Programming with Java GUI components

Java includes libraries to provide multi-platform support for Graphic User Interface objects. The "multi-platform" aspect of this is that you can write a program on a Macintosh and have the same graphic objects show up when the program is run under UNIX or Windows (modulo some minor problems with arrangements of objects on the screen.

Java's GUI components include labels, text fields, text areas, buttons, pop-up menus, etc. The SWING Toolkit also includes containers which can include these components. Containers include frames (windows), canvases (which are used to draw on), and panels (which are used to group components). Panels, buttons, and other components can be placed either directly in frames or in panels inside the frames.

These GUI components are automatically drawn whenever the window they are in is drawn. Thus we will not need to explicitly put in commands to draw them.

Actions on these GUI components are handled using Java's event model. When a user interacts with a component (clicks on a button, types in a field, chooses from a pop-up menu, etc.), an event is generated by the component that you interact with. For each component of your program, the programmer is required to designate one or more objects to "listen" for events from that component. Thus if your program has a button labeled "start" you must assign one or more objects that will be notified when a user clicks on the button. We begin our discussion of GUI programming in Java with a discussion of buttons in more detail.

If you would like more information on how to use GUI components in Java, there are excellent tutorials at Sun's Java website or elsewhere on the web. In particular, this document does not talk about how to use layout managers. Those that are most useful are the `FlowLayout`, `GridLayout`, and `BorderLayout` managers.

1 Buttons:

JButton is a class in package javax.swing that represents buttons on the screen. The most common constructor is:

```
public JButton(String label);
```

which, when executed, creates a button with "`label`" printed on it. Generally the button is large enough to display label. There is also a parameterless constructor that creates an unlabeled button.

Buttons respond to a variety of messages, but the only one likely to be useful to you is getText(), which returns a String representing the label on the button.

You add an "`ActionListener`" to a button with the method:

```
public void addActionListener(ActionListener listener);
```

We will discuss how to create `ActionListener`s below.

1.1 Adding buttons to a JFrame or JPanel

We can add a button to a frame (window) or panel of a frame by sending the message:

```
myFrame.add(startButton);
```

Normally we include such code in the constructor for the frame or panel. Hence usually we just write `this.add(startButton)` or simply `add(startButton).`

Below is some sample code for a class, `ButtonDemo`, representing a specialization of `JFrame` with two buttons. The class extends `JFrame`, which is part of `javax.swing`. The constructor for `JFrame` takes a `String` parameter and creates a window with that string in the title bar. The constructor for `ButtonDemo` calls the superclass constructor, and then sets the size of the new `JFrame` to be 400 x 200. The `setLayout` command tells the new `JFrame` that new components should be added from left to right across the panel. If it runs out of space new components will be added in new rows, again from left to right, if space is available (often, it is not). The code for creating buttons should be self-explanatory. The add commands add the buttons to the frame. We explain the `JButtonListener` class below. It creates objects which perform actions in response to button clicks. Thus we tell the two buttons (with the `addActionListener` method) to notify `myButtonListener` when the user clicks on them. Finally, note that we had to import `java.awt.*` because `Color` and `FlowLayout` are in that package, we had to import `javax.swing.*` because `JFrame` and `JButton` are classes in that package, and we had to import `java.awt.event.*` because `ActionEvent` and `ActionListener` are from that package. You might want to automatically import all three packages in all GUI programs you write, though if you are creating a ".jar" file or double-clickable application then importing the entire libraries may add unneeded bulk to your files. In that case, just import exactly the classes you are using.

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

public class ButtonDemo extends JFrame {
    protected JButton startButton, stopButton;

    // Constructor sets features of the frame, creates buttons, adds them to the frame, and
    // assigns an object to listen to them
    public ButtonDemo() {
        super("Button demo"); // set title bar

        setSize(400,200); // sets the size of the window

        startButton = new JButton("Start"); // create two new buttons w/labels start and stop
        stopButton = new JButton("Stop");

        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout()); // layout objects from left to right
                                                 // until fill up row and then go to next row
        contentPane.add(startButton);
        contentPane.add(stopButton);

        // create an object to listen to both buttons:

        ButtonListener myButtonListener = new ButtonListener();
        startButton.addActionListener(myButtonListener);
        stopButton.addActionListener(myButtonListener);
    }
    // Trivial main program associated with new window
    public static void main(String[] main) {
        ButtonDemo app = new ButtonDemo(); // Create an instance of Buttons
        app.setVisible(true);              // Show it on the screen
    }                                  // -- leave this out and you won't see see frame!
}
```

The main program associated with the class simply creates an object from the class and tells it to display itself on the screen. Main programs associated with frames almost always do this minimal amount of work.

1.2 ActionListeners for Buttons

Objects which implement the `ActionListener` interface are required to implement a method `actionPerformed` which takes a parameter of type `ActionEvent`. When a user clicks on a button, all objects that have been added as `ActionListener`'s for that button are notified by calling their `actionPerformed` method with a parameter with information on the exact event that occurred. Thus the system automatically creates the `ActionEvent` object and sends it along to the listener. You are unlikely to ever need to manually create an `ActionEvent` object in this course.

The most useful method of `ActionEvent` is

```
    public Object getSource();
```

If you send this message to an `ActionEvent` representing a click of a button, it will return the button that was clicked. The class `ButtonListener` below implements `ActionListener`, and hence an object of this class can be added to a button as a listener. The `actionPerformed` method is very simple. It uses `getSource` to obtain the button clicked and then, depending on what the button was, prints a different message. (Recall that the same `ActionListener` was added to each of the two buttons in `ButtonDemo` so the `actionPerformed` method needs to figure out which one was clicked.) Note that because this class is so simple (it doesn't even have instance variables) it needs no explicit constructor. Instead, like all classes without explicit constructors, it has a default constructor (with no parameters) that just allocates storage for new objects constructed. Note that we must import the `java.awt.event` package because `ActionEvent` and the `ActionListener` interface are both in that package.

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

public class ButtonListener implements ActionListener{

  public void actionPerformed(ActionEvent evt) {
     Object actionSource = evt.getSource(); // Label of the button clicked in evt
     if (actionSource == startButton) {
        System.out.println("Start button");
     } else if (actionSource == stopButton) {
        System.out.println("Stop button");
     }
  }
}
```

The careful reader will notice that the above class has a major problem. We have used the variables `startButton` and `stopButton`, but they are not defined as either instance variables or parameters.

We could get around this with a bit of extra work either by passing in the buttons in the constructor for the class or by casting the results of `evt.getSource()` to be of type `JButton` and then sending it the message `getText()`, which would return the label on the button as a string. That could then be compared with the actual labels, `"Start"` and `"Stop"`. However, in the next section we will see there is a solution that will allow the listener class access to all of the instance variables (including `startButton` and `stopButton`) of `ButtonDemo`.

1.3 Inner Classes

While the code presented above defines a class which is an extension of JFrame and another class implementing `ActionListener`, it seems a bit heavy to have to create a completely separate class (which goes in a separate file) in order to create a listener for the two buttons.

Two alternatives are possible. One is to let the frame itself be the `ActionListener` for the button. We can do this by changing the original class header to read:

```
    public class ButtonDemo extends JFrame implements ActionListener
```

We must also copy the `actionPerformed` method from class `ButtonListener`, and finally, we change the two statements adding the `ActionListener` to have the object itself do the listening:

```
     // tell buttons that this object should be notified
     startButton.addActionListener(this);
     stopButton.addActionListener(this);
```

This is the style generally used in CS 51 for handling action events in simple cases. The advantage to that only one class is needed instead of two. However it is generally good design to separate the code for building and handling GUI components from the code which actually does the work spawned by the action events. Moreover, there is another style which is almost as simple, but more general. It involves the use of what are called "inner classes".

Java allows classes to be nested in each other. Often when dealing with events we will have a class which is so special that it will really only ever be used in conjunction with another class. Rather than making this auxiliary class public, we can stick it inside of the other class. Thus we could include the `ButtonListener` class inside of `ButtonDemo`. The code now would have the following structure:

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

public class ButtonDemo extends JFrame {
    protected JButton startButton, stopButton;

    public ButtonDemo() {
        ...
        // create an object to listen to both buttons:
        ButtonListener myButtonListener = new ButtonListener();

        // tell buttons that myButtonListener should be notified
        startButton.addActionListener(myButtonListener);
        stopButton.addActionListener(myButtonListener);
    }
    public static void main(String[] main) {
        ... // As before
    }

    private class ButtonListener implements ActionListener{

        public void actionPerformed(ActionEvent evt)  {
            ... // As before
        }
    }
}
```

Notice that ButtonListener is now declared to be private, meaning that it can only be used inside the containing class, ButtonDemo. The method actionPerformed is still public, however. If it were declared protected, then it would only be available from within ButtonListener and not within ButtonDemo (and we need to use it in the constructor for ButtonDemo).

These nested classes should be contained within a single file named ButtonDemo.java. Another advantage of using nested classes is that all of the instance variables (and methods) of the outer class are visible inside the inner class. This can be handy for picking up information from other components of the outer class.

1.4 A further refinement

Some experts might object to the design of the ButtonListener class above because it handles two different buttons. Indeed we could divide it into two separate classes

```
protected class StartButtonListener implements ActionListener{

    public void actionPerformed(ActionEvent evt) {
        System.out.println("Start button");
    }
}

protected class StopButtonListener implements ActionListener{

    public void actionPerformed(ActionEvent evt)  {
        System.out.println("Stop button");
    }
}
```

The code in the ButtonDemo constructor assigning listeners would then be replaced by:

```
    startButton.addActionListener(new StartButtonListener());
    stopButton.addActionListener(new StopButtonListener());
```

Each of the actionPerformed methods is now much simpler because there is no need to determine what button caused the event.

We can go further still by recognizing that the only difference between the two classes is the string to be printed in each case. We can accommodate those differences by including a constructor which takes a String parameter:

```
protected class BetterButtonListener implements ActionListener {
    private String buttonType;

    public BetterButtonListener(String buttonType) {
        this.buttonType = buttonType;
    }

    public void actionPerformed(ActionEvent evt) {
        System.out.println(buttonType + " button");
    }
}
```

The code in the `ButtonDemo` constructor assigning listeners would then be replaced by:

```
    startButton.addActionListener(new BetterButtonListener("Start"));
    stopButton.addActionListener(new BetterButtonListener("Stop"));
```

Thus there are two different listeners, but they are both instances of the same class.

2 Other GUI components

2.1 Labels

A `JLabel` is a very simple component which contains a string. The constructors are

```
    public JLabel()              // creates label with no text
    public JLabel(String text)   //create label with text
```

The methods available are

```
    public String getText()         // return label text
    public void setText(String s)   // sets the label text
```

2.2 Text Fields

A `JTextField` is an area that the user can type one line of text into. It is a good way of getting text input from the user. The constructors are

```
    public JTextField ()             // creates text field with no text
    public JTextField (int columns)
            // create text field with appropriate # of columns
    public JTextField (String s)     // create text field with s displayed
    public JTextField (String s, int columns)
            // create text field with s displayed & approp. width
```

Methods include:

```
    public void setEditable(boolean s)
           // if false the TextField is not user editable
    public String getText()         // return label text
    public void setText(String s)   // sets the label text
```

Many other methods are also available for this component (see also the documentation for its superclass, JTextComponent).

When the user types into a text field and then hits the return or enter key, it generates an event which can be handled by the same kind of ActionListener used with JButtons. Thus in order to respond to the user's hitting of the return key while typing in a JTextField, myField, we can write:

```
public void actionPerformed(ActionEvent evt)  {
     String contents = myField.getText();
     System.out.println("The field contained: "+contents);
}
```

Notice that this use of myField in actionPerformed only makes sense if we either have an inner class (and hence have access to all of the instance variables of the outer class) or if the JFrame or JPanel where myField is defined is the associated ActionListener.

If for some reason you'd like to be notified every time any change is made to the JTextField (whether or not the user hit the return key), one can associate a TextListener to the field. See details in the Java on-line documentation.

2.3 Text Areas

JTextArea is a class that provides an area to hold multiple lines of text. It is fairly similar to JTextField except that no special event is generated by hitting the return key.

The constructors are:

```
public JTextArea(int rows, int columns)
    // create text field with appropriate # rows &amp; columns
public JTextArea(String s, int rows, int columns)
    // create text field with rows, columns, &amp; displaying s
```

Methods:

```
public void setEditable(boolean s)
    // if false the text area is not user editable
public String getText() // return text in the text area
public void setText(String s) // sets the text
public void append(String s)  // append the text to text in the text area
```

Unlike JTextField, hitting return or enter does not generate an event. See the example CompressedGrid from your homework.

2.4 JComboBox menus

JComboBox provides a pop-up list of items from which the user can make a selection.

The constructor is:

```
public JComboBox()     // create new choice button
```

The most useful methods are:

```
    public void addItem(Object s) // add s to list of choices
    public int getItemCount()     // return # choices in list
    public Object getItemAt(int index) // return item at index
    public Object getSelectedItem() // return selected item
    public int getSelectedIndex() // return index of selected
```

When an object is added to a `JComboBox`, the results of sending `toString()` to the combo box is displayed in the corrresponding menu.

You can also set the combo box so the user can type an element in the combo box rather than being forced to select an item from the list. This is done by sending the combo box the message `setEditable(true)`.

When a user selects an item it generates an ActionEvent which can be handled as usual with an `actionPerformed` method. One can determine whether the combo box generated the event by sending the event the getSource() method, just as we did with other components.