Lab 4: Concurrency Tutorial

In this lab, you will learn how to write multithreaded programs in Java. For the purposes of this course, threads will be implemented by implementing the abstract “Runnable” class. While multithreading has the advantage of speeding up the program, it may also interfere with program correctness if done improperly. In particular, it is important to perform synchronization when threads share global variables in the program.

How to Write a Runnable Object

1. Open Eclipse and create a new project called “concurrent_prog”
2. In the package explorer, create a package under the “src” folder and call it “com”. Create a folder under “com” and call it “concurrent”.
3. Under the “concurrent” folder, create a new file and name it “ConcurrentProg.java”
4. Copy the following skeleton code and paste it into ConcurrentProg.java. Save the file.

```
package com.concurrent;

public class ConcurrentProg {
	private static int tagCounter = 0;

	public static void main(String[] args) {

	}

	/*
	 * This function increments tagCounter by 100000 and returns the
	 * new tag
	 */
	private static int incBy100000() {
		for (int i = 0; i < 100000; i++) {
			tagCounter++;
		}

		return tagCounter;
	}
}
```

5. Our goal is as follows: In the main function, we want to create 4 threads. Each thread writes the following phrase to standard output: “This is thread <x>”, where <x> refers to the unique tag assigned to the thread. A thread can get its tag after a call to the incBy100000() method, which updates the shared tagCounter variable; thus, the four tags will have to be 100000, 200000, 300000, and 400000.

6. To achieve our goal, we must first implement the abstract Runnable class and override its run() method (you might want to check out online Java references for more on abstract Java classes). *Inside* the ConcurrentProg class, create a private class that implements Runnable and call it PrintMsg. Your code should now look as follows:

```
package com.concurrent;

public class ConcurrentProg {
	private static int tagCounter = 0;

	public static void main(String[] args) {

	}

	/*
	 * This function increments tagCounter by 100000 and returns the
	 * new tag
	 */
	private static int incBy100000() {
		for (int i = 0; i < 100000; i++) {
			tagCounter++;
		}

		return tagCounter;
	}

	private class PrintMsg implements Runnable {

	}
}
```

7. Inside the PrintMsg class, create a method that overrides the run() method of Runnable. PrintMsg should now look as follows:

```
	private class PrintMsg implements Runnable {
		@Override
		public void run() {

		}
	}
```

8. Now, we want to actually define what the run() method should do. First, we need to assign a tag for the current thread. To do so, invoke the incBy100000() method and store its value in a variable:

```
@Override
public void run() {
      int tag = incBy100000();
}
```

9. Next, we want to print the message to standard output. We can call Sytem.out.println() to accomplish this:

```
@Override
public void run() {
      int tag = incBy100000();
      System.out.println("This is thread " + tag);
}
```

10. Now that we have implemented the run() method, we can start creating new threads. This should be done in the main() function of the ConcurrentProg class by calling the start() method of Runnable. The start() method is called by first creating a new thread, as shown below:

```
public static void main(String[] args) {
      ConcurrentProg cp = new ConcurrentProg();
      for (int i = 0; i < 4; i++) {
            new Thread(cp.new PrintMsg()).start();
      }
}
```

11. The complete code is shown below:

```
package com.concurrent;

public class ConcurrentProg {
      private static int tagCounter = 0;

      public static void main(String[] args) {
            ConcurrentProg cp = new ConcurrentProg();
            for (int i = 0; i < 4; i++) {
                  new Thread(cp.new PrintMsg()).start();
            }
      }
```

```
     /*
      * This function increments tagCounter by 1000 and returns the
      * new tag
      */
     private static int incBy100000() {
          for (int i = 0; i < 100000; i++) {
               tagCounter++;
          }

          return tagCounter;
     }

     private class PrintMsg implements Runnable {
          @Override
          public void run() {
               int tag = incBy100000();
               System.out.println("This is thread " + tag);
          }
     }
}
```

12. Build the project and try running the code several times. You'll very likely notice that in most runs, the tags will be incorrect (i.e., they will not be 100000, 200000, 300000, and 400000). This is because we have not implemented synchronization, which the next section will address.

How to Synchronize When Accessing or Updating a Shared Variable

1. Notice that the four threads we have created in the above example all access one shared variable: tagCounter. Since these threads execute concurrently, we need to have some way to prevent other threads from updating or accessing tagCounter while one thread is using or modifying that variable. This is where the *synchronized* keyword comes in. To do so, we first need to create a "lock" object for tagCounter, which is basically just an object of type Object(). We will call this tagCounter_lock, and declare it as a member of the ConcurrentProg class as follows:

```
private static Object tagCounter_lock = new Object();
```

2. Next, we need to identify any piece of code in PrintMsg that accesses or updates tagCounter. In this case, there is only one line of code that updates tagCounter: The call to incBy100000() (this line is an update to tagCounter because incBy100000() has the side effect of updating tag counter). Thus, we need to synchronize this line of code to ensure that only one thread can execute that line at any given time. The synchronization is done by modifying the run() method of PrintMsg as follows:

```
@Override
public void run() {
      int tag;
      synchronized (tagCounter_lock) {
            tag = incBy100000();
      }
      System.out.println("This is thread " + tag);
}
```

This means that whenever a thread is in a synchronized(tagCounter_lock) block, other threads will not be able to execute what's inside the block until that thread has completed its execution of the block.

3. Build and run the project again. This time, you should notice that the tags are correct.
4. Finally, note that in this example, only one line had to be synchronized, and only one variable was shared. In many concurrent programs, there can be more than one shared variable, and, usually, more than one line would have to be synchronized.