Lab 09: Multi-Threading with Java

Objective(s):

1. Multi-Threading

1: Multi-Threading

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. However, in most normal computing situations, the threads do not really do this. 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.

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.

```
Display 19.1 contains a very simple action GUI. When the "Start" button is clicked, the GUI draws
circles one after the other until a large portion of the window is filled with circles. There is
1/10 of a second pause between the drawing of each circle. So, you can see the circles appear
one after the other. If you are interested in Java programming, this can be pretty exciting for
the first few circles, but it quickly becomes boring. You are likely to want to end the program
early, but if you click the close-window button, nothing will happen until the program is finished
drawing all its little circles. We can use threads to fix this problem.
```

Thread.sleep

In Display 19.1 , the following method invocation produces a 1/10 of a second pause after drawing each of the circles:

doNothing(PAUSE);

which is equivalent to

doNothing(100);

The method doNothing is a private helping method that does nothing except call the method Thread.sleep and take care of catching any thrown exception. So, the pause is really created by the method invocation.

Thread.sleep(100);

This is a static method in the class Thread that pauses whatever thread includes the invocation. It pauses for the number of milliseconds (thousandths of a second) given as an argument. So, this pauses the computation of the program in Display 19.1 for 100 milliseconds or 1/10 of a second.

"Wait a minute," you may think, "the program in Display 19.1 was not supposed to use threads in any essential way." That is basically true, but every Java program uses threads in some way. If there is only one stream of computation, as in Display 19.1 , then that is treated as a single thread by Java. So, threads are always used by Java, but not in an interesting way until more than one thread is used. You can safely think of the invocation of

Thread.sleep(milliseconds);

as a pause in the computation that lasts (approximately) the number of milliseconds given as the argument. (If this invocation is in a thread of a multithreaded program, then the pause, like anything else in the thread, applies only to the thread in which it occurs.) The method Thread.sleep can sometimes be handy even if you do not do any multithreaded programming. The class Thread is in the package java.lang and so requires no import statement.

The method Thread.sleep may throw an InterruptedException, which is a checked exception and so must be either caught in a catch block or declared in a throws clause.

The classes Thread and InterruptedException are both in the package java.lang,

so neither requires any import statement.

Syntax:

```
Thread.sleep( Number_Of_Milliseconds );
```

Example

```
try
{
Thread.sleep(100); //Pause of 1/10 of a second
}
catch (InterruptedException e)
{
System.out.println("Unexpected interrupt");
}
```

Example: A Non Responsive GUI Base Programming Example using thread for 100 millisecond pause (Display 9.1)

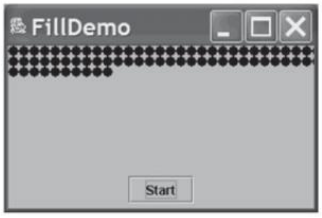
```
import ijavax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.event.ActionListener;
```

```
import java.awt.event.ActionEvent;
/**
Packs a section of the frame window with circles, one at a time.
*/
public class FillDemo extends JFrame implements ActionListener
{
public static final int WIDTH = 300;
public static final int HEIGHT = 200;
public static final int FILL_WIDTH = 300;
public static final int FILL_HEIGHT = 100;
public static final int CIRCLE_SIZE = 10;
public static final int PAUSE = 100; //milliseconds
private JPanel box;
public static void main(String[] args)
{
FillDemo gui = new FillDemo();
gui.setVisible( true );
}

public FillDemo()
{
setSize(WIDTH, HEIGHT);
setTitle("FillDemo");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLayout( new BorderLayout());
box = new JPanel();
add(box, "Center");

JPanel buttonPanel = new JPanel();
buttonPanel.setLayout( new FlowLayout());
JButton startButton = new JButton("Start");
startButton.addActionListener( this );
buttonPanel.add(startButton);
add(buttonPanel, "South");
}

public void actionPerformed(ActionEvent e)
{
fill( ); //Nothing else can happen until actionPerformed returns, which does
        // not happen until fill returns.
}



public void fill()
{
Graphics g = box.getGraphics();
```

```
for ( int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
for ( int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
{
g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
doNothing(PAUSE);    //Everything stops for 100 milliseconds(1/10 of a
                     second).
}
}

public void doNothing(int milliseconds)
{
try
{
Thread.sleep(milliseconds);
}
catch(InterruptedException e)
{
System.out.println("Unexpected interrupt");
System.exit(0);
}
}
}
```

RESULTING GUI (When started)

RESULTING GUI (While drawing circles)



If you click the close-window button while the circles are being drawn, the window will not close until all the circles are drawn.

RESULTING GUI (After all circles are drawn)

Issue with the program

Now that we have discussed the new items in the program in Display 9.1 , we are ready to explain why it is nonresponsive and to show you how to use threads to write a responsive version of that program.

Recall that when you run the program in Display 9.1 , it draws circles one after the other to fill a portion of the frame. Although there is only a 1/10 of a second pause between drawing each circle, it can still seem like it takes a long time to finish. So, you are likely to want to abort the program and close the window early. But, if you click the close-window button, the window will not close until the GUI is finished drawing all the circles.

Here is why the close-window button is nonresponsive: The method fill , which draws the circles, is invoked in the body of the method actionPerformed . So, the method actionPerformed does not end until after the method fill ends. And, until the method actionPerformed ends, the GUI cannot go on to do the next thing, which is probably to respond to the close-window button.

Solving the issue using Multithreading

Display 9.2 contains a program that uses a main thread and a second thread to implement the technique discussed in the previous subsection. The general approach was outlined in the previous subsection, but we need to explain the Java code details. We do that in the next few subsections.

Example: Display 9.2 Threaded Version of FillDemo

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
//The GUI produced is identical to the GUI produced by Display 19.1 except that in this
//version the close-window button works even while the circles are being drawn, so you can
//end the GUI early if you get bored.
public class ThreadedFillDemo extends JFrame implements ActionListener
{
public static final int WIDTH = 300;
public static final int HEIGHT = 200;
public static final int FILL_WIDTH = 300;
public static final int FILL_HEIGHT = 100;
public static final int CIRCLE_SIZE = 10;
public static final int PAUSE = 100; //milliseconds
private JPanel box;
public static void main(String[] args)
{
ThreadedFillDemo gui = new ThreadedFillDemo();
gui.setVisible( true );
```

```
}
public ThreadedFillDemo()
{
setSize(WIDTH, HEIGHT);
setTitle("Threaded Fill Demo");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLayout( new BorderLayout());
box = new JPanel();
add(box, "Center");
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout( new FlowLayout());
JButton startButton = new JButton("Start");
startButton.addActionListener( this );
buttonPanel.add(startButton);
add(buttonPanel, "South");
}
public void actionPerformed(ActionEvent e)
{
Packer packerThread = new Packer();
packerThread.start();
}
private class Packer extends Thread
run is inherited from Thread but needs to be
{
overridden. This definition of run is identical
public void run()
to that of fill in Display 19.1.
{
Graphics g = box.getGraphics();
for ( int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
for ( int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
{
g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
doNothing(PAUSE);
}
}
public void doNothing( int milliseconds)
{
try
{
Thread.sleep(milliseconds);
}
catch (InterruptedException e)
{
System.out.println("Unexpected interrupt");
System.exit(0);
}
```

```
}
} //End Packer inner class
}
```

The method actionPerformed in Display 9.2 differs from the method actionPerformed in our older, nonresponsive program (Display 9.1) in that the invocation of the method fill is replaced with the following:

Packer packerThread = new Packer( );

packerThread.start( );

This creates a new, independent thread named packerThread and starts it processing. Whatever packerThread does, it does as an independent thread. The main thread can then allow actionPerformed to end and the main thread will be ready to respond to any click of the close-window button

We need only to discuss the method start and we will be through with our explanation. 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. As we have already seen, the run method of the class Packer in Display 9.2 draws the circles we want, so the invocation

packerThread.start( );

does this as well, because it calls run . Note that you do not invoke run directly. Instead, you invoke start , which does some other needed things and then invokes run.

Exercise:

1) The definition of the class Packer includes no instance variables. So, why do we need to bother with an object of the class Packer ?Why not simply make all the methods static and call them with the class name Packer ?

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.run();
}
....
}
```

The previous 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 . In Display 9.3 , we have rewritten the program in Display 9.2 using the Runnable interface. The program behaves exactly the same as the one in Display 9.2 .

Example: Display 9.3 The Runnable Interface

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
public class ThreadedFillDemo2 extends JFrame
implements ActionListener, Runnable
{
public static final int WIDTH = 300;
public static final int HEIGHT = 200;
public static final int FILL_WIDTH = 300;
public static final int FILL_HEIGHT = 100;
public static final int CIRCLE_SIZE = 10;
public static final int PAUSE = 100; //milliseconds
private JPanel box;
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;
}
}
public static void main(String[] args)
{
ThreadedFillDemo2 gui = new ThreadedFillDemo2();
gui.setVisible( true );
}
public ThreadedFillDemo2()
```

```
{
setSize(WIDTH, HEIGHT);
setTitle("Threaded Fill Demo");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setLayout( new BorderLayout());
box = new Jpanel();
add(box, "Center");
Panel buttonPanel = new Jpanel();
buttonPanel.setLayout( new FlowLayout());
JButton startButton = new JButton("Start");
startButton.addActionListener( this );
buttonPanel.add(startButton);
add(buttonPanel, "South");
}

public void actionPerformed(ActionEvent e)
{
startThread();
}

public void run()
{
Graphics g = box.getGraphics();
for (int y = 0; y < FILL_HEIGHT; y = y + CIRCLE_SIZE)
for (int x = 0; x < FILL_WIDTH; x = x + CIRCLE_SIZE)
{
g.fillOval(x, y, CIRCLE_SIZE, CIRCLE_SIZE);
doNothing(PAUSE);
}
}

public void startThread()
{
ThrIn the run() method of Display 19.5 , make the thread sleep a random amount
of time between one and fi ve milliseconds. You should see an increase in the
number of problems caused by race conditions. Can you explain whead theThread
= new Thread( this );
theThread.start();
}

public void doNothing( int milliseconds)
{
```

```
try
{
Thread.sleep(milliseconds);
}
catch (InterruptedException e)
{
System.out.println("Unexpected interrupt");
System.exit(0);
}
}
}
```

Exercise:

2. Because sleep is a static method, how can it possibly know what thread it needs to pause?

3. Where was polymorphism used in the program in Display 9.2 ? (Hint: We are looking for an answer involving the class Packer .)

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 two. 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 and 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 such as

sharedVariable++;

Unfortunately, this will not 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, consider the Counter class shown in Display 9.4 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.

The way we will demonstrate the race condition is to do the following:

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 fi nish and then output the value of the counter. If there are no race conditions, then its value should be 30,000. If there are 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.

Example: Display 9.4 Counter Class

```
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 only new tool that we need for our demonstration program is a way to wait for all the threads to finish. If we do not 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 . This is a checked exception so we must use the try/catch mechanism.

The class RaceConditionTest in Display 9.5 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!

Example: Display 9.5 The RaceConditionTest Class

```
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)
The single instance of the Counter object.
{
Array of 30,000 threads.
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();
```

```
}
. In the run() method of Display 19.5 , make the thread sleep a random amount
of time between one and fi ve milliseconds. You should see an increase in the
number of problems caused by race conditions. Can you explain why?
// Wait for the threads to finish
start each thread.
for (i = 0; i < threads.length; i++)
{
try
{
Waits for the thread to complete.
threads[i].join();
}
catch (InterruptedException e)
{
System.out.println(e.getMessage());
}
}
System.out.println("The counter is " + masterCounter.value());
}
}
```

Sample Dialogue (output will vary)

The counter is 0

The counter is 29998

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 or parallel programming course.

Exercise:

4. In the run() method of Display 9.5 , make the thread sleep a random amount of time between one and fi ve milliseconds. You should see an increase in the number of problems caused by race conditions. Can you explain why?

5. Here is some code that synchronizes thread access to a shared variable. How come it is not guaranteed to output 30,000 every time it is run?

```
public class Counter
{
private int counter;
public Counter()
{
counter = 0;
}
public int value()
{
return counter;
}
public synchronized void increment()
{
counter++;
}
}
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();
}
System.out.println("The counter is " + masterCounter.
value());
}
}
```

Programming Tasks

1) Create a class named TimerAlarm that extends Thread and implements a timer. Do not use the built-in Timer class for this exercise. Your class constructor should take as input an integer named t representing time in milliseconds and an interface object named obj that defines a method named alarmAction( ) . You will need to define this interface. Every t milliseconds the class should invoke method obj.alarmAction( ) . Add pause( ) and play( ) methods that disable and enable the invocation of alarmAction( ) . Test your class with code that increments and

prints out a counter.

2) This program simulates what might happen if two people who share the same bank account happen to make a simultaneous deposit or withdrawal and the bank does not account for race conditions by recreating the situation described in Displays 9.4 and 9.5 with a simple BankAccount class. The BankAccount class should store

an account balance and have methods to retrieve the balance, make a deposit, and make a withdrawal. Do not worry about negative balances.

Next, create an array of thousands of threads where each thread has a reference to the same BankAccount object. In the run() method, even numbered threads deposit one dollar and odd numbered threads withdraw one dollar. If you create an even number of threads, then after all threads are done the account balance should be zero. See if you can find a number of threads so that you consistently end up with a balance that is not zero. If you want to increase the likelihood of a race condition, then make each thread sleep a short random number of milliseconds in the run() method.

Add the synchronized keyword to fix the problem and ensure a balance of zero after all the threads are done.

3) Can you really get errors by using an unsynchronized counter with multiple threads? Write a program to find out. Use the following unsynchronized counter class, which you can include as a nested class in your program:

```
static class Counter {
   int count;
   void inc() {
      count = count+1;
   }
   int getCount() {
      return count;
   }
}
```

Write a thread class that will repeatedly call the inc() method in an object of type *Counter*. The object should be a shared global variable. Create several threads, start them all, and wait for all the threads to terminate. Print the final value of the counter, and see whether it is correct.

Let the user enter the number of threads and the number of times that each thread will increment the counter. You might need a fairly large number of increments to see an error. And of course there can never be any error if you use just one thread.

4.1) Implement three classes: Storage, Counter, and Printer. The Storage class should store an integer. The Counter class should create a thread that starts counting from 0 (0, 1, 2, 3 ...) and stores each value in the Storage class. The Printer class should create a thread that keeps reading the value in the Storage class and printing it.

Write a program that creates an instance of the Storage class and sets up a Counter and a Printer object to operate on it.

4.2) Modify the program from the previous exercise to ensure that each number is printed exactly once, by adding suitable synchronization.