PHAS3459: Scientific Programming Using Object Oriented Languages

Module 8: Threads

Dr. B. Waugh and Dr. S. Jolly
Department of Physics & Astronomy, UCL

Contents

1. Aims of Module 8

This module introduces *threads*. After this module you should be able to write programs that perform multiple tasks in parallel, and understand some of the additional pitfalls that can occur in multi-threaded code.

2. Threads

A *thread* is a single sequential flow of control within a program. The programs we have seen so far have only one thread, that controlled by the `main` method. This may include loops and call other methods, but only one statement is executed at a time, and there is never any ambiguity about the order in which the statements will be executed.

The concept of a thread becomes more interesting and useful when we use several threads in the same program. This is called *concurrency* and is a powerful technique: we can have many pieces of code operating simultaneously, doing different things, under the control of one master program. Each thread has a start point, an execution sequence and an end, but cannot run on its own: it must be spawned from a controlling program or from another thread.

If our program is running on a computer with several processors, or several processor cores, then the different threads may literally be doing different things at the same time. On a computer with only a single processor it is still possible to have several threads, but in this case the operating system will use *time-slicing* to give turns to the threads so that the processor runs a few instructions from one thread before turning to the next thread and then the next.

Many applications are multi-threaded, particularly graphical ones. For example, a web browser may have one thread controlling the downloading of JPEG images, another for scrolling the page, a thread responding to keyboard input, and so on.

Using multiple threads can sometimes simplify the architecture of a program by separating tasks that can be carried out independently: the programmer doesn't need to decide explicitly when to switch between tasks. It is often useful to be able to respond quickly to input from a user while carrying out time-consuming calculations in the background. One thread can wait for data to arrive or for events to occur, such as a mouse click or the entry of a string from the keyboard, without delaying other operations. Another thread may perform actions that need to take place repeatedly, perhaps at pre-defined intervals.

Multi-threaded programming is becoming increasingly important as computer manufacturers start to put ever larger numbers of processor cores into their machines. Each of these cores may in fact run more slowly that earlier high-speed processors, but applications can run faster by exploiting the larger number of tasks that can be carried out simultaneously.

However, there are also disadvantages. In some cases it can be difficult to break a process down into threads that can be executed separately, and the result may be a much more complicated program that is harder to understand and modify. There are also new types of bugs that don't occur in single-threaded programs. Data can be corrupted if the same variables are accessed simultaneously by more than one thread, and careful use of *synchronization*, explained later in this module, is needed to avoid this. It is also possible for a program to get into a state of *deadlock*, where each thread is waiting for another thread to do something before it can proceed, so none can get any further and the program freezes. It is generally wise to avoid using threads unless there is a real advantage to doing so!

3. The Life Cycle of a Thread

To use threads effectively you need to understand their life-cycle, which is described in this section. On the other hand, it can be hard to understand the life-cycle without trying out an example or two, so you may want to move back and forth between this section and the next.

At any given time a thread can be in one of a number of states. When it is first created its status is *new*. When its `start` method is called it enters the *runnable* state. While in the *runnable* state it may be actually *running*, or it may be *ready* but waiting for the scheduler to give it a turn using the processor. A thread can also move from its *running* state into a *waiting* state, while it waits for another thread to complete a particular action, a *sleeping* state while it pauses for a set interval, or a *blocked* state while it waits for another thread to exit a `synchronized` piece of code (see the Section on Synchronization).

You may see other terms used to describe the set of states a Java thread can be in, but they are all different ways of describing the same processes. We have omitted the *suspended* state and the `suspend` and `resume` methods since these are not safe to use and are *deprecated*: it is recommended by the maintainers of the Java language that you should not use them, and they may be removed from future versions of the language.

4. Creating a Thread

A thread cannot be made active unless it has a defined `run()` method. There are two ways to do this. The first is to extend the `Thread` class and override its `run` method:

```
public class SimpleThread extends Thread {
  public SimpleThread(String name) { super(name); }
  public void run() {
    System.out.println("Thread is now alive: "+getName());
  }
}
```

To use this class, we create an instance and then call its `start()` method. This is a method provided by the `Thread` class that will call our `run()` method *on a separate thread*. If we mistakenly call the `run()` method directly, it will execute on the current thread, and the calling code will not proceed until the `run()` method has returned.

```
SimpleThread newThread = new SimpleThread("first thread");
newThread.start();
```

The other way, which we will use in this course, is to implement the `Runnable` interface and pass an instance of your class to the `Thread` constructor:

```
public class SimpleTask implements Runnable {
  private String _name;
  public SimpleTask(String name) {_name = name;}
  public void run() {
    System.out.println("Thread is now alive: "+_name);
  }
}
```

```
SimpleTask task = new SimpleTask("second thread");
Thread newThread = new Thread(task);
newThread.start();
```

The first method can be simpler in some cases, but the second is more generally useful and allows the class in question to extend a class other than `Thread`. This is often a requirement, particularly in graphical programs where we may want to extend an existing graphical class such as `Applet` (see Module 9).

Let's start with a relatively simple example that starts two threads that will run in parallel. One thread will print odd numbers while the other prints even numbers. The order in which the numbers appear on the screen will depend on how the threads are interleaved and may be different each time you run the program. First we write a `Runnable` class that will print a series of numbers to the screen:

```
public class NumberSeriesTask implements Runnable {
  private int start;
  private int end;
  private int step;
  public NumberSeriesTask(int start, int end, int step) {
    this.start = start;
    this.end = end;
    this.step = step;
  }
  public void run() {
    for (int i=start; i<end; i+=step) {
      System.out.println(i);
    }
  }
}
```

Our main program will include this code to start the two threads:

```
Thread odds = new Thread(new NumberSeriesTask(1,1000,2));
Thread evens = new Thread(new NumberSeriesTask(2,1000,2));
odds.start();
evens.start();
```

5. Interrupting a Thread

Sometimes we need to tell a thread to stop what it is doing and do something else instead. Typically this is used when we want to cancel the task the thread is performing before it has completed, and this is the case we will deal with here. This may be because a user has clicked the "cancel" button in the user interface, because the program itself is shutting down, because a certain time has elapsed, or for any of a variety of other possible reasons.

Java provides a mechanism called *thread interruption* to allow the controlling method to send a signal to a thread, asking it to stop the task it is performing and terminate. This signal is only a request: the `stop` method of the `Thread` class, which forces a thread to terminate immediately, can easily lead to malfunctioning code and is now deprecated[1]. If you may need to interrupt a thread, it is up to you to ensure that the task it is running, including the `run` method as well as any methods it calls, will respond appropriately to this signal.

The way to detect when an interrupt signal has been sent to the thread in which your code is running is to check the return value of `Thread.currentThread().isInterrupted()`. If this returns `true`, your method should return as soon as possible, after any necessary tidying up such as closing open files. This check should be carried out quite frequently, especially before starting a time-consuming operation. Often this is checked within a loop.

Some methods in the Java API are *blocking*: they stop execution of a thread from proceeding until some condition is satisfied. This may involve waiting for input from a device, or waiting for a given time to elapse. This obviously means that the interrupt status of the thread cannot be checked during this time. In the case of input and output methods there is no easy way round this. Fortunately, the `Thread.sleep` method, which causes the current thread to enter a *sleeping* state for the given number of milliseconds before resuming its execution, and which you will see and use often in Java programs that use threads, does respond to interruption. It checks regularly to see if the current thread has been interrupted, and if it has it terminates early by throwing an `InterruptedException`. The calling code can then catch this exception and respond appropriately. Since the interrupt status of the thread is cleared when the `InterruptedException` is thrown, any code subsequently executed in the same thread will not know that it needs to finish early unless the interrupt status is turned back on using `Thread.currentThread().interrupt()`. This is not necessary if we know that we have anyway reached the end of the program or the `run` method of the thread's `Runnable` object.

Now we know enough to modify our thread example so that it will run for a fixed time instead of a given number of iterations. Our `NumberSeries` class while run forever unless it is interrupted:

```
public class NumberSeries implements Runnable {
  private int start;
  private int step;
  public NumberSeries(int start, int step) {
    this.start = start;
    this.step = step;
  }
  public void run() {
    int i = start;
    while (true) {  // run until interrupted
      if (Thread.currentThread().isInterrupted()) return;
      System.out.println(i);
      i += step;
    }
  }
}
```

Our main program will let both threads run for five seconds before interrupting them:

```
Thread odds = new Thread(new NumberSeries(1,2));
Thread evens = new Thread(new NumberSeries(2,2));
odds.start();
evens.start();
try {
  Thread.sleep(5000); // main thread pauses for 5 seconds
                      // while other threads run
}
catch (InterruptedException e) {
  // stop early if main thread is interrupted
}
odds.interrupt();
evens.interrupt();
```

If the main program did not interrupt the other threads in this example, the program would keep running indefinitely. The rule is that the program will not exit until all *user threads* have terminated. All the threads used in this course will be user threads: both the main thread and those we create explicitly in our code. The other type of thread is called a *daemon thread* and can be used for recurring tasks that must be run as long as the program is running, but should not in themselves prevent it from terminating. They will not be discussed further here.

---

[1] For a detailed explanation of why `Thread.stop` and related methods are unsafe see https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency /threadPrimitiveDeprecation.html. ↩

---

6. Waiting for a Thread to Finish

In some cases we might want to wait until one thread has finished before continuing with a related task. In this case we can use the `join()` method of the thread we are waiting for. However, the `join()` method may throw an exception if it is interrupted, so we have to be prepared to catch this.

```
longCalculationThread.start();

// Some other time-consuming calculations here...

// Now we want to wait for the other thread to finish:
try {
  longCalculationThread.join();
} catch (InterruptedException e) {
  // Current thread was interrupted, so might want to
  // return early, or perhaps even try interrupting
  // longCalculationThread
}
// Now we can continue knowing that longCalculationThread
// has finished its job.
```

7. Synchronization

If we have several threads that are completely independent of one another then the order in which instructions from the different threads are executed doesn't really matter. However, we often want threads to cooperate in some way to achieve an overall task, and at some point this usually involves several threads accessing the same variable, object, file, screen or some other resource. This can lead to *thread interference* when instructions from different threads, acting on the same data, are *interleaved*. This can lead to corruption of data and to unexpected outcomes.

The following example illustrates what can happen when two threads use the same object:

```
public class Counter {
  private int value = 1;
  public Counter() {}
  public void count() {
    int limit = value + 1000;
    while (value < limit) {
      System.out.println(value);
      value++;
    }
  }
}
```

```
public class CounterThread implements Runnable {
  private Counter c;
  private Thread t;

  public CounterThread (Counter c) { this.c = c; }

  public void start() {
    if (t == null) {
      t = new Thread(this,"counter");
      t.start();
    }
  }

  public void run() {
    if ( t!= null) {c.count();}
  }
}
```

```
Counter c1 = new Counter(); Counter c2 = new Counter();
CounterThread ct1 = new CounterThread(c1);
CounterThread ct2 = new CounterThread(c1); // use same Counter
// Or use different Counters :
// CounterThread ct2 = new CounterThread(c2);
ct1.start();
ct2.start();
```

In the above example, two threads are modifying the `count` and `limit` variables in an indeterminate way. One thread starts; then some time later the second thread starts and modifies the same variables that the first thread is using. Precisely when the variables are modified depends on the scheduling of when the threads are run, so the pattern of numbers in the output may be different on different computers or even when the program is run twice.

We could avoid this problem by creating a separate `Counter` object for each thread. However, it is often necessary for several threads to access the same object, and we need a way of making sure that only one thread at a time can make changes to the object.

This is done using the `synchronized` keyword. We can declare a method `synchronized`:

```
public synchronized void count() {
  int limit = value + 1000;
  while (value < limit) {
    System.out.println(value);
    value++;
  }
}
```

This means that while one thread (say thread A) is running the method, no other thread can also be doing so. When thread A enters a `synchronized` method, it obtains a *lock* on the object to which the method belongs. When thread B reaches a call to the same method, or another `synchronized` method of the same object, it tries to obtain a lock but has to wait for the lock to be released by thread A. It enters a *blocked* state while it waits.

Similarly, if a `static` method is declared `synchronized`, it will obtain a lock on the *class* rather than on a particular instance. This ensure that only one synchronized static method of each class can run at one time, but it does not stop a synchronized *static* method from running at the same time as a synchronized *non-static* method.

There are some disadvantages of using `synchronized` methods. There is a processing *overhead*: the computer takes some extra time to check if a lock is available and obtain it if it is needed. Since a `synchronized` method can only be used by one thread at a time, it can also act as a bottleneck if it is called often on different threads, so should be avoided where the reason for using multiple threads is to achieve greater performance. A significant risk when using synchronization is that a program can enter a state of *deadlock*, where two or more threads are each waiting each other to release a lock, so none of them can proceed. Some software packages, including the Java API, provide *thread-safe* classes, which use synchronization and other more sophisticated techniques to guarantee consistency while maintaining high performance when accessing objects from multiple threads, as well as *unsynchronized* versions that provide higher performance when synchronization is not needed.

Synchronization is often used for `put` and `get` methods to ensure that data is not modified at the same time as it is being read.

8. Immutable Objects

Because of the disadvantages of using synchronization, a technique that is particularly useful in concurrent applications is the use of *immutable* objects. If an object never changes after it has been created, it can never be found in an inconsistent state, not matter how many different threads are sharing it.

There is no simple way of guaranteeing that the instances of a class are immutable in Java, since even if all member variables are made `final` to ensure they cannot be changed, they may refer to collections the elements of which *can* be modified. Thus there is no substitute for careful thought, but there are some rules that should be followed for a start:-

- Member variables should be declared `final`.
- Member variables that cannot be made `final`, or refer to objects (e.g. collections) that are themselves mutable, should be made `private` and only accessed by methods that do not modify their contents.
- To be safer, subclasses should be prevented from overriding methods, e.g. by making the class itself `final`.

One clear consequence is that to be immutable, a class should have no setter methods: all member variables should be set once in the constructor and not changed thereafter.

9. High-Level Concurrency Objects

Recent versions of Java, from version 5.0, include a variety of classes in the `java.util.concurrent` package and its subpackages. These go beyond the basic building blocks we are using in this module, and are useful for more demanding applications such as those that use multiple threads to process large numbers of similar tasks. *Lock objects* and *atomic variables* provide alternative ways of synchronizing the actions of several threads. The *concurrent collections* provide collection classes such as a `Map` that can be be used efficiently and safely by multiple threads simultaneously. *Executors* provide a way of managing threads to carry out large numbers of tasks, for example by maintaining a *pool* of threads.

In this section we will show an example of using multiple threads to perform a calculation in parallel, in an attempt to get the result sooner. The algorithm we will use is a (rather inefficient) way of calculating the value of pi using a Monte Carlo method. We generate points that are randomly distributed in a unit square, and calculate the fraction of points that lie within unit distance of the origin. Given enough points, this ratio should be pi/4, so we simply multiply the ratio by 4 to get an estimate of pi.

```
Random rand = new Random();
long nPoints = 10000000L;
long nIn  = 0;
for (long iPoint = 0; iPoint < nPoints; ++iPoint) {
  double x = rand.nextDouble();
  double y = rand.nextDouble();
  double r2 = x*x + y*y;
  if (r2 < 1.0) ++nIn;
}
double pi = 4.0 * nIn / nPoints;
```

In order to parallelise this calculation, we will simply divide the total number of points among several threads. Each thread will calculate a separate estimate of pi, and we will take the mean of these as our overall results. We could create a class that implements `Runnable` as above, but we also need to return a result from the calculation so instead we will implement the `Callable` interface.

```
public class MonteCarloPiCalculatorTask implements Callable<Double> {
  private final long n_points;

  public MonteCarloPiCalculatorTask(long nPoints) {
    this.n_points = nPoints;
  }

  @Override
  public Double call() {
    Random rand = new Random();
    long n_in  = 0;
    for (long iPoint = 0; iPoint < n_points; ++iPoint) {
      double x = rand.nextDouble();
      double y = rand.nextDouble();
      double r2 = x*x + y*y;
      if (r2 < 1.0) ++n_in;
    }
    return 4.0 * n_in / n_points;
  }
}
```

We can test this by calling it in a simple serial program:

```
  long nPoints = 10000000L;
  MonteCarloPiCalculatorTask task = new
MonteCarloPiCalculatorTask(nPoints);
  double pi = task.call();
  System.out.println(pi);
```

Using it in a parallel program is more complicated. We create a *thread pool*, represented by an `ExecutorService` object, and use the `Future` class to represent the result of each of the parallel calculations.

```
  long nPoints  = 10000000L;
  int  nThreads = 4;
  ExecutorService threadPool = Executors.newFixedThreadPool(nThreads);
  List<Future<Double>> futures = new ArrayList<Future<Double>>();
  for (int iThread = 0; iThread < nThreads; ++iThread) {
    MonteCarloPiCalculatorTask task = new
MonteCarloPiCalculatorTask(nPoints/nThreads);
    Future<Double> future = threadPool.submit(task);
    futures.add(future);
  }
  double sum = 0.0;
  for (int iThread = 0; iThread < nThreads; ++iThread) {
    double result = futures.get(iThread).get();
    sum += result;
  }
  threadPool.shutdown();
  double pi = sum/nThreads;
```

10. Priorities

It is possible to give some threads a higher priority, for example to prioritise those that interact with the user over background calculations. In general this is unnecessary, and is not guaranteed to result in the same behaviour on different systems, or even to make any difference at all. In the worst case it is possible that boosting the priority of one thread could stop another from ever executing. Therefore it is a good idea only to adjust thread priorities if testing shows that there is a real problem with using the default priority everywhere. This may be the case in some applications where it is vital to provide a responsive user interface while carrying out a lot of computation in the background.

The *priority* of a thread is given by an integer value from 1 (lowest) to 10 (highest), with 5 being the default. There are mnemonic constants defined in the `Thread` class: `Thread.MIN_PRIORITY`, `Thread.NORM_PRIORITY` and `Thread.MAX_PRIORITY`.

```
Thread t1 = new Thread(someRunnableObject, "T1");
t1.setPriority(Thread.MAX_PRIORITY);
t1.start();
```

11. Summary

This module has introduced the basics of multi-threaded programming, and outlined some of the problems that can occur. There are many more techniques that are not covered here, but which you may want to investigate if you ever need to write code that uses threads.