MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Java Concurrency - Part 1

Daniel Playne
d.p.playne@massey.ac.nz

Java Concurrency

In Java, all programs are executed within a Java Virtual Machine.

A *process* in Java corresponds to running a JVM and interprocess communication faces similar challenges to native applications.

Threads execute within the same JVM and are able to access shared objects within that JVM.

Java Concurrency - Threads

As with (almost) everything in Java, threads are represented by a class `java.lang.Thread`.

The function `run()` can be overridden and will be executed whenever the virtual machine creates a new thread.

```
public class MyThread extends Thread {
    public void run() {
        ...
    }
    ...
}
```

Java Concurrency - Threads

Note that the function run() is not called directly, the virtual machine has to set up the thread for execution before run() can be called.

Instead the function start() is called to tell the virtual machine to set up the thread, after which it will call run().

```
public class Example01Thread extends Thread {
   public static void main(String args[]) {
      Thread t = new Example01Thread();
      t.start();
      System.out.println("Hello from Main Thread");
   }
   public void run() {
      System.out.println("Hello from Example01Thread");
   }
}
```

Java Concurrency - Runnable

An alternative (and generally preferable) method for using threads is to implement the `Runnable` interface.

This allows us to use threads without forcing our classes to be a subclass of `Thread` (and due to single inheritance not a subclass of anything else).

It also required to use some of the more advanced concurrency features in Java.

Java Concurrency - Runnable

```
public class Example02Runnable implements Runnable {
   public static void main(String args[]) {
      Thread t = new Thread(new Example02Runnable());
      t.start();
      System.out.println("Hello from Main Thread");
   }

   public void run() {
      System.out.println("Hello from Example02Runnable");
   }
}
```

Java Concurrency - Interrupts

Threads can be set to wait for a given period of time (given in milliseconds) using `Thread.sleep()`.

A thread may be interrupted by another thread calling `thread.interrupt()`.

This can be detected by catching an `InterruptedExecption`.

Java Concurrency - Interrupts

```
public class Example03Interrupts implements Runnable {
   public static void main(String args[]) throws InterruptedException {
      Thread t = new Thread(new Example03Interrupts());
      t.start();

      System.out.println("Main Thread: sleeping for 5 seconds.");
      Thread.sleep(5000);

      System.out.println("Main Thread: interrupting Thread.");
      t.interrupt();

      System.out.println("Main Thread: completed.");
   }

   public void run() {
      System.out.println("Thread: sleeping for " +
         (Long.MAX_VALUE / (1000 * 60 * 60 * 24 * 365)) + " years.");

      try {
         Thread.sleep(Long.MAX_VALUE);
      } catch(InterruptedException exception) {
         System.out.println("Thread: interrupted.");
      }

      System.out.println("Thread: completed.");
   }
}
```

Java Concurrency

Like other threading libraries, the main thread can ensure that a child thread has terminated by calling `join`.

```
public class Example04Join implements Runnable {
   public static void main(String args[]) {
      Thread t = new Thread(new Example04Join());
      t.start();

      try {
         System.out.println("Main Thread: joining Thread");
         t.join();
      } catch(InterruptedException exception) {
         System.out.println("Main Thread: interrupted");
      }

      System.out.println("Main Thread: completed");
   }

   public void run() {
      System.out.println("Thread: sleeping");

      try {
         Thread.sleep(5000);
      } catch(InterruptedException exception) {
         System.out.println("Thread: interrupted");
      }
      System.out.println("Thread: completed");
   }
}
```

Java Concurrency - Synchronisation

To synchronise access to shared resources, Java has a keyword called `synchronized`.

This keyword allows methods or a block of code to be restricted so that it can only be executed by one thread at a time.

Java takes care of managing a lock that each thread must get before it enters the block of code and releasing it after it leaves.

Java Concurrency - Synchronisation

Consider the following program that uses a shared object `Counter` which provides two main functions `increment()` and `decrement()`.

```
public class Example05Synchronized implements Runnable {
   private static Counter mCounter = new BasicCounter();

   public static void main(String args[]) throws InterruptedException {
      System.out.println("Creating Thread");
      Thread t = new Thread(new Example05Synchronized());
      t.start();
      for(int i = 0; i < 1000; i++) {
         mCounter.decrement();
      }
      t.join();
      System.out.println("Counter value: " + mCounter.value());
   }

   public void run() {
      for(int i = 0; i < 1000; i++) {
         mCounter.increment();
      }
   }
}
```

Java Concurrency - Synchronisation

A basic counter could be implemented as follows, which will be at risk of race-conditions when two thread call `increment` or `decrement` at the same time.

```
private static class BasicCounter implements Counter {
   private int mCount = 0;

   public void increment() {
      mCount++;
   }
   public void decrement() {
      mCount--;
   }
   public int value() {
      return mCount;
   }
}
```

Java Concurrency - Synchronisation

By adding the `synchronized` keyword to these two functions, only one thread may be executing either `increment` or `decrement` at the same time.

```java
private static class SynchronizedCounter implements Counter {
   private int mCount = 0;

   public synchronized void increment() {
      mCount++;
   }
   public synchronized void decrement() {
      mCount--;
   }
   public int value() {
      return mCount;
   }
}
```

Java Concurrency - Synchronisation

The `synchronized` keyword can also be used to protect a block of code rather than an entire method.

In this case you must provide an object that is locked by the threads.

```
private static class SynchronizedCounter implements Counter {
   private int mCount = 0;
   private Object mLock = new Object();

   public void increment() {
      synchronized(mLock) {
         mCount++;
      }
   }
   public void decrement() {
      synchronized(mLock) {
         mCount--;
      }
   }
   public int value() {
      return mCount;
   }
}
```

Java Concurrency - Synchronisation

It is worth noting that in this example, the object used to synchronize the block of code is a local data member.

The threads are synchronized with respect to the instance of the class because each instance will have its own version of `mLock`.

This gives the same behaviour as synchronizing the entire method.

```
private static class SynchronizedCounter implements Counter {
   private int mCount = 0;
   private Object mLock = new Object();

   public void increment() {
      synchronized(mLock) {
         mCount++;
      }
   }
   ...
}
```

Java Concurrency - Synchronisation

If threads need to synchronise across every instance of a class, then a static member of a class may be used as a lock.

This object will be shared between every instance of the class and so the threads will be synchronized with the same object.

```
private static class SynchronizedCounter implements Counter {
   private int mCount = 0;
   private static object mLock = new Object();

   public void increment() {
      synchronized(mLock) {
         mCount++;
      }
   }
   ...
}
```

Java Concurrency - Synchronisation

In general, synchronizing a block of code is more flexible as it allows only the necessary sections of code to be protected.

It is also possible to have different lock objects to protect different sections of code that could be executed at the same time.

Obviously you must be careful of causing deadlock when using multiple lock objects.

Java Concurrency - Monitors

Synchronization in Java is built on an internal entities called *intrinsic locks*, *monitor locks* or more commonly just *monitors*.

Every object has an *intrinsic lock* associated with it, when a thread needs exclusive access to an object it should *acquire* the object's intrinsic lock and then *release* it again once it has finished.

During this time the thread is said to *own* the intrinsic lock during this time and no other thread can acquire that lock during this time.

Java Concurrency - Monitors

When a thread invokes a `synchronized` method it will acquire the *intrinsic lock* for that object.

For a `synchronized` block of statements, it will acquire the *intrinsic lock* for the object provided to the `synchronized` statement.

A `static synchronized` method will use the lock from the `Class` object associated with the class.

Java Concurrency - Monitors

A thread cannot acquire a lock currently held by another thread.

However, a thread can acquire a lock that it already holds - *reentrant synchronisation*.

This can be helpful when a synchronized method calls another synchronized method of the same object.

Java Concurrency - wait/notify

Each `Object` in Java also has the functions `wait()`, `notify()` and `notifyAll()`.

These methods can be used to cause a Thread to pause execution using `wait()` until it is woken up by another Thread calling `notify()` or `notifyAll()`.

Java Concurrency - wait/notify

Main thread - creates a thread and waits for a while and notifies `mLock`.

```
public class Example06WaitNotify implements Runnable {
   private static Object mLock = new Object();
   private static boolean mCondition = false;

   public static void main(String args[]) throws InterruptedException {
      System.out.println("Main Thread: creating Thread");
      Thread t = new Thread(new Example06WaitNotify());
      t.start();

      System.out.println("Main Thread: sleeping");

      Thread.sleep(2000);

      synchronized (mLock) {
         mCondition = true;
         System.out.println("Main Thread: notifying Thread");
         mLock.notifyAll();
      }
      t.join();
      System.out.println("Main Thread: completed");
   }
   ...
```

Java Concurrency - wait/notify

Child thread - will block and wait for the main thread to wake it up.

```
public class Example06WaitNotify implements Runnable {
   private static Object mLock = new Object();
   private static boolean mCondition = false;

   ...
   public void run() {
      synchronized (mLock) {
         while(mCondition == false) {
            System.out.println("Thread: waiting");
            try {
               mLock.wait();
            } catch(InterruptedException exception){}
         }
      }
      System.out.println("Thread: completed");
   }
}
```

Java Concurrency - wait/notify

For the `wait()` and `notify()` methods to work correctly, the threads calling them should hold the lock on the object being waited on / notified using `synchronized`.

When `wait` is called, the lock will be released until the thread is awoken by another thread calling `notify`.

The thread waking up waiting thread(s) should hold the lock on the object before calling `notify`.

Java Concurrency - wait/notify

Be warned that calling `wait()` will only release the lock on the object being waited on.

If the thread is inside a different `synchronized` block, it will not also release the lock on this object.

This will result in a waiting thread also holding a lock on another object, if this lock is needed by the notifying thread then you may end up in deadlock.

Java Concurrency - wait/notify

Consider the following example:

```
private class QueueWait {
   private Queue<Integer> queue = new LinkedList<Integer>();
   public synchronized Integer get() {
      Integer result = queue.poll();

      while(result == null) {
         synchronized(queue) {
            try {
               queue.wait();
            } catch(InterruptedException exception) {}
            result = queue.poll();
         }
      }
      return result;
   }

   public synchronized void put(Integer i) {
      synchronized(queue) {
         queue.add(i);
         queue.notify();
      }
   }
}
```

Summary

- Java threads - `Runnable`
- Synchronisation - `synchronized`
- `wait/notify`