OOP Assignment V

1. Why an applet must use multithreading if it needs to run continuously? Explain with an example program.

If we don't use multithreading, or a timer, and update the contents of the applet continuously by calling the `repaint()` method, the processor has to update frames at a blinding rate. Too many calls to the `paint()` method might be coalesced (effectively ignored) and will block the GUI, making the applet unresponsive.

For example, the scrolling text (moving banner) problem without a thread looks like:

```
// Imports and shit
public class TextScrollNoT extends Applet {
    String text;

    public void init () { }

    public void start() {
     text = // get text
     while (true) {
           repaint();
           char ch = text.charAt(0);
           text = text.substring(1, text.length()) + ch;
      }
    }

    public void paint(Graphics g) {
      g.drawString(text, 20, 90);
    }
}
```

Here the `repaint()` method is being called many times per second (depending on the processor speed) on the main thread, too many calls being the reason for the GUI block and unresponsiveness, and undesired output.

The same problem with threads looks like:

```
// Imports and shit
public class TextScroll extends Applet implements Runnable {
    String text;
    public void init () { }

    public void start() {
     text = // Get text
      Thread t = new Thread(this);
      t.start();
    }
```

```
    public void run() {
      char ch;
      try {
            while (true) {
                  repaint();
                  Thread.sleep(150);
                  ch = text.charAt(0);
                  text = text.substring(1, text.length()) + ch;
            }
      }
      catch (InterruptedException e) { }
    }

    public void paint(Graphics g) {
      g.drawString(text, 20, 90);
    }
}
```

The use of a thread shifts the overhead of updating the frames to a child thread instead of bulking up the main thread, making the UI responsive. The use of the `sleep()` method causes the updating calls to wait a while before performing the next update, reducing the number of times the GUI is refreshed, giving the desired output.

2. What are events, event sources and event listeners? Explain with a simple example program?

The java window manager sends a program an **Event** notification when a certain action such as mouse movement, key presses, selections, etc. are preformed. The window manager generates various kinds of events the program needn't listen to.

A program must specify what kind of events it wants to receive. This is done by installing appropriate **Event Listener** objects.

The UI component that generates the event is called the **Event Source** (duh).

An **Event Listener** performs a check on when a **Event** has occurred, and takes appropriate action accordingly.

For example, we want to draw a line based on the mouse movement. To achieve this, we will

1. Listen to the **MouseEvent**
2. Add appropriate **MouseListener** and **MouseMotionListener**s
3. Perform the action (draw line in this case) based on the listeners.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

/* <applet code="DrawLine" width=848 height=480>
 * </applet>
 * */

public class DrawLine extends Applet {

    Point startPoint, endPoint;

    public DrawLine () {

    // For pressing and releasing
        addMouseListener(new MouseAdapter() {

            public void mousePressed (MouseEvent e) {
                super.mousePressed(e);
                startPoint = e.getPoint();
            }

            public void mouseReleased (MouseEvent e) {
                super.mouseReleased(e);
                endPoint = e.getPoint();
                repaint();
            }
        });

        // For the motion event
        addMouseMotionListener(new MouseMotionAdapter() {

            public void mouseDragged (MouseEvent e) {
                endPoint = e.getPoint();
                repaint();
            }

        });
    }

    public void init () {
        super.init();
    }

    public void paint (Graphics g) {
        super.paint(g);
        g.drawLine(startPoint.x, endPoint.x, startPoint.y,
endPoint.y);

    }
}
```

3. Write a java program that will implement the following:

a. **Writes some primitive data to a disk file.**

b. **It then reads the raw bytes to check how the primitives were stored.**

c. **Finally, it reads the data as primitives.**

```
// PrimitveReadWrite.java
import java.io.*;

public class PrimitveReadWrite {

     public static void main (String [] args) throws IOException {

      FileOutputStream fout = new FileOutputStream("Test.dat");
         DataOutputStream dos = new DataOutputStream(fout);
         dos.writeDouble(1812.4008);
         dos.writeInt(666);
         dos.writeChar('X');
         dos.close();


         BufferedInputStream bufferedInput = new
BufferedInputStream(new FileInputStream("Test.dat"));
         byte[] buffer = new byte[22];
         bufferedInput.read(buffer);
         System.out.print("\n\nRaw Read: ");
         System.out.write(buffer);

         // Raw Read: @?Q?kP???X


         FileInputStream fin = new FileInputStream("Test.dat");
         DataInputStream din = new DataInputStream(fin);
         double d = din.readDouble();
         int i = din.readInt();
         char c = din.readChar();
         System.out.println("\n\nPrimitve read: " + d + ", " + i +
", " + c);
         din.close();

         // Primitve read: 1812.4008, 666, X

         // Content in the file (visible): @?Q?kP???X

     }
}
```

4. Write a java program that will copy the contents of one file to another using a programmer managed Buffer.

```
import java.io.*;

public class Copy3 {

    public static void main (String [] args) throws IOException,
FileNotFoundException {

    File source = null;
    File dest = null;

    InputStream is = null;
    OutputStream os = null;

    try {

        source = new File(args[0]);
        dest = new File(args[1]);

        is = new FileInputStream(source);
        os = new FileOutputStream(dest);

        byte[] buffer = new byte[1024];
        int length;

        while ((length = is.read(buffer)) > 0) {
             os.write(buffer, 0, length);
        }

    }
    finally {
        is.close();
        os.close();
    }
  }
}
```

5. What are anonymous inner classes? Explain with a swing program defining anonymous inner class to handle some events.

A class that has no name, and is defined inside another class is called an anonymous inner class. It is used to override a method of a class or an interface. The java compiler automatically creates a class whenever required.

For example in swing based programs:

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class SumApplication {

    JTextField field1, field2;
    JButton computeButton;
    JLabel labelSum;

    public SumApplication () {

        JFrame frame = new JFrame("Number Operations");
        frame.setSize(600, 400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());

        field1 = new JTextField("");
        field2 = new JTextField("");

        JPanel fieldPanel = new JPanel(new GridLayout(3, 1));

        computeButton = new JButton("Compute");

        computeButton.addActionListener(new ActionListener() {
            // Inner class is made here…
            public void actionPerformed(ActionEvent e) {
                double a = 0.0, b = 0.0;
                double sum = 0.0;
                try {
                    a = Double.parseDouble(field1.getText());
                    b = Double.parseDouble(field2.getText());
                    sum = a + b;

                }
                catch (Exception e1) { // Handle error and shit  }
                finally {
                    labelSum.setText("  " + a + "\n+ " + b + "\n= " +
sum);
                }
            }
        });
```

```
        fieldPanel.add(field1);
        fieldPanel.add(field2);
        fieldPanel.add(computeButton);

        labelSum = new JLabel("", JLabel.CENTER);

        frame.add(fieldPanel);
        frame.add(labelSum);

        frame.setVisible(true);

    }

    public static void main (String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SumApplication();
            }
        });

    }

}
```

The above program on compilation generates three .class files, one for the main class and two for each of the anonymous inner classes created:

1. `ActionListener` class object added to the `computeButton` as an action listener
2. Inner class implementing the `Runnable` interface, inferred from the `SwingUtilities.invokeLater` block, that dispatches the applet to a child thread.