An Introduction to Programming with Java Threads

Andrew Whitaker
University of Washington

9/13/2006

This document provides a brief introduction to programming with threads in Java. I presume familiarity with the concepts described by Andrew Birrell in: *An Introduction to Programming with Threads*. Readers are advised to first consult Birrell's excellent introduction to threads.

Thread Creation

There are two ways to create a thread in Java. The first technique is to sub-class the `Thread` class. The `Thread.start` method launches the new thread. The thread will execute the user-provided `run` method.

```
public class MyThread extends Thread {

   public void run () {
      // body of thread goes here
   }

   public static void main (String [] args) {
      Thread t = new MyThread();
      t.start(); // launch the new thread
   }
}
```

The second technique is to implement the `Runnable` interface. The `Runnable` interface is more flexible because it allows the body of a thread to reside in any class (not simply those that sub-class `Thread`).

An instance of Runnable can be used to instantiate a new Thread object, as shown below:

```
public class MyRunnable implements Runnable {
   public void run () {
      // body of thread goes here
   }
}

public class SomeOtherClass {
   public void someMethod() {
      Runnable r = new MyRunnable();
      Thread t = new Thread(r);
      t.start();
   }
}
```

A common mistake is to invoke `Thread.run` instead of `Thread.start`. This **does not** create a new thread; it simply invokes the run method.

```
Thread t = new MyThread();

// Don't do this!!!  Always call start() instead
t.run();
```

Sometimes, you will see threads defined with *inner classes* (a class within a class). This is syntactic sugar, but it can make your code smaller and easier to read.

```
Thread t = new Thread (new Runnable () {
   public void run () {
      // do something in the thread body
   }
});
```

Regardless of how it is defined, a Java thread terminates when its `run` method exits or returns. There is no safe way for one thread to force another thread to terminate.

Mutual Exclusion

Java does not use a special Mutex type. Instead, every Java object can serve as a lock (I use the term "lock" and "mutex" interchangeably). The lock associated with an object is sometimes called an "intrinsic lock".

The `synchronized` keyword provides access to Java's mutual exclusion facility. `synchronized` can be used in two ways: synchronized methods and free-standing synchronized blocks.

```
public class MyClass {

   private Object foo = new Object();

   public synchronized void myMethod () {
      // The intrinsic lock for this object is held throughout this method
   }

   public void anotherMethod () {

      synchronized (this) {
         // The intrinsic lock for this object is held during
         // this block
      }

      synchronized (foo) {
         // The intrinsic lock for the "foo" object is held
         // throughout this block
      }
   }
}
```

In addition to a per-object lock, Java also provides a per-class lock. This provides mutual exclusion for static methods (*static* methods do not require an object instance of the class; therefore, the per-object lock isn't very useful).

Java locks are *re-entrant*. Thus, the following code is valid:

```
public class MyClass {

   public synchronized void foo () {
      // do something
   }

   public synchronized void bar () {
      bar(); // this call is OK -- it won't deadlock
   }
}
```

Condition Variables

Java does not use a special condition variable type. Instead, every object can act as a condition variable. `Java.Object` contains the methods `wait`, `notify`, and `notifyAll`, which are analogous to Modula-2's `wait`, `signal`, and `broadcast` methods (everybody's gotta be different...).

In Java, all condition variable operations require holding the object's intrinsic lock (by using the `synchronized` keyword). Invoking the `wait` operation has the effect of releasing the condition variable's associated lock.

Java's condition variables are subject to *spurious wakeups*. That is, the condition is not guaranteed to be true when `wait` returns. In practice, this is dealt with by always using a `while` loop instead of an `if` for the condition test (see example below).

Java's `wait` operation throws an `InterruptedException`. This allows one thread to interrupt the flow of execution of another thread. Annoyingly, all `wait` invocations must catch this exception, even for programs that do no make use of the interruption facility.

```java
// A very simple condition variable example: one thread produces a string;
// the other thread consumes the string

import java.util.List;
import java.util.LinkedList;

class Consumer extends Thread {
   private List list;
   private Object conditionVariable;

   public Consumer (List list,Object condVar) {
      this.list = list;
      this.conditionVariable = condVar;
   }

   public void run () {
      // We must grab the lock to perform a condition variable operation
      synchronized (conditionVariable) {

         // This must be a while statement to catch spurious wakeups
         while (list.isEmpty()) {
            try {
               conditionVariable.wait();
            }
            catch (InterruptedException ie) {
               System.err.println("Unexpected exception: "+ ie);
            }
         }

         System.out.println("Consuming object: " + list.remove(0));
      }
   }
}

class Producer extends Thread {
   private List list;
   private Object conditionVariable;

   public Producer (List list, Object condVar) {
      this.list = list;
      this.conditionVariable = condVar;
   }

   public void run () {
      // we must hold the lock to perform condition variable operations
      synchronized (conditionVariable) {

         list.add("Hello world");

         // wake somebody up...
         conditionVariable.notify();
      }
   }
}
```

```
public class ProducerConsumer {
   public static void main (String [] args) {
      Object condVar = new Object();
      List list = new LinkedList();

      Thread consumer = new Consumer(list, condVar);
      consumer.start();

      Thread producer = new Producer(list, condVar);
      producer.start();
   }
}
```

Thread Interruption

Java contains a thread “interruption” facility, which is similar to the Modula-2 alert facility. See Sun’s documentation (Google java.lang.Thread) for details.

Atomic Primitives

Java 1.5 contains a set of classes for performing atomic operations on primitive types (`AtomicInteger`, `AtomicBoolean`, `AtomicLong`, etc.). The operations on these classes are thread-safe -- you can use them instead of locks for operations on primitive types.

```
import java.util.concurrent.atomic.AtomicInteger;

public class Foo {
   // use this instead of int x = 0;
   private final AtomicInteger x = new AtomicInteger(0);

   // thread-safe version of ++x; no locks required
   public int incrementX () {
      return x.incrementAndGet();
   }
}
```