Lab 5: Threads

In this lab, you will work with threads.

Objectives:

- To be able to create multiple threads
- To synchronize threads

Part 1: Subclassing the Thread Class

In this part, you will simulate the operation of a stock-retrieval application. You will write a class that extends *Thread* and then create a couple instances and start the threads.

Steps:

_1. Create a new Java project named **Lab05**.

_2. Create a class named **StockTicker** in the *part1* package. The new class should have a *main* method.

_3. Within the new class, write a *private* static member inner class named **StockTickerThread** that extends *java.lang.Thread*. Complete the inner class as follows:

   a. Define a String field named *symbol* that will hold the stock symbol.

   b. Write a one-argument constructor that accepts a stock-symbol String and assigns it to the field.

   c. Override the *run* method. You will complete it in the next step.

_4. Complete the overridden *run* method:

   a. Code an infinite loop.

   b. Within the loop, use the *java.util.Random* class to obtain a random *int* that's in the range of 0 to 5000 (use the *nextInt* method).

      Use the random integer as the number of milliseconds to sleep to simulate contacting a remote stock-ticker service.

   c. Retrieve another random integer, this one between 0 and 100 -- this will simulate the stock value.

      Print the symbol and value to the console.

_5. Complete the *main* method as follows:

   a. Create two instances of the StockTickerThread, using "IBM" and "MSFT" (or whatever you want) for the symbols.

   b. Start a thread on each instance.

_6. Run the program:

   You should see simulated stock quotes from the two symbols.

   To stop the program, press the red square icon in the Console's titlebar.

_7. Now let's make terminating the program more clean. Follow these steps:

   a. Edit the StockTickerThread class and define a boolean field named **cancel** and initialize it to *false*.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. Write a method named **stopTicker** that sets the *cancel* field to *true*.

c. Modify the "forever" loop so that it exits if *cancel* is *true*.

d. Edit *main*, and after starting the threads, print a message telling the user they can press Enter to terminate.

   Then use *System.in.read()* to wait for the user to press Enter, at which point your program should call *stopTicker* on both thread objects.

e. Run and test. Can you stop the threads by pressing Enter into the Console?

Part 2: Implementing the Runnable Interface

In the last part, you created threads by subclassing the java.lang.Thread class. However, you cannot use that technique if the class that comprises the thread already has a superclass -- in Java, a class can only have one superclass. If that's the case, then the class can implement the *java.lang.Runnable* interface instead.

In this part, you will create a Java *applet* that use a so-called background thread. Since applets must extend either *javax.swing.JApplet* or *java.awt.applet.Applet*, your applet will need to implement *Runnable* so it can support the thread.

The background thread will perform an animation -- it will move a text string around the applet's rectangle, "bouncing" the string off of the rectangle's edges.

Steps:

_1. Create a class named **AnimatorApplet** in the *part2* package that extends *javax.swing.JApplet* and implements *java.lang.Runnable*. This class should not have a *main* method, since applets are invoked automatically by the container in which they run (typically, a Web browser or an applet-viewer).

_2. Override the following methods from the Applet class. You don't need to add any code yet.

- **public void start()** -- The applet container calls this method when it's OK for the applet to perform continuous processing, such as an animation
- **public void stop()** -- The applet container calls this method when the continuous processing should stop (for example, if the user clicks on a hyperlink to leave the page containing the applet)

_3. Implement the *run* method from the Runnable interface. You will complete it in a later step.

_4. First, let's work on controlling the background animation thread. Follow these steps:

a. Define a field in the class that will contain the thread's reference:

```
private Thread me;
```

b. In the *start* method, write code that creates a java.lang.Thread object and stores its reference in the *me* variable, then starts the new thread. Hint: You can use the Java *this* keyword to refer to the the object (the applet) that implements Runnable.

c. In the *stop* method, set the *me* field to *null*.

d. In the *run* method, code a *while* loop that runs as long as the current thread is *me*:

```
while (Thread.currentThread() == me)
{
}
```


Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

This may seem convoluted, but it's the conventional way to control a background thread in an applet.

_5. Now let's work on the animation code. Follow these steps:

a. Define the following fields:

```
private int x;      // x position
private int y;      // y position
private int dx = 5; // x speed
private int dy = 5; // y speed

private boolean  firstTime;
```

b. Write a private method that you will call from the thread to actually draw the text. Note that it uses a XOR technique -- XOR drawing lets you erase an object by drawing again in the same place. In other words, if you call this method twice, the first time it will draw the text and second time it will erase it.

```
private void draw(Graphics g)
{
    g.setColor(Color.black);
    g.setXORMode(Color.white);
    g.drawString("Hello", x, y);
}
```

Import from the *java.awt* package.

c. Write a another method that you will call from the thread to calculate a new position for the text -- you will complete this method in the next step.

```
private void move()
{
    // retrieve applet's rectangle width and height
    Dimension d = getSize();
}
```

_6. To complete the *move* method, follow these steps:

a. Calculate a new position (x,y) by adding the *dx* and *dy* fields to *x* and *y* respectively.

b. We want the text to "bounce" off of the applet's edges, so we need to reverse the speed (dx,dy) when we detect that the text has moved beyond an edge. To make this a little more interesting visually, you will randomize the speed on each bounce.

Start by looking for the case when *x* is less than zero:

```
if (x<0)
{
}
```

If x is less than zero, set *dx* to a random number between 1 and 5. You can use the *java.util.Random* class's nextInt() method.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

c. Next, look for the case where *x* is greater than *d.width*. If that's true, then set *dx* to a random number between -1 and -5.

d. Repeat the two tests for *y*, setting the *dy* variable to an appropriate positive or negative random value.

_7. Now you can complete the *run* method, which comprises the thread. Follow these steps:

a. Inside of the loop, *sleep* the thread for 100 milliseconds.

b. Retrieve a *Graphics* object that you can use to draw to the applet's surface:

```
Graphics g = getGraphics();
```

c. Now it's time to erase the text from the last time the thread ran. But if this is the first time through the loop, there's nothing to erase. So check that case, then call your *draw* method:

```
if (firstTime == true)
    firstTime = false;
else
    draw(g);
```

d. In any case, call your *move* method, followed by a call to *draw* to draw the text at its new position.

_8. Add code to the *start* method to set *firstTime* to *true* before starting the thread.

_9. Add code to the *stop* method to set *firstTime* to *false*.

_10. To run your program in Eclipse's applet viewer, right-click on *AnimatorApplet.java* in the Package Explorer, then choose **Run As - Java Applet**.

The applet viewer will instantiate your applet and call its *start* method -- you should see your animated text.

You can use the applet viewer's *Applet - Stop* and *Applet - Start* menu items to control your thread's execution.

Part 3: Thread Synchronization

In this part, you will demonstrate how using multiple threads can lead to a *race condition* and how synchronization can solve such problems.

Steps:

_1. Create a class named **TestSync** in the *part3* package with a *main* method.

_2. Inside the TestSynch class, write a private static inner class named **MyCounter**. Complete the inner class as follows:

a. Define a private field of type *int* named **count**.

b. Write a method named **adjustCount** that accepts no arguments and returns *void*. You will complete the method in the next step.

_3. The *adjustCount* method is where the race condition will occur. Complete it as follows:

a. Set the *count* field to 12. Since *count* is a field, it will be shared among all of the threads that call this method -- that will lead to the race condition.

b. Sleep the current thread for a random interval between 0 and 1000 milliseconds.

c. Increment the *count* field.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

d. Sleep again for a random interval between 0 and 1000 milliseconds.

e. Test the value of *count* -- if it's not 13, print a message to the console with the current thread's *name* and the value of *count* You can retrieve the thread name using an expression like *Thread.currentThread().getName()*.

_4. Create another private static inner class within TestSynch named **MyThread** that extends *java.lang.Thread*. Complete it as follows:

a. Define a field named **obj** of type *MyCounter*.

b. Write a two-argument constructor that accepts a String and a *MyCounter* object. The constructor should call the superclass constructor with the String (thread name) and assign the MyCounter reference to the field.

c. Override the *run* method and write an infinite loop. Inside the infinite loop, call the *adjustCount* method on the MyCounter reference stored in the field.

_5. Complete the *main* method in TestSynch as follows:

a. Create a MyCounter object.

b. Create three MyThread objects, passing Strings of "Thread1", "Thread2" and "Thread3" and the reference to the MyCounter object.

c. Start each of the threads.

_6. Run the program.

If there was only one thread calling *adjustCount*, the value of *count* would always be 13 -- why is that not the case here?

Kill the program by click the red-square icon on the Console toolbar.

_7. Edit the *adjustCount* method definition and add the *synchronized* keyword, then test again. There should be no output. How does the synchronized keyword fix the problem?

Part 4: Joining Threads

One handy use of threads is to start a lengthy process in a separate thread and then have the "main" thread go about its business doing other things. Then when the "main" thread needs the results of the background operation, it can *join* with the thread, which causes the "main" thread to wait if the background thread hasn't yet finished.

In this part of lab, you will read a text file as a "lengthy process" using a separate thread.

Steps:

_1. Create a class named **TestJoin** in the *part4* package with a *main* method.

_2. Inside of the TestJoin class, write a private static inner class named **ReadFileThread** that implements RunnableR. Complete the inner class as follows:

a. Define two private String fields -- one to hold a filename and the other to hold the results of reading a text file.

b. Write a one-argument constructor that accepts the filename and stores it in the field.

c. Write a "get" method that returns the String field containing the text read from the file.

d. Override the *run* method and complete it as described in the next step.

_3. Complete the thread's *run* method as follows:

a. Create a StringBuffer that you will use to accumulate the characters from the file.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

b. Open the text file:

```
FileReader fr = new FileReader(fname);
```

Note that you will need to use a *try-catch* block to handle exceptions.

c. Define a *int* to hold each character read from the file, then loop through, reading a character at a time from the file:

```
int ch = fr.read();
while (ch != -1)
{
    ch = fr.read();
}
```

d. Inside the loop, above the *read*, sleep the thread for 10 milliseconds to make this a lengthy process.

e. Append each character read into the StringBuffer. You will need to cast from *int* to *char*.

f. Outside of the loop, close the FileReader and then convert the StringBuffer object into a String and store its reference in the field you defined earlier.

_4. Complete the *main* method as follows:

a. Create an instance of the ReadFile thread, passing the program's own source file name as the text file: *src/part4/TestJoin.java*.

b. Set the thread's priority to the minimum value.

c. Start the thread.

d. In a real program, the main thread would now do something useful, in parallel with the background thread. In this lab, just print out 50 'a' characters, sleeping 100 milliseconds between each character. Then print a "Waiting" message.

e. Join with the background thread.

f. Retrieve the result String from the thread object and print it to the console.

_5. Run your program. Can you see the parallel activity?

Play around with the sleep times and the count of 'a' characters and see if you can get the thread to complete before main and vice versa.

Reinforcement

In this part, you will practice some of the techniques you learned in this lab.

In this part, you will add a background thread to a GUI application. The background thread will simulate a lengthy printing process.

Steps:

_1. Import the **{Lab Installation Directory}/starters/lab05/reinforcement** folder into the *lab05* project.

Examine and run this class to see what it does (not much).

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

_2. Your job is to add a simulated printing process when the user clicks the *Print* button and to interrupt and stop the print if the user presses the *Cancel* button. Here are some hints:

   a. We recommend that you use a static member inner class for the thread as in previous parts.

   b. You can use anonymous inner classes or lambdas to respond to the button clicks.

   c. You can simulate printing 10 pages by looping and sleeping for 1000 milliseconds for each "page". You should print a message to the console indicating as each page "prints" and a message when the printing is finish.

   d. For extra credit, only start a new print job when the previous is finished. Display a message box if the user asks to print while a previous print is in process.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2016. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.