Concurrent Programming (Part II)
Lecture 10: Terminating Threads and Variable *Visibility*

Dr Kevin Bryson

K.Bryson@cs.ucl.ac.uk

Room 8.04

Course Web Site on Moodle

http://moodle.ucl.ac.uk/course/view.php?id=753

Enrolment Key: ATOMIC

Overview of Lecture

- We are at a transition point within the course:
  - Everything we have talked about so far has assumed a ‘shared memory model’ for concurrency (i.e. Threads within a single JVM that can see the same variables).
  - The next lectures will examine how to handle concurrency within distributed systems (i.e. across different JVMs).
- Previous lectures have looked at creating Threads and controlling them once they are created:
  - Using low-level synchronization mechanisms
  - Using higher-level Java 5 concurrency objects (the last lecture)
- In this lecture I will now show you how to safely close down the Threads once you have finished with them.
- I will also use this to explain a rather subtle aspect of shared-memory concurrency that we have only briefly touched upon, that of ***visibility of shared variables.***

You don’t want any rogue threads left behind … you need to ensure they are all terminated correctly !

Recall terminating a Thread using a ‘done’ variable

- No need to synchronize since setting and testing Booleans is atomic

```
public class MyThread extends Thread {
   private boolean done = false;

   public void run() {
      while (!done) {
         // Do lots of work … Do lots of work …
      }
   }
   public void setDone(boolean b) {
      done = b;
   }
}
```

DO NOT DO THIS !

WHAT IS WRONG WITH THIS CODE ?

The hierarchical memory model …

- Even within a shared memory system – a variable written by one thread may not be immediately ***visible*** from other threads. Why?
- Single processor systems:
  - The variable *may* be stored in a **processor register** for efficiency reasons within the region of code currently being executed by the "variable writing thread" … rather than being modified in main memory.
- Multiprocessor systems:
  - The value *may* be modified in the cache of the "writing processor" but not actually flushed to main memory.
  - Even if its value is flushed to main memory, it's possible that the "reading processor" *may* have a **stale value** in its cache which has not been updated.
- In all these different cases … other threads will not "see the variable changing" (i.e. variable visibility problems)

Volatile & synchronization solve these visibility problems …

- The ***volatile*** keyword ensures that all threads will read and write this variable directly from main memory – it guarantees ***visibility*** of this variable between threads (however, clearly it does not synchronize them!)
- ***Sychronization*** also guarantees this variable ***visibility*** since the Java specification states that all threads must also "synchronize their working memories" with the main memory at the entry and exit to synchronized segments.
- **Visibility** is the other reason (in addition to avoiding interference) why all sections of code using shared variables should be synchronized.

What could potentially happen in the previous example code ?

Terminating a Thread using a ‘done’ variable

- No need to synchronize since setting and testing Booleans is atomic.
- BUT WE ALSO NEED TO MAKE THE VARIABLE VOLATILE TO ENSURE VISIBILITY BETWEEN ALL THREADS.

```
public class MyThread extends Thread {
   private volatile boolean done = false;

   public void run() {
      while (!done) {
         // Do lots of work … Do lots of work …
      }
   }
   public void setDone(boolean b) {
      done = b;
   }
}
```

See Terminator1 Listing – does it work ?

What limitations exist with this method ?

Using Interrupts to Terminate Threads

- A Thread can call the ‘interrupt()’ method on another Thread object to terminate the other Thread.
- See Terminator2 Listing.
- Does it work ?
- See Terminator3 Listing.
- Does it work ?
- Can you explain this difference in behaviour ?

Methods that throw InterruptedException

- Some methods throw InterruptedException such as sleep() and wait().
- If a Thread is blocking within these methods, and interrupt() is called on the Thread – then an InterruptedException will be thrown to terminate the Thread.
- If the Thread is not blocked … then the interrupt() method will just set an interrupt flag within the Thread object.
- The value of this flag can be tested with isInterrupted().
- See Terminator4 Listing – does it work ?

Summary

- We have examined concerns with the **visibility** of variables within multithreaded programs.
- We have also examined and demonstrated how to terminate Threads using the interrupt() method.
- This was the last lecture on ‘shared memory concurrency’.
- In the next lecture we will start to look at concurrency mechanisms when you do not have a shared memory – such as distributed systems.