Graphical User Interfaces – GUI

Flavors of Java

Java was designed to be the language of the Internet. It uses a virtual machine architecture so that once a Java program is compiled, that program can run on any computer that has a Java virtual machine, JVM, of the same version, regardless of manufacturer or operating system. What this means to you as a programmer is that you no longer have to worry about changing your program when a new operating system comes out. Highly paid systems programmers at places like Javasoft do that work for you, leaving you time to improve your program.

You are already familiar with two different flavors of the JVM, the one on your PC that came in the J2SDK and the one on the RCX that came in LeJOS. A JVM provides services, threads, file access, communications, etc., that are typical of an operating system. These two JVMs provide very different levels of service. For example, the RCX JVM does not provide methods to access files because on the RCX there is no disk to store files on. Many other methods are missing because there is simply not enough memory on the RCX to hold the code for these methods. Likewise, classes like RCXPort are not in the standard Java because RCX IR ports are very uncommon on general-purpose computers.

The point of this discussion is to highlight the fact the capabilities of programs written in Java depend upon the virtual machine that the program is running on. It may surprise you to learn that many computers have two JVMs installed and that the one that comes with the J2SDK is not the more common one.

Virtually every web browser has an embedded JVM. Like the LeJOS JVM, this JVM has severely limited capabilities. By default, programs running on this JVM cannot save or delete files. Programs cannot open connections to other computers. They are not allowed to run other programs. In fact, programs running on a browser’s JVM are not even called programs, they are called applets. The reason for this is clear; on the Internet there are bad people writing programs. These people delight in causing other people grief by deleting files, crashing computers or disrupting mail service. They write programs called viruses or worms that spread throughout the Internet causing problems on thousands, or millions, of computers. The developers of Java made it extremely difficult to write these bad programs and spread them through web browsers.

A real strength of Java is writing graphical programs that run on the web. That area is beyond the scope of this book because applets cannot communicate with the RCX. An appendix describes the differences and gives example code for an applet.
Swing and AWT

Graphical user interfaces, GUIs, are the preferred way for modern programs to communicate with human users. The standards for the appearance of window-based programs date back around 20 years, to the CUA standard. CUA required programs to have a certain “look”. A program needed a solid rectangle for its main window. Across the top is a bar for a title and an “X” that the user can click on to close the window. Main frames also have a menu bar located just under the title bar. The menu bar starts with the words “File” and “Edit” on the left and has “Help” as the rightmost entry. Under “File”, clicking on “Open” opens a file chooser box. This standard continues on for many pages, describing what programs should, and do, look like. This standard has become so pervasive that most windowing operating systems implement these elements, e.g. frames, file choosers, so user programs can call them directly instead of having to start from scratch for every program.

Java has two different sets of classes to implement GUIs: AWT and Swing. AWT, Abstract Windowing Toolkit, was introduced with the first version of Java. AWT relies on the operating system to provide display elements, such as frames and file choosers, by simply wrapping the operating system call in Java code (this is the same as forward() and backward() wrapping calls to ROM in MyMotor.java). This made it easier for the system programmers who created Java to finish the JDK. It also makes the Java applications look like any other application on a particular computer. The problem with this is that different computers have a different look for all the elements and a Java program that looks good on one computer looks terrible on another. For that reason, making a Java based program look the same on all computers, Swing was born.

Swing moves a lot of the functionality provided by operating systems into Java libraries. It largely replaces AWT for writing new code, although the parts of AWT that don’t make operating system calls, like picking colors, are still used. The naming convention for elements that have been replaced is to prepend a J, e.g. Button becomes JButton.

Just to be clear, Swing is huge with hundreds of classes and thousands of methods. Each item you see on the screen, e.g. buttons, scroll bars, text fields, etc., has its own class in the Swing library. In the next sections, we will create a Swing based application that uses many elements. Note that using AWT elements in a Swing application where equivalent Swing elements exist, e.g. Button (AWT) and JButton (Swing), should not be done.
**HelloWorldG.java**

As usual, we start with “Hello World”. At right is GUI version that displays “Hello World” in the title bar of a frame. Three of the four lines are obvious. The first line creates a new Swing frame object. The third line sets the size in pixel, 400 pixels wide by 300 pixels high. The fourth line makes the frame appear on the screen (by default frames are not displayed because a you may want to wait until you have completed adding text and graphics to the display). The second line is a bit less clear.

You should recall creating classes, GoHome.java and CenterBump.java, which implemented the SensorListener and TimerListener interfaces. These classes had methods that were called when a specific event occurred, e.g. when the robot ran into something, the stateChanged method was called in CenterBump.

MyWindowAdapter is the same sort of thing. It extends WindowAdapter and overrides the WindowClosing method. By default, WindowClosing simply makes the window, in this case our JFrame, disappear. It does not cause a program to stop running, which is the right thing to do most of the time. However, here we do want to stop the program so we override the method by extending the class and replacing the one method.

This is a good point to highlight the difference between implementing interfaces and extending classes. An interface simply specifies the names of methods that an implementing class has to define. GoHome had to have a method named “timedOut” because the TimerListener interface required it. It would be an error if GoHome did not define it. On the other hand, extending another class means that all the methods in the extended class exist in your new class. You can override existing methods by defining new methods with the same name but you don’t have to. WindowAdapter has 10 methods that implement four different interfaces that have to do with windows changing. By extending this class and overriding one method, we don’t have to deal with implementing all the other methods associated with those interfaces but those methods will be there and, we assume, will do something reasonable if they are ever called.
**HelloWorldG2.java**

The previous version, HelloWorldG, opened a frame and displayed “Hello World” but did not do it in a graphical way. This new version, HelloWorldG2.java, HWG2 for short, does. We’ll see how by examining the differences between the versions.

First, HWG2 extends JPanel. JPanel is container. You can put things into it and you can draw on it. HWG2 extends it so that we have a place to draw.

After creating an instance of HWG2, we have an instance to draw on and then we add it to the content pane of our JFrame so that it will show up on our screen when we make the JFrame visible.

Finally, we override the paint method from JPanel. Every time an element in a GUI is to be displayed, its paint method is called. The parameter to the paint method is the graphics object that the instance will paint upon. You can picture this as a tablet or canvas. In this example, we set the color to red and then draw “Hello World” at location (50,50), i.e. 50 pixels from the top edge and 50 pixels from the left edge of our graphics object.
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.Vector;

class DummyDisplayMap extends HelloWorldG2 {
    static public void main(String[] args) {
        JFrame jf = new JFrame();
        jf.addWindowListener(new MyWindowAdapter());
        getData();
        jf.setSize(400, 300);
        jf.getContentPane().add(new DummyDisplayMap());
        jf.setVisible(true);
    }
    public void paint(Graphics g) {
        if (coorVec.size() == 0) return;
        int height = getSize().height;
        int width = getSize().width;
        g.setColor(Color.orange);
        g.drawLine(width/2, 0, width/2, height);
        g.drawLine(0, height/2, width, height/2);
        java.util.Iterator vi = coorVec.iterator();
        g.setColor(Color.black);
        Point endPt = (Point) vi.next();
        Point begPt = null;
        while (vi.hasNext()) {
            begPt = endPt;
            endPt = (Point) vi.next();
            g.drawLine(begPt.x, begPt.y, endPt.x, endPt.y);
        }
    }
    static Vector coorVec = new Vector();
    static void getData() {
        for (int idx = 0; idx < 10; idx++) {
            Point x = new Point((int) (Math.random() * 100),
                          (int) (Math.random() * 100));
            coorVec.add(x);
        }
    }
}

**DummyMapData.java**

The program at right is a first step to graphing the map data that our robot has found. This class starts by extending HelloWorldG2. Doing so gives us access to our JPanel for drawing and our MyWindowAdapter class for keeping track of our windows.

We add one method to main, getData. In this dummy program, getData simply creates 10 points, XY pairs, by calling the random number generator, which returns a float with a range of 0.0 to 1.0, and multiplying by 100 to give us ints with a range of 0 to 100. It stores each XY pair in a Point object. These objects are then stored in a Vector. A Vector is like an array but more robust because it grows as you add things to it and the things you add don’t have to be of the same class.

Finally, the paint method has been changed. First, the size of the graphics object is found and axes are drawn to bisect it both horizontally and vertically. Then we use the iterator interface to retrieve the points in order. Note that we are not drawing points; we are drawing lines and a line is defined by its two end points. To initialize our line, we get an end point. But the end point of one line is the beginning point of the next. So we loop while we have another point, making the last end point our next beginning point, getting another end point and drawing the line that connects them.
**MapData.java**

This is the penultimate, next to last, program in this series. In this program, we combine receive data from the last chapter on communications with our map display program. We extend DummyDisplayMap so that we don’t have to replicate the paint method or the MyWindowAdapter class. Note that we are creating a new main method in each new class since our class name changes each time and we use the class name to create an instance of our class in main.

We have copied getData into this class definition. Java, as compared to C++, does not allow a class to extend more than one other class. There are reasons for this restriction that are beyond the scope of this book.
**MapDisplayButton.java**

MapDisplayButton is the end of our journey into Swing. This class adds a button so that you can upload data on command.

There are several new pieces in this class. First, a constructor has been added, the method MapDisplayButton. Note that this method does not have a type; since its name is the same as the class, it returns, without needing a return statement, an instance of an object of this class. The reason for having a constructor is that we are doing something that cannot be conveniently done in a static context, adding a listener to a button.

Creating a button is also new. It’s created at the class level so it is available to any method in the class. It is also an instance variable; every time a new MapDisplayButton instance is created a new JButton with the name of JB_recv is also created.

Creating the button is one thing; using it is another thing entirely. To be useful, a button needs a listener. SymAction is another listener class, just like SensorListener and MyWindowAdapter. Whenever a button is pressed, it generates an ActionEvent that is passed to every registered listener by calling the listener’s actionPerformed method. In SymAction, we check to see if the source is the JB_recv button (since we only have one button, it is fairly certain that it is JB_recv). If it is, we get data from the RCX and then repaint our panel. repaint calls the paint methods for every component in our MapDisplayButton instance. That displays the newly received mapping data.
Last Word on Swing, AWT and GUIs

This is just the briefest possible introduction to Swing, AWT and GUIs. Graphic User Interfaces are the key to Java. If you are at all serious about programming, the next step is to get an IDE with a GUI builder. The major products in this category include Sun’s Forte, Borland’s JBuilder and IBM’s Visual Age. There are versions of these products that are free and other versions that cost thousands of dollars. And, you do get what you pay for.

You can build GUIs without a GUI builder. However, keeping track of the code is tedious, at best. GUI code gets very big and very ugly. By that I mean, each button you add to your application requires a minimum of 4-6 lines of code. Fifty buttons generate 200-300 lines of code with very slight, but significant, differences. Maintaining and updating such code is hard.

A GUI builder is based upon the principles of RAD, Rapid Application Development. You start with a blank screen and a palette of tools. You click on a tool and place the selected object, e.g. JFrame, JButton, JTextField, etc., where you want it. The GUI builder then generates the source code (a text file that you can edit), including listeners, for you to compile. If you want to move objects around or change their sizes, you do so using drag and drop techniques.

One technique that I have found useful is to never edit the files the GUI builder produces. Instead, I extend the class and put all of my routines, like the getData method in the last example, into the new class. The reason for this is clear the first time the GUI builder eats the code you haven’t backed up.

In closing, notice how each class built upon the last class. At each step, we made sure our code did what we expected it to. Then, we added new code knowing that our old code did what it was supposed to. This is a debugging technique that is very powerful. It is also in tune with a fundamental philosophy of Java; code once, use many times.