) { throw new IllegalStateException(...); }

        // Interrupt existing thread
        t.interrupt();
        t = null;
    }
}
```

[l] Although Java's threading is good overall, the `stop()` method was one of the clearest mistakes of Java 1.0.

[m] `ThreadDeath` is technically a subclass of `Error`, not `Exception`. The mechanics are the same, except that the compiler does not force you to catch or declare `Error`s.

(Note: the `if(...){throw ...}` lines are just sanity checks, and `IllegalStateException` "signals that a method has been invoked at an illegal or inappropriate time."[n].)

`Thread.interrupt()` just sets the **interrupted status** to `true`. Then, the code inside the target thread must do one of two things:

1. Call a blocking method, like `wait()`. Apart from its main purpose (waiting for another thread to do something), `wait()` also checks the interrupted status. If the thread has *already been* interrupted when `wait()` is called, or is interrupted during the call, it throws `InterruptedException`.

   Other blocking methods do likewise; e.g. `Thread.sleep()`, `BlockingQueue.put()`, etc. We'll talk about blocking queues later.

2. Check the interrupted status directly, with `Thread.interrupted()` (which returns `true` or `false`). When interrupted, you can take whatever steps you like to end the task, though throwing `InterruptedException` yourself is often appropriate.

8.3 Blocking Methods

All methods take *some* time to run, but methods (like `wait()`) are said to **block** when they take a relatively long time, during which (of course) the calling code cannot do anything. Remember that even a few milliseconds is a "long time" compared to most simple computational tasks.

Blocking methods are a good place to interrupt a thread. They only occur at specific points in the code, so you don't have to worry about the interruption occurring *anywhere*. And if blocking methods weren't interruptible, stopping a thread may not even be possible in a reasonable amount of time.

Let's revisit the `run()` method in the fireworks example:

Listing 20: Catching `InterruptedException`.

```
@Override
public void run()
{
    try
    {
        while(true)
        {
            ...
            monitor.wait(); // May throw InterruptedException
            ...
        }
    }
    catch(InterruptedException e)
    {
        // Not essential, but perhaps a good idea:
        Logger.getLogger("Fireworks").info("Production interrupted!");
    }
}
```

The purpose of `while(true)` may now become apparent. `InterruptedException` is the way out of the loop. We're telling the task to run forever *until interrupted*. When the interrupted

[n] https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/IllegalStateException.html

status becomes `true`, `monitor.wait()` will discover it and throw an exception, which we catch. Then `run()` ends, and so does the thread.

The logging step is not strictly essential, and not a threading-specific topic. However, logging is good practice, and empty catch blocks are not.

Of course, `FireworksFactory` has this as well:

Listing 21: Re-throwing `Interrupted-Exception`.

```java
public Firework getNextFirework() throws InterruptedException
{
    ...
    monitor.wait(); // May throw InterruptedException
    ...
}
```

Here we *don't* catch `InterruptedException` but rather let it be automatically thrown from the method. Why did we do this?

The situation is different here. We're in a different thread, started in some other class, in some other `run()` method, which has called `getNextFirework()` (whether directly or indirectly). To stop this thread, we need to get back to that other `run()` method and let it end. Throwing `InterruptedException` is the conventional way to do that.

You can look at it another way too: because `getNextFirework()` calls a blocking method, *it itself* is also a blocking method. Calling `getNextFirework()` results in a waiting period. And the convention is that blocking methods throw `InterruptedException`.

8.4 Checking the Interrupted Status

If you're using threads to execute long-running, CPU-intensive algorithms, you may not call any blocking methods. If you need maximum performance due to the computational complexity of the task, you probably don't want to do any `wait()`ing, for instance.

So, if you still want your computation thread to be interruptible, you have to regularly check its interrupted status:

Listing 22: Checking the interrupted status.

```java
@Override
public void run()
{
    long result = 0L;
    for(int i = 0; i < 1000000000; i++) // Long-running task
    {
        result += i;
        if(Thread.interrupted()) // Check for interruption.
        {
            break; // Stop the loop, and therefore the thread.
        }
    }
    System.out.println("result = " + result);
}
```

However, if we check the interrupted status *this often*, the checks themselves may slow down the algorithm. Instead, we could let the algorithm perform some more work between checks:

Listing 23: Checking the interrupted status less frequently.

```java
@Override
public void run()
{
    final int INTERVAL = 10000;
    long result = 0L;
    for(int i = 0; i < 1000000000; i += INTERVAL) // Same long-running task.
    {
        for(int j = 0; j < INTERVAL; j++) // But do more work before checking.
        {
            result += i + j;
        }
        if(Thread.interrupted()) // Check on each 10000'th iteration.
        {
            break;
        }
    }
    System.out.println("result = " + result);
}
```

In the previous two examples, we could have thrown and caught `InterruptedException`, but, since we're already inside `run()`, we can end it with ordinary control statements instead. When *not* inside `run()`, it's reasonable to throw an exception:

Listing 24: Checking the interrupted status and throwing `Interrupted-Exception`.

```java
public long performCalculation() throws InterruptedException
{
    final int INTERVAL = 10000;
    long result = 0L;
    for(int i = 0; i < 1000000000; i += INTERVAL) // Same long-running task.
    {
        for(int j = 0; j < INTERVAL; j++).
        {
            result += i + j;
        }
        if(Thread.interrupted())
        {
            throw new InterruptedException();
        }
    }
    return result;
}
```

8.5 Uninterruptible Blocking

Previously, we said that blocking methods throw `InterruptedException`, but sadly this isn't *always* true. Some long-running operations are **uninterruptible**, and you may need to carefully work around them.

For instance, input/output (IO) operations can block uninterruptibly (depending on which ones you use). IO operations include console input/output, file reading/writing, some inter-process communication, and network sending/receiving. These block because such operations are much slower than the CPU, which needs to wait. But they can also block *indefinitely* for various reasons, the simplest example being waiting for console input:

Listing 25: Uninterruptible IO.

```
public int readInteger()
{
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter an integer: ");
    return sc.nextInt(); // Will block indefinitely and uninterruptibly.
}                        // InterruptedException WILL NOT be thrown!
```

`Scanner.nextInt()` blocks until user input is ready, but this isn't really `Scanner`'s fault. To do its job, it needs to call another uninterruptible blocking method, `System.in.read()`, to retrieve individual characters, before parsing them to obtain an integer.

(Also note that `read()` itself doesn't generally obtain those characters immediately as the user types them, either. The underlying OS/platform typically holds off providing them until the user presses Enter, to allow the user to first edit their input if needed.)

Fortunately, there are sometimes ways to tell *whether* an IO operation will block before we invoke it:[8]

Listing 26: Interruptible IO.

```
public int readInteger() throws InterruptedException
{
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    Scanner sc = new Scanner(br);
    System.out.println("Enter an integer: ");

    while(!br.ready())     // Do we have user input yet? (Doesn't block.)
    {
        Thread.sleep(100); // Sleep for 100 ms; may throw InterruptedException.
        ...                // Maybe do other things too while waiting.
    }
    return sc.nextInt();   // Won't block, because the input is ready now.
}
```

Here, we insert a `BufferedReader` in between `System.in` and `Scanner`. Then `BufferedReader`'s `ready()` method tells us whether we can read a complete line of text (presumably containing an integer) *without* blocking. So we poll the `ready()` method. That is, we call it multiple times, "sleeping" for a short interval between each call, until the user input is ready.

To sleep, we call `Thread.sleep()`. This blocks for the specified number of milliseconds, *but* also checks the interrupted status, and throws `InterruptedException` when needed. Not only that, but between `sleep`s we can do any arbitrary tasks at the same time as waiting for our user input (all on the same thread).

This won't work for all IO operations though. Sometimes `ready()` always returns `false`, because the operation always blocks (or at least is always considered to block, even if only for a short time). In these situations, you may need to bring out the "heavy guns" of the `java.nio` package (where "NIO" simply means "new IO"); for instance:

Listing 27: Interruptible IO using `java.nio`.

```java
public String readURL(URL url) throws InterruptedException, IOException
{
    try(ReadableByteChannel chan = Channels.newChannel(url.openStream()))
    {
        ByteBuffer buffer = ByteBuffer.allocate(65536);
        ByteArrayOutputStream bigBuffer = new ByteArrayOutputStream();

        int bytesRead = chan.read(buffer); // Blocks while reading, but can
        while(bytesRead != -1)             // throw ClosedByInterruptException.
        {
            bigBuffer.write(buffer.array(), 0, bytesRead)
            buffer.clear();
            bytesRead = chan.read(buffer); // As above.
        }
        return bigBuffer.toString("UTF-8");
    }
    catch(ClosedByInterruptException e)
    {
        throw new InterruptedException(e); // Re-throw the "right" exception.
    }
}
```

In Listing 27, we use `ReadableByteChannel` to download a URL (e.g. a web page). Its `read()` method will block, and could block for some time depending on the state of the network. Network IO is very slow and unreliable compared to anything you would do *internal* to the computer. Fortunately, it blocks interruptibly, although it throws a slightly different exception, `ClosedByInterruptException` (a subclass of `IOException`), to indicate interruption.

(In case you haven't seen it, "`try(<declaration>){...}`" is a try-with-resources statement, commonly associated with IO operations. It acts as a normal `try`, but once finished automatically calls `close()` on the object(s) declared in parentheses, irrespective of whether an exception occurs. It's significantly messier to do this with a `finally` block.)

Finally, there is one more obvious uninterruptible blocking operation we might (sometimes) be concerned about: the `synchronized` statement itself!

Locking a monitor at the beginning of a `synchronized` statement can block, and does not throw `InterruptedException`. Typically you only lock a monitor for a very short time, in which case it doesn't matter that it's uninterruptible. (Even in the fireworks example, where we had an infinite loop inside a `synchronized` statement, we still unlocked the monitor every time we called `wait()`, and we didn't do much between calls.)

If you do need to lock *interruptibly*, there is a way. Inside the `java.util.concurrent.locks` package, there is a `Lock` interface, along with an implementing class `ReentrantLock` (among others). These provide more flexible locking facilities than Java's pure-`Object` monitors. We have to give up the `synchronized` statement, and instead call `lock()` and `unlock()`. However, we can call `lockInterruptibly()`, which will throw `InterruptedException` if the thread is interrupted beforehand, or while we're waiting.

9 Further Reading

- Neil Coffey (2014), Javamex UK, http://www.javamex.com/.

This site contains a series of tutorials and explanations on Java's multithreading capabilities.

- Jeremy Manson and Brian Goetz (2004), JSR 133 (Java Memory Model) FAQ, https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html.

  Java's memory model was updated in Java 1.5. This explains what was wrong, what was done to fix it.

9 References

[1] Blaise Barney. POSIX Threads Programming. Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/pthreads/, 2017. Accessed: 2017-08-03. **[See section 1.]**

[2] David Beazley. Inside the Python GIL. http://www.dabeaz.com/python/GIL.pdf, 2009. Accessed: 2017-08-03. **[See section 2.]**

[3] Mozilla Contributors. MDN Web Docs: Web Workers API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API, 2017. Accessed: 2017-08-03. **[See section 1.]**

[4] Cppreference.com contributors. C reference: Thread support library. http://en.cppreference.com/w/c/thread, 2015. Accessed: 2017-08-03. **[See section 1.]**

[5] Cppreference.com contributors. C++ reference: Thread support library. http://en.cppreference.com/w/cpp/thread, 2016. Accessed: 2017-08-03. **[See section 1.]**

[6] Mehmet Dogon. Spurious wakeups are real! https://hazelcast.com/blog/spurious-wakeups-are-real-4/, 2014. Accessed: 2019-06-15. **[See section 7.3.]**

[7] The Open Group. The single unix ® specification, version 2: `pthread_cond_wait`, `pthread_cond_timedwait` - wait on a condition. https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_cond_wait.html, 1997. Accessed: 2019-06-15. **[See section 7.3.]**

[8] Heinz M. Kabutz. Timeout on console input. *The Java(tm) Specialists' Newsletter*, (153), 11 2007. Accessed: 2019-06-15. **[See section 8.5.]**

[9] Microsoft. .NET Documentation: Threading (C#). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/threading/index, 2017. Accessed: 2017-08-03. **[See section 1.]**

[10] Oracle. The Java™ Tutorials: Concurrency. https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html, 2016. Accessed: 2017-08-03. **[See section 1.]**

[11] Oracle. The Java™ Tutorials: Defining and Starting a Thread. https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html, 2016. Accessed: 2017-07-30. **[See section 3.]**

[12] Python Software Foundation. The Python Wiki: GlobalInterpreterLock. https://wiki.python.org/moin/GlobalInterpreterLock, 2017. Accessed: 2017-08-03. **[See section 1.]**