Ian Kenny

January 30, 2022

5CCS2OSC Operating Systems and Concurrency

Introduction to multithreading in Java

Multithreading

As we saw last week, applications can be single-threaded. This means there is only one thread of execution through the program. This was the traditional way of writing programs for decades, and is still common now.

However, in order to achieve concurrency, we can now create multithreaded programs: programs in which there are multiple threads of execution occurring ‘at the same time’.

Whether ‘at the same time’ means literally ‘at the same time’ depends on the availability of multiple cores in the processor running the application.

If there is a single core, then multithreading creates the illusion that tasks are occurring simultaneously. If there are multiple cores then it can be the case that the tasks are literally executing concurrently.

Threads in Java

Java provides core language support for threads. Indeed, all Java programs are threaded even if the programmer doesn't explicitly create threads.

When the user runs a Java program, the JVM itself creates a number of threads. It creates a Main thread to run the program itself but also creates a number of 'background' threads to deal with garbage collection, etc.

If the programmer adds a UI to their application, e.g. using JavaFX, then this will also be threaded even though the programmer doesn't need to explicitly do that.

Creating a thread class in Java

Typically, when using threads in Java, we want to create a new class that is capable of running code in a thread. There are a number of ways to create a new thread class in Java. We will now look at these.

The first approach is to create a new class that extends the `Thread` class and then to override the `run()` method from the `Thread` class in this new class.

Extending the Thread class

```
public class ExtendThreadExample extends Thread {

   @Override
   public void run() {

      System.out.println("ExtendThreadExample run method
          has started.");
      System.out.println("ExtendThreadExample run method
          has ended.");
   }

   public static void main(String[] args) {

      ExtendThreadExample thread = new
          ExtendThreadExample();

      thread.start();
   }
}
```

Listing 1: ExtendThreadExample.java

Extending the `Thread` class

This program creates an object of type `ExtendThreadExample`, which is a thread class, and then calls the `start()` method on that object. This causes the thread to start executing.

If we want our thread class to actually do anything, we need to override the `run()` method.

The `run()` method will be called, at some point, by the JVM.

The `run()` method of the `ExtendThreadExample` class simply prints out two lines of text to the console.

We will return to this topic but, for now, we note that when a thread reaches the end of its `run()` method it terminates.

Implementing `Runnable`

An alternative approach is to create a class that implements the `Runnable` interface.

With this approach we are forced to override the `run()` method (in the previous example that was optional (but necessary)) since `run()` is now an interface method.

The advantage to this approach is that we can make a class implement as many interfaces as we like (which is the only ‘multiple inheritance’ supported by Java). If we extend a class, on the other hand, we can’t extend another one. So, if we extend `Thread` then we can’t extend any other class which may be too much of a restriction. This topic is discussed further below.

Implementing Runnable

```java
public class RunnableExample implements Runnable {

   @Override
   public void run() {

      System.out.println("RunnableExample run method has
          started.");
      System.out.println("RunnableExample run method has
          ended.");
   }

   public static void main(String[] args) {

      Thread thread = new Thread(new RunnableExample());

      thread.start();
   }
}
```

Listing 2: RunnableExample.java

Implementing `Runnable`

In this case, to create the thread we use the `Thread` class. We pass to the `Thread` constructor an instance of our `Runnable` class.

We then call `start()` on our `Thread` object. This will cause the `RunnableExample run()` method to be called, at some point.

Using a lambda expression

Another approach is to implement the `Runnable` as a lambda expression. This only offers the advantage of a more concise syntax, as shown on the next slide.

`Runnable` is an example of what is called a *functional interface*: an interface with only one abstract method. You will notice in the example that the method name `run` is not mentioned anywhere. This is because it is implied by the fact that there is only one method in the interface (it is a functional interface) and that method does not receive any parameters (which is why the parentheses are empty).

Using a lambda expression

```
public class LambdaExample {

   public static void main(String[] args) {

      Runnable runnable = () -> {
         System.out.println("LambdaExample run method has
            started.");
         System.out.println("LambdaExample run method has
            ended.");
      };

      Thread thread = new Thread(runnable);

      thread.start();
   }
}
```

Listing 3: LambdaExample.java

Extend Thread or Implement Runnable

There are other reasons why you might implement `Runnable` rather than extend `Thread`.

When you extend `Thread` you are using that class directly to create a new thread. This means that each time you create a different thread object you get exactly that: a different thread, and the thread management code and thread behaviour are coupled in the same object. This means you can't share the behaviour part of the different threads: each has their own.

On the other hand, if you have a class that implements `Runnable` you can create an object of that type and pass the *same object* to multiple threads. This makes data sharing between the threads much easier since they are all accessing the same `Runnable` object.

Extend `Thread` or Implement `Runnable`

In general, this type of decoupling is a good thing anyway. In this case, the `Thread` class provides the vehicle and the `Runnable` provides the the behaviour.

Creating threads

Each of the examples shown above demonstrated the creation of a thread class and then the starting and running of that thread.

When we run these programs, each of them starts a Main thread and then the thread that we introduced ourselves (and others, as mentioned above).

The Main thread and our new thread are two independent threads. Neither owns or is owned by the other. The OS will manage both of them.

What if we want to create more than one of our own threads? We can just create them with more thread objects.

Multiple threads

```java
public class RunnableExample2 implements Runnable {

   @Override
   public void run() {

      System.out.println("Inside Thread: " +
          Thread.currentThread().getName());

      System.out.println("RunnableExample run method has
          started.");
      System.out.println("RunnableExample run method has
          ended.");
   }

   public static void main(String[] args) {

      Thread thread0 = new Thread(new RunnableExample2());

      thread0.start();

      Thread thread1 = new Thread(new RunnableExample2());

      thread1.start();
   }
}
```

Listing 4: RunnableExample2.java

Multiple threads

The output from this program is as follows. Is this what you would expect?

```
Inside Thread: Thread-0
RunnableExample run method has started.
RunnableExample run method has ended.
Inside Thread: Thread-1
RunnableExample run method has started.
RunnableExample run method has ended.
```

Multiple threads

What about this output from the same program?

```
Inside Thread: Thread-1
Inside Thread: Thread-0
RunnableExample run method has started.
RunnableExample run method has ended.
RunnableExample run method has started.
RunnableExample run method has ended.
```

Multiple threads

And this one?

```
Inside Thread: Thread-0
Inside Thread: Thread-1
RunnableExample run method has started.
RunnableExample run method has started.
RunnableExample run method has ended.
RunnableExample run method has ended.
```

Multiple threads

Without intervention, threads do not obey the order of execution that we might naively expect.

As stated earlier, each thread created in a Java program, hence by the JVM, is independent. Each is mapped to a kernel thread and then scheduled by the OS.

Not only do they not start in the order we might expect, their execution is interleaved too.

But, from what we know of CPU scheduling, that is what we would expect. The OS is here using preemptive scheduling so it interrupts one thread to run another.

Pausing a thread

One element of thread cooperation that is provided by the `Thread` class is the ability to put a thread to ‘sleep’, i.e. pause the thread.

You might do this because your thread needs to perform a regular task but doesn’t need to spend all that much time doing it. You acknowledge this fact by graciously relinquishing the CPU.

Another reason you might put a thread to sleep is for timing purposes. You may need a thread to wait for a length of time.

There is a method in the `Thread` class for this. It takes a parameter specifying the number of milliseconds that the thread should sleep for. The timings are approximate.

Pausing a thread

```java
public class SleepExample implements Runnable {

   @Override
   public void run() {

      System.out.println("SleepExample run method has
          started.");

      try {
         Thread.sleep(2000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }

      System.out.println("SleepExample run method has
          ended.");
   }

   public static void main(String[] args) {

      Thread thread = new Thread(new SleepExample());

      thread.start();
   }
}
```

Listing 5: SleepExample2.java

Pausing a thread

If you run the program on the previous slide, you will see that there is an approximately two-second pause between the two lines of console output.

Stopping a thread

We have seen how to start a thread, and how to pause one, but how do we stop a thread?

With the examples we have seen so far, there was no need to stop the threads because they terminated of their own accord.

As we know, when a Java program reaches the end of its `main()` method it terminates.

In the case of threads, when a thread reaches the end of its `run()` method it terminates.

If we want a thread to continue to execute, for some perhaps indeterminate period, we need to make sure it doesn't exit its `run()` method until we are ready.

Stopping a thread

In the `Thread` class there is actually a method called `stop()`. This method is deprecated, however.

The issue with `stop()`, and the reason it is considered best not to use it, is that it leaves your thread, hence your application, in an indeterminate state.

It is better to control the execution of your threads yourself, and to make sure they exit when you are ready for them to do so.

For this we can simply use a boolean. Consider the program on the next few slides.

Stopping a thread: the run() method

```
public void run() {

    System.out.println("SleepExample run method has
        started.");

    while (!stopped) {

        System.out.println("Executing run method.");

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    System.out.println("Stop has been requested.");
}
```

Listing 6: Stoppable.java

Stopping a thread: the boolean

```
private boolean stopped = false;

public boolean isStopped() {
   return stopped;
}

public void setStopped() {
   this.stopped = true;
}
```

Listing 7: Stoppable.java

Stopping a thread: the main() method

```
public static void main(String[] args) throws
    InterruptedException {

   Stoppable stoppable = new Stoppable();

   Thread thread = new Thread(stoppable);

   thread.start();

   Thread.sleep(5000);

   stoppable.setStopped();

   System.out.println("Exiting main method.");
}
```

Listing 8: Stoppable.java

The output

```
SleepExample run method has started.
Executing run method.
Executing run method.
Executing run method.
Executing run method.
Executing run method.
Exiting main method.
Stop has been requested.
```

Note the order in which the output statements are executed. The statement announcing the end of the `main()` method comes before that announcing that the request to stop the thread has been received. Is this what you would have expected?

Thread independence

Although the previous example doesn't offer conclusive proof of this, it does suggest that the Main thread does not ‘own’ the thread created in the line:

```
Thread thread = new Thread(stoppable);
```

This is because the Main thread appears to exit first. The threads might exit the other way round but, either way, they are independent of each other.

The JVM will continue to run any threads that have not exited, even in the case where the Main thread has exited. The Main thread is ‘just another thread’. This can be counter-intuitive at first.

The next example program demonstrates this more conclusively.

Thread independence

```java
public class IndependenceDemo implements Runnable {

   public void run() {

      while (true) {

         System.out.println("Executing thread run method.");

         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
   }

   public static void main(String[] args) throws InterruptedException {

      IndependenceDemo stoppable = new IndependenceDemo();
      Thread thread = new Thread(stoppable);
      thread.start();
      Thread.sleep(4000);
      System.out.println("Exiting main method.");
   }
}
```

Listing 9: IndependenceDemo.java

Output (sample)

```
Executing thread run method.
Executing thread run method.
Executing thread run method.
Executing thread run method.
Exiting main method.
Executing thread run method.
Executing thread run method.
Executing thread run method.
(continues)
```

Thread independence

If we want to ensure that a thread exits when the Main thread exits then we can make it a *daemon* thread.

If we call[1]:

```
thread.setDaemon(true);
```

Before the thread is started, this will ensure that when the Main thread exits, so does the thread.

[1] Assuming our thread reference is called `thread`.

`Thread.join()`

It is sometimes the case that there needs to be some coordination between threads in terms of their execution. As we have seen, the default case is no coordination.

However, one thread might need to wait until another thread has completed. For example, if the second thread is computing some value that the first thread needs and can't proceed without.

In this case we can use the `join()` method. If thread A calls `join()` on thread B, thread A will block until thread B terminates.

In the example on the next slide, the main thread will wait until the thread (called `thread`) has terminated. It will block at the line `thread.join()`.

Thread.join()

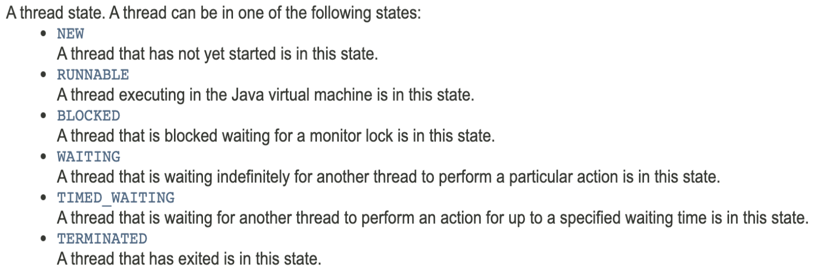
```java
public class JoinExample implements Runnable {
   private float result;
   public void run() {
      float temp = 0.0f;
      float inc = 0.001f;

      for (int i = 0; i < 1000; i++) {
         temp += inc;
         try {
            Thread.sleep(10);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      result = temp;
   }
   public float getResult() {
      return result;
   }
   public static void main(String[] args) throws InterruptedException {
      JoinExample runnable = new JoinExample();
      Thread thread = new Thread(runnable);
      thread.start();
      System.out.println("Waiting for thread to terminate.");
      thread.join();
      System.out.println("Thread terminated.");
   }
}
```

Listing 10: JoinExample.java

Thread states in the JVM

The JVM defines its own set of thread states that do not typically reflect the OS thread states (Runnable = is running!)



A thread state. A thread can be in one of the following states:

- `NEW`
  A thread that has not yet started is in this state.
- `RUNNABLE`
  A thread executing in the Java virtual machine is in this state.
- `BLOCKED`
  A thread that is blocked waiting for a monitor lock is in this state.
- `WAITING`
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- `TIMED_WAITING`
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- `TERMINATED`
  A thread that has exited is in this state.

source: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html

The Java Memory Model

Before we can discuss some of the issues with concurrency, and in Java in particular, we need to consider the Java Memory Model, i.e. where Java puts stuff in the RAM.

We already discussed this topic to some extent in the previous week, but we will now consider it in more detail because it matters greatly.

The Java Memory Model

We already discussed how when a process is started it gets a number of memory segments. The code and the global data/constants are copied from the executable file, and the OS allocates two areas of memory exclusive to the process: the *stack* and the *heap*.

When a process creates threads, each thread gets its own, exclusive stack, but shares the heap with the process and all other threads started by the process.

This sharing of the heap means that we can very easily share data between threads.

Primitives, references and objects

When you create a *primitive* type variable (`int`, `boolean`, etc.) as a local variable or method parameter, these are stored on the stack of the running thread.

When you create an *object* you typically do something like this:

```
Operation op = new Operation(2, 5, "*");
```

This creates two things: a reference and the object itself. If you create an object in this manner in a method then the reference will be stored on the *stack* but the object itself will be stored on the *heap*.

If you pass a reference to an object as a method parameter, again, the reference goes on the stack but the object itself will be on the heap.

Primitives, references and objects

However, objects often *contain* primitive values too. Think of a `Person` object that contains the person's age as an `int`, for example.

This primitive value is a part of the object so it stored on the heap, with the rest of the object's fields.

It is also typical for objects to contain references to other objects. In this case, the reference is stored, **as part of the object**, on the heap. The object that the reference is referring to will be stored *elsewhere on the heap*.

The `Evaluator` program

We will now look at at very simple program that illustrates some of the ideas above.

This program is not presented as an ideal solution to this particular problem. It is designed to illustrate various things, that's all.

The `Evaluator` is a very simple (and unfinished) program to do simple binary arithmetic operations.

We will now do a walkthrough of where the data is stored in this program. We will not do every item of data but enough to explain the various cases.

The Operation class

The Operation class represents a simple binary arithmetic operation.

```java
public class Operation {
   private int value1;
   private int value2;
   private String operator = "NOT_SET";

   public Operation (int v1, int v2, String op) {
      value1 = v1;
      value2 = v2;
      operator = op;
   }
   public int getValue1() {
      return value1;
   }
   public int getValue2() {
      return value2;
   }
   public String getOperator() {
      return operator;
   }
}
```

Listing 11: Operation.java

The Evaluator class

The Evaluator class evaluates operations.

```java
public class Evaluator {

   private Operation op;

   public Evaluator(Operation op) {
      this.op = op;
   }
   public float evaluate(Message message) {

      float result = Float.MIN_VALUE;

      message.setMessage("NOT_SUPPORTED"); // default message

      if (op.getOperator().equals("*")) {
         result = op.getValue1() * op.getValue2();
         message.setMessage("SUCCESS");
      }
      return result;
   }
}
```

Listing 12: Evaluator.java

The Message class

The Message class represents a simple string message.

```
public class Message {
   private String message;

   public Message() {
      message = "NOT_SET";
   }
   public String getMessage() {
      return message;
   }
   public void setMessage(String m) {
      message = m;
   }
}
```

Listing 13: Message.java

The program

The program below simply uses the classes above to perform a simple operation.

```
public class Main {
   public static void main(String[] args) {
      int op1 = 2;
      int op2 = 5;
      String operation = "*";

      Operation op = new Operation(op1, op2, operation);

      Message message = new Message();

      float result = new Evaluator(op).evaluate(message);

      System.out.println(result + " " + message.getMessage());
   }
}
```

Listing 14: Main.java

The Java Memory Model

Let's start with the `Main` class which contains the `main()` method.

The JVM will start a thread to run this program. The `main()` method is just that: a method. This means that variable declarations made in the `main()` method are local to that method. So this line:

```java
    Operation op = new Operation(op1, op2, operation);
```

Listing 15: Main.java

Creates a reference called op that is stored on the stack, and an object of type `Operation` that is stored on the heap.

This line of code contains a method call itself, so the values of op1 and op2 are copied to the stack. The `String` reference `operation` is also copied to the stack. The value "*" will be stored on the heap (since it is an object).

The Java Memory Model

Let's now look at the Operation class. The following three instance variables are declared at class scope therefore are a part of the object and will be put on the heap as part of the object's memory footprint:

```
    private int value1;
    private int value2;
    private String operator = "NOT_SET";
```

Listing 16: Operation.java

So, the String reference operator is placed on the heap with the other variables. The String literal "NOT_SET", which is an object, will also be placed on the heap but in an area reserved for String literals.

The Java Memory Model

Now consider the `Operator` constructor. This method was called from the `main()` method, as we just saw.

```
public Operation (int v1, int v2, String op) {
   value1 = v1;
   value2 = v2;
   operator = op;
}
```

Listing 17: Operation.java

So, as discussed, the three parameters have been pushed onto the stack (in the case of the `String` parameter, only the reference has been pushed onto the stack, not the object).

This method does not create any further local variables so all that happens is the three assignment operations. This means that the values in the object (`this`) are changed (on the heap) to the values in the three parameters.

The Java Memory Model

The next line of code in the `main()` method simply creates a reference to an object of type `Message`. Consider the next line after that.

```
    float result = new Evaluator(op).evaluate(message);
```

Listing 18: Main.java

This line creates a local variable `result` that will be stored on the stack. It also creates a new `Evaluator` which is stored on the heap (the address of this object is known to the JVM, of course, despite there being no reference created), and then the `evaluate()` method is called with the `message` parameter the purpose of which is to get a success or error message back from the method.

The Java Memory Model

Finally, consider the `evaluate()` method.

```
public float evaluate(Message message) {

   float result = Float.MIN_VALUE;

   message.setMessage("NOT_SUPPORTED"); // default message

   if (op.getOperator().equals("*")) {
      result = op.getValue1() * op.getValue2();
      message.setMessage("SUCCESS");
   }
   return result;
}
```

Listing 19: Evaluator.java

The Java Memory Model

The final point to note is that this method receives a `Message` object reference. The reference is pushed onto the stack whilst, as we now know, the object itself is stored on the heap. The reference `message` is actually a copy of the reference. Since a reference is an address of an object, this means that the method receives the address of the object.

The effect of this is when this method calls a mutator method using that reference, as it does with `message.setMessage(‘‘SUCCESS’’)`, the object itself is changed.

Thus we are able to create an object outside of the `evaluate()` method, pass it to the method, and then see the results of changes to that object after the `evaluate()` method has ended, back in the `main()` method.

Implications for threads

We will now look at the implications of this memory model on the operation of threads.

Consider the program on the next slide. This program contains a class that implements the `Runnable` interface, therefore, has a `run()` method. Inside the `run()` method is a loop that simply iterates 100,000 times whilst incrementing a counter each iteration. Once the loop has finished, the method prints out the value of the counter and the thread name.

In the `main()` method, the program creates two `Runnable` objects and passes each of them to a separate thread, and then starts both threads.

Implications for threads

```java
public class MemoryDemoRunnable implements Runnable {
   private int counter = 0;
   public void run() {
      for (int i = 0; i < 100_000; i++) {
         counter++;
      }
      System.out.println(Thread.currentThread().getName()
          + " " + counter);
   }
   public static void main(String[] args) {
      MemoryDemoRunnable runnable1 = new
          MemoryDemoRunnable();
      Thread thread1 = new Thread(runnable1);
      MemoryDemoRunnable runnable2 = new
          MemoryDemoRunnable();
      Thread thread2 = new Thread(runnable2);
      thread1.start();
      thread2.start();
   }
}
```

Listing 20: MemoryDemoRunnable.java

Implications for threads

As you may have expected, the output is as follows:

```
Thread-0 100000
Thread-1 100000
```

Or

```
Thread-1 100000
Thread-0 100000
```

Implications for threads

In the main thread we see that two objects of type `Runnable` are created. These objects will be stored on the heap for the process, whilst the two references are stored on the stack of the main thread.

Consider the layout of `MemoryDemoRunnable` itself. It has a `run()` method that creates a local variable called `i`, that is used as an index for the loop, and is stored on the stack of whichever thread is executing (`Thread-0` or `Thread-1`).

The two threads have different instances of `i`; distinct to each other and distinct between `run()` method calls even of the same thread.

Implications for threads

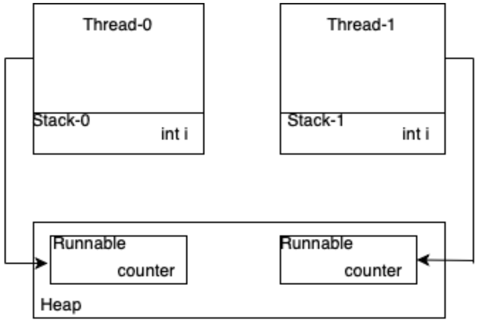
Now consider the variable called `counter`.

This variable is an instance variable so it is not a local variable but it is distinct between different objects of `MemoryDemoRunnable` class.

However, between different calls to `run()` the value of `counter` persists.

The crucial point here is that the variable counter is **not** being shared by the two threads since they are running distinct `Runnable` objects.

A pictorial representation



Main thread not shown.

Shared objects

Let's consider what happens if objects are shared between threads. In this case, the `Runnable` object.

We might wish both threads to use the same `Runnable` object if, for example, they are cooperating on some task.

We make the simple change to the program shown on the next slide. Now only one `Runnable` object is created.

Shared objects

```
public class MemoryDemoRunnable2 implements Runnable {
   private int counter = 0;
   public void run() {
      for (int i = 0; i < 100_000; i++) {
         counter++;
      }
      System.out.println(Thread.currentThread().getName()
          + " " + counter);
   }
   public static void main(String[] args) {
      MemoryDemoRunnable2 runnable1 = new
          MemoryDemoRunnable2();
      Thread thread1 = new Thread(runnable1);
      Thread thread2 = new Thread(runnable1);
      thread1.start();
      thread2.start();
   }
}
```

Listing 21: MemoryDemoRunnable2.java

Shared objects

If we consider the `run()` method again, we know that the local variable `i` is not shared so won't cause any issues in terms of threads both accessing the same variable.

However, the `counter` variable is now shared between threads since they now share the same `Runnable` object.

What effect do you think this will have on the operation and output of the program?

Shared objects

Potentially, we could expect the counter to reach 200,000 since each thread is independently iterating 100,000 times and there are two threads.

In practice, this won't happen. The problem is that the two threads will interfere with each other. The value of the counter will depend on when each of the two threads access it and change its value.

Non-atomic operations

The operation `counter++` looks atomic, i.e. it looks like it can't be interrupted or split in any way. This isn't true though.

The ++ operator involves three separate operations: read the value, increment the value, write the value.

This is a problem because, in this example, the threads might overlap in their execution of `counter++` due to them being interrupted by each other.

Race conditions

In our example, Thread-0 could initiate the `counter++` operation and read the existing value. At this point it could be interrupted by Thread-1 which also starts the operation and reads the existing value: the same value that Thread-0 read.

Thread-1 then increments the counter and writes the value back to memory. At this point, Thread-0 resumes, increments **its** value of the counter and writes that back. This causes a **lost update** because Thread-0 overwrites the value created by Thread-1.

If the two threads had executed sequentially (which we assume for now was the logic of this operation rather than a mistake), then we would expect *both* increments to be recognised and the value of the counter to have increased by two.

Race conditions

Such a scenario is called a **race condition** and is a common issue in computer science. It is a race because the end result of an operation depends on a race between the threads, rather than on the logic of the program.

Now let's look at some example outputs from our most recent program.

Race conditions

```
Thread-0 125520
Thread-1 129595

Thread-1 107462
Thread-0 137097

Thread-0 148311
Thread-1 148311
```

We can see that the result is indeterminate. We can't be sure what the result will be because it depends on the order in which the threads perform their operations.

Synchronisation

In the next lecture we will take more about this problem, and some simple and not-so-simple ways in which we can solve it.