**ISTE-121 Day13/14 – Synchronization and Thread Control II**

**Why do we need synchronization?**

If two or more threads share a resource, they must synchronize their access to that resource, or unexpected things may happen. For example, run the ThreadingExample.jar file from today’s downloads.

Each time you click the “Add new thread” button, another thread is started. They **all** write their output to the TextArea. As you add more and more threads, eventually the messages start to get intermingled. This is because they are not synchronizing their access to the text area, which is a shared resource.

Add the command line argument “sync” and run it again (click Build->Run Arguments to get the command line window at the top and type “sync” in it). This time, access to the text area is synchronized, so you see no overlap.

For another example, consider the PrintDemo example (see today’s downloads).

**How? – Synchronized Blocks – PrintDemo**  
Open TestThread1.java in JGrasp.

In this file, you see three classes: PrintDemo, ThreadDemo, and TestThread1. TestThread1 is the main class. It instantiates a PrintDemo object (PD) and passes it to the constructor of ThreadDemo. Two ThreadDemo objects are created. ThreadDemo extends Thread, so T1 and T2 are Threads. When started, their run methods execute.

Run this program 2-3 times. The output is different each time, because the threads overlap at will, and there is no way to predict which will run when, or when a thread will be moved from the Running state to the Runnable state because its time slice expires.

Let’s note a few things about this program.

* There is exactly one PrintDemo object in the entire program, though it is passed to several methods/classes (that is, you only see **new PrintDemo()** once).
* The constructor for ThreadDemo calls setName for **this**, which sets the name of the underlying Thread object.
* PrintDemo calls Thread.currentThread().getName() to get the thread’s name. PrintDemo is **not** a Thread, so it has **not** inherited the getName() method. Instead, it calls the static method currentThread() from the Thread class to find which Thread it belongs to at that point in time. It then calls **that** thread’s getName() method.
* In ThreadDemo, we call this.getName(), because ThreadDemo **is-a** Thread and can call Thread methods directly (no need for Thread.currentThread()).

Now, save a copy of this called TestThread2 (rename class TestThread1 to TestThread2 and Save As). In the run method of ThreadDemo, change the lines:

PD.printCount();  
 System.out.println("Thread " + threadName + " exiting.");  
To

synchronized(PD) {  
 PD.printCount();  
 System.out.println("Thread " + threadName + " exiting.");  
 }  
and run this version 2-3 times. You should get the same output every time. This is because, the **synchronized(PD)** line sets a lock in the PrintDemo. We did not declare any kind of lock in the PrintDemo class … the lock is inherited from the super class Object … **every Object has such a lock**.

Whichever thread gets to the **synchronized** statement first, sets the lock. The lock remains set until the synchronized block is exited, when it is automatically unset.

If a thread gets to the synchronized statement when the lock is set, it is **blocked** until the lock is unset.

**Lock Objects**  
Now, save a copy of TestThread2, as TestThread4.java. At the very top of the file, add the line:

import java.util.concurrent.locks.\*;

Then, in the main method of TestThread4, after the line:

PrintDemo PD = new PrintDemo();

Add the line:

Lock myLock = new ReentrantLock();

And pass myLock as a **third parameter** to the ThreadDemo constructor (both times). In ThreadDemo, change the constructor to expect this third parameter, and **save it as a new attribute** (named myLock).

Finally, in the run method, change:

synchronized(PD) {  
 PD.printCount();  
 System.out.println("Thread " + this.getName() + " exiting.");  
 }

To

myLock.lock();  
 PD.printCount();  
 System.out.println("Thread " + this.getName() + " exiting.");  
 myLock.unlock();

Run this 2-3 times and you should see the same output as from TestThread2.

So why bother? Now the lock is **explicit** (instead of being part of some object) and can be passed as a parameter. Occassionally this is just what you need. Beware, however. If you set the lock and an exception occurs, it may **never be unset**, causing all threads to be blocked. To avoid, set the lock as follows:

try {  
 myLock.lock();  
 PD.printCount();  
 System.out.println("Thread " + this.getName() + " exiting.");  
 }  
 catch(Exception e) {  
 // do what you need to do to handle the exception  
 }  
 finally {  
 myLock.unlock();  
 }  
This will ensure the lock is unlocked, even in the case of an exception.

**Thread Scheduling**

We know that when a thread loses its turn on the processor because its time slice runs out, it returns to the Runnable state. In a strict time-slicing scheme, each thread gets the same size time slice and only loses the processor when that time slice has expired, or the thread does something to cause it to be blocked.

Another approach is to use priorities when threads are inserted into the Runnable state. In this case, the system maintains a queue of Runnable threads and when a thread enters the Runnable state, it does not necessarily go to the end of the queue. Instead, the queue is sorted by priority, so higher priority threads are inserted nearer the head of the queue and lower priority threads are inserted nearer the end of the queue.

There are priorities in Java, which vary from 1 to 10. The default priority for threads is 5. A thread can see its priority by the getPriority() method (in a Runnable, which does NOT inherit from the Thread class, you need to call Thread.currentThread().getPriority()). A thread can also change its priority with the setPriority() mutator.

**Deadlock**

This is when threads are all waiting for a resource that some other thread has locked. For this to happen, there must be a circular dependency among the threads.

**Animation**

Sometimes we want the GUI to change as we watch … this is essentially animation. To do this, we will use the following structure:

* Main program: extends Application (possibly implements EventHandler<ActionEvent>)
  + Declares global variables, including GUI components
    - This may include several objects of a class that extends some  
      GUI component and also implements Runnable
  + Instantiates threads and starts them
  + Waits for them to finish
* Threads (or Runnables) usually extend some GUI component to be animated
  + Declare non-shared, per-thread variables
  + run() – thread main
    - does calculations for changes of animated items
    - changes the animated items

See this in the Animate1 folder in today’s downloads. There is a Start/Stop button in the CENTER that will start or stop the animation.

**In Class Exercise**

Modify the code so that when the program begins, all four buttons are red. When the Start button is pressed, the buttons should cycle through all four of their colors a total of three times each. In the third color cycle, the buttons should stop changing color when they reach the color yellow.

When a button reaches the color yellow on the third cycle, it will print to the console, “Button N finished!”, where N is the button number. The buttons should be numbered in the same order in which they were added to the GUI so that Grumpy is Button 0, Dopey is Button 1, Bashful is Button 2, and Sleepy is Button 3.

After a button has printed to the console, it should exit its run method. If the user would like to see the buttons cycle through their colors again, they just need to press the Stop button and then the Start button again.