Threading

In the last lecture we saw an example where it would be convenient to have parts of our program run separately. For example, if we want a ball to continue bouncing while the main method waits for input using the Scanner class, then we have a problem because the program blocks and waits for the Scanner to complete. You have probably seen windows stop responding while they are busy with heavy computation. The solution to this problem is threading. We can run the GUI in one thread so it is responsive while putting the heavy computation in another thread.

A thread is a separate computation process. In Java, you can have programs with multiple threads. You can think of the threads as computations that execute in parallel. On a computer with enough processors, the threads might indeed execute in parallel. In many normal computing situations, the threads do not really execute in parallel. Instead, the computer switches resources between threads so that each thread in turn does a little bit of computing. To the user this looks like the processes are executing in parallel. Additionally, you can't predict when the computer will switch from one thread to another so you should just consider them as executing in parallel.

The Thread class

In Java, a thread is an object of the class **Thread**. The normal way to program a thread is to define a class that is a derived class of the class Thread. An object of this derived class will be a thread that follows the programming given in the definition of the derived (thread) class.

Where do you do the programming of a thread? The class Thread has a method named **run**. The definition of the method run is the code for the thread. When the thread is executed, the method run is executed. Of course, the method defined in the class Thread and inherited by any derived class of Thread does not do what you want your thread to do. So, when you define a derived class of Thread, you override the definition of the method run to do what you want the thread to do.

Here is a simple example that makes a thread that outputs "hello" while the main thread outputs "world".

```
public class Foo extends Thread
{
      @Override          // Inherited from thread but overridden here
      public void run()
      {
            System.out.println("hello");
      }
}
public class Test
{
      public static void main(String[] args)
      {
            Foo fooThread = new Foo();
            fooThread.start(); // NOT fooThread.run()
            System.out.println("world");
      }
}
```

The method start initiates the computation (process) of the calling thread. It performs some overhead associated with starting a thread and then it invokes the run method for the thread

Here is another version that really shows this is running in two separate threads:

```
import java.util.Scanner;
class Foo extends Thread
{
      // Inherited from thread but overridden here
      @Override
      public void run()
      {
            while (true)
            {
                  System.out.println("hello");
                  try
                  {
                        Thread.sleep(1000);
                  }
                  catch (InterruptedException e)
                  {
                        System.out.println(e);
                  }
            }
      }
}

public class Test
{
      public static void main(String[] args)
      {
            Foo fooThread = new Foo();
            fooThread.start(); // NOT fooThread.run()
            Scanner kbd = new Scanner(System.in);
            while (true)
            {
                  String s = kbd.next();
                  System.out.println(s);
            }
      }
}
```

Class exercise: Modify the bouncing ball program to update the ball motion in a separate thread.

The Runnable Interface

There are times when you would rather not make a thread class a derived class of the class Thread. The alternative to making your class a derived class of the class Thread is to have your class instead implement the Runnable interface. The Runnable interface has only one method heading:

    public void run()

A class that implements the Runnable interface must still be run from an instance of the class Thread. This is usually done by passing the Runnable object as an argument to the thread constructor. The following is an outline of one way to do this:

```
public class ClassToRun extends SomeClass implements Runnable
{
     ....
    public void run()
    {
        //Fill this just as you would if ClassToRun
        //were derived from Thread.
     }
     ....
    public void startThread()
    {
        Thread theThread = new Thread(this);
        theThread.start();
    }
     ....
 }
```

The above method startThread is not compulsory, but it is one way to produce a thread that will in turn run the run method of an object of the class ClassToRun. Here is an example of our repeat Hello program using the Runnable interface.

```
import java.util.Scanner;
public class Test implements Runnable
{
      @Override
      public void run()
      {
            while (true)
            {
                  System.out.println("hello");
                  try
                  {
                        Thread.sleep(1000);
                  }
                  catch (InterruptedException e)
                  {
                        System.out.println(e);
                  }
```

```
                }
        }

        public static void main(String[] args)
        {
                Test t = new Test();
                Thread mainThread = new Thread(t);
                mainThread.start();
                Scanner kbd = new Scanner(System.in);
                while (true)
                {
                        String s = kbd.next();
                        System.out.println(s);
                }
        }
}
```

One of the main advantages to this technique is it is easy to make an existing class into a thread, since you don't need to create a separate thread class, but can instead modify an existing class and tack on a run() method with the code you want to run in the thread.

Race Conditions and Thread Synchronization

When multiple threads change a shared variable it is sometimes possible that the variable will end up with the wrong (and often unpredictable) value. This is called a race condition because the final value depends on the sequence in which the threads access the shared value. For example, consider two threads where each thread runs the following code:

```
int local;
local = sharedVariable;
local++;
sharedVariable = local;
```

The intent is for each thread to increment sharedVariable by one so if there are two threads then sharedVariable should be incremented by 2. However, consider the case where sharedVariable is 0. The first thread runs and executes the first two statements, so its variable local is set to 0. Now there is a context switch to the second thread. The second thread executes all four statements, so its variable local is set to 0, incremented, and sharedVariable is set to 1. Now we return to the first thread and it continues where it left off which is the third statement. The variable local is 0 so it is incremented to 1 and then the value 1 is copied into sharedVariable. The end result after both threads are done is that sharedVariable has the value 1 and we lost the value written by thread two!

You might think that this problem could be avoided by replacing our code with a single statement like:

```
sharedVariable++;
```

Unfortunately this won't solve our problem because the statement is not guaranteed to be an “atomic” action and there could still be a context switch to another thread “in the middle” of executing the statement.

To demonstrate this problem first consider the Counter class shown below. This simple class merely stores a variable that increments a counter. It uses the somewhat roundabout way to increment the counter on purpose to increase the likelihood of a race condition.

```
public class Counter
{
      private int counter;
      public Counter()
      {
            counter =0 ;
      }
      public int value()
      {
            return counter;
      }
      public void increment()
      {
            int local;
            local = counter;
            local++;
            counter = local;
      }
}
```

The way we will demonstrate the race condition is to:

1. Create a single instance of the Counter class.
2. Create an array of many threads (30,000 in the example) where each thread references the single instance of the Counter class.
3. Each thread runs and invokes the increment() method.
4. Wait for each thread to finish and then output the value of the counter. If there were no race conditions then its value should be 30,000. If there were race conditions then the value will be less than 30,000.

We create many threads to increase the likelihood that the race condition occurs. With only a few threads it is not likely that there will be a switch to another thread inside the increment() method at the right point to cause a problem.

The only new tool that we need for our demonstration program is a way to wait for all the threads to finish. If we don't wait then our program might output the counter before all the threads have had a chance to increment the value. We can wait by invoking the **join()** method for every thread we create. This method waits for the thread to complete. The join() method throws InterruptedException which is a checked exception so we must use the try/catch mechanism.

The class RaceConditionTest below illustrates the race condition. You may have to run the program several times before you get a value less than 30,000. Problems as a result of race conditions are often rare occurrences. This makes them extremely hard to find and debug!

```
public class RaceConditionTest extends Thread
{
    private Counter countObject;

    public RaceConditionTest(Counter ctr)
    {
            countObject = ctr;
      }

    public void run()
    {
       countObject.increment();
    }

    public static void main(String[] args)
    {
            int i;
            Counter masterCounter = new Counter();
            RaceConditionTest[] threads = new RaceConditionTest[30000];

            System.out.println("The counter is " +
                  masterCounter.value());
            for (i = 0; i < threads.length; i++)
            {
                  threads[i] = new RaceConditionTest(masterCounter);
                  threads[i].start();
            }

            // Wait for the threads to finish
            for (i = 0; i < threads.length; i++)
            {
                  try
                  {
                        threads[i].join();
                  }
                  catch (InterruptedException e)
                  {
                        System.out.println(e.getMessage());
                  }
            }
            System.out.println("The counter is " +
                  masterCounter.value());
      }
 }
```

So how do we fix this problem? The solution is to make each thread wait so only one thread can run the code in increment() at a time. This section of code is called a **critical region**. Java allows you to add the keyword **synchronized** around a critical region to enforce the requirement that only one thread is allowed to execute in this region at a time. All other threads will wait until the thread inside the region is finished.

In this particular case we can add the keyword synchronized to either the method or around the specific code. If we add synchronized to the increment() method in the Counter class then it looks like this:

```
public synchronized void increment()
{
      int local;
      local = counter;
      local++;
      counter = local;
}
```

If we add synchronized inside the code then we can write:

```
public void increment()
{
      int local;
      synchronized (this)
      {
            local = counter;
            local++;
            counter = local;
      }
}
```

Either version will result in a counter whose final value is always 30,000. There are many other issues involved in thread management, concurrency, and synchronization. These concepts are often covered in more detail in an operating systems course.