Completing a Simple Graphical Interface

Purpose:
In this lab, we complete the application begun in a previous lab. We see how to capture user input and how the user interface interacts with the model. (Note: the implementation we develop is simple and minimal. It does not exhibit the desired MVC structural pattern.)

Setup:
Create a directory named Lab212/gui in your Java directory, and copy the files from ~labCourse/Labs/Lab212/gui to your gui directory.

The application:
The application will simply count the number of times a button is pressed. The model is a basic counter. We provide a class Lab212.gui.Counter for this purpose.

Functionality of the user interface:
The user interface is the same as in a previous lab. It will contain a button and label. The label displays the number of times the button has been pressed. Compile and execute CounterGUIStart to remind yourself of what the interface looks like.

Event listeners:
Any user action, such as pressing a button, clicking the mouse, keying a character, etc. can generate one or more events. In particular, pressing the button labeled “Press me!” generates an action event. This event is modeled by an instance of the class java.awt.event.ActionEvent. Closing the application window generates a window event, modeled by an instance of the class java.awt.event.WindowEvent.

We need to create objects, called listeners, that will be notified of these events. The listeners will respond to the events in an appropriate way. The listener for the button action event will respond by incrementing the count. The listener for the window closing event will respond by terminating the program.

A Window listener:
• Open the file Show.java (which is in your Java/Lab212/gui directory).

The Show method inFrame creates a JFrame, which is a subclass of java.awt.Window. Thus a JFrame is a component that can generate WindowEvents. We will implement a WindowListener that responds to the events generated by the JFrame.

• Scan the documentation of the interface java.awt.event.WindowListener. (Java API documentation can be found here.)
If we want to implement a `WindowListener`, we must implement the seven methods specified by this interface. The only events we are really interested in are those that result from the window being closed. These are `WINDOW_CLOSING`, which occurs when the user attempts to close the window, and `WINDOW_CLOSED` which occurs when the window has been closed. Thus the methods we are interested in implementing are

```java
public void windowClosing (WindowEvent e)
```

and

```java
public void windowClosed (WindowEvent e)
```

What about the other five methods specified in the interface `WindowListener`? We could simply implement them as “do nothing” methods. But since it is common that an implementation is interested in only one or two of the methods specified in the interface, the standard libraries include a class `java.awt.event.WindowAdapter`. This class implements the `WindowListener` interface with default, empty methods. All we need to do is extend this class, and override the methods we are interested in.

To summarize, we need a subclass of `WindowAdapter` in which we implement the methods `windowClosing` and `windowClosed`. The result should be that when a user closes the application’s window, the application terminates.

What should we do when the user attempts to close the window, generating a `WINDOW_CLOSING` event? The standard approach is to release the window’s resources and close the window. This is accomplished by invoking the window’s `dispose` method. Disposing of the window causes a `WINDOW_CLOSED` event to be generated. We handle this event in the `windowClosed` method by terminating the program. We implement the appropriate methods quite simply as follows:

```java
public void windowClosing (WindowEvent e) {
    e.getWindow().dispose();
}
public void windowClosed (WindowEvent e) {
    System.exit(0);
}
```

We create an instance of an anonymous subclass of `WindowAdapter` overriding `windowClosing` and `windowClosed` methods as follows:

```java
new WindowAdapter () {
    public void windowClosing (WindowEvent e) {
        e.getWindow().dispose();
    }
    public void windowClosed (WindowEvent e) {
        System.exit(0);
    }
}
```

To associate this instance with our particular `JFrame`, we use the `JFrame` method

```java
public void addWindowListener (WindowListener l)
```
(Recall that `WindowAdapter` is a class that implements the interface `WindowListener`.)

The method `inFrame` of `Show` requires a `JPanel`, window title, and window width and height. The fragment:

```java
if (windowTitle == null || windowTitle.equals(""))
    windowTitle = jp.getClass().toString();
```

gives `windowTitle` the name of the class of `jp` as a default.

The line

```java
JFrame frame = new JFrame(windowTitle);
```

creates a `JFrame` instance with `windowTitle` as its title.

The fragment:

```java
frame.addWindowListener(
    new WindowAdapter() {
        public void windowClosing (WindowEvent e) {
            e.getWindow().dispose();
        }
        public void windowClosed (WindowEvent e) {
            System.exit(0);
        }
    });
```

adds a `WindowListener` as described above.

The line

```java
frame.getContentPane().add(jp,BorderLayout.CENTER);
```

adds the `JPanel` to the `JFrame`. (`BorderLayout` is the default layout manager for a `JFrame`’s content pane. The content pane itself is a `JFrame`.)

The last fragment,

```java
if (width == 0 || height == 0)
    frame.pack();
else
    frame.setSize(width,height);
    frame.setVisible(true);
```

sizes the window and makes it visible.

**Handling `ActionEvents` generated in the `JPanel`:**

The `JPanel` presents visual components that interact with the model. In our case, the model is a simple counter. A `JButton` is used to increment the counter. A `JLabel` displays the current counter value. We need to program a listener for the `JButton`. 
A listener for the JButton:

We must define a class to handle the events generated by the JButton. This class will implement the interface java.awt.event.ActionListener.

• Scan the documentation of the interface java.awt.event.ActionListener.

Notice that this interface specifies only one method:

```java
void actionPerformed (ActionEvent e)
```

Our implementation of this method will increment the counter, and then display the new count in the JLabel. To change the text of the JLabel, we can use the JLabel method setText.

The constructor of the class that implements ActionListener will take a Counter and JLabel as arguments. We name the class SimpleButtonListener. The constructor is specified as follows:

```java
public SimpleButtonListener (Counter counter, JLabel label)
```

The method actionPerformed increments the Counter, and displays the new count in the JLabel.

• Implement the class SimpleButtonListener.

Associating the SimpleButtonListener with the JButton:

We now need to complete the class CounterGUI. This class must have access to the model: either it gets a model instance as a constructor argument, or it creates one in the constructor. To make things simple, we’ll have the CounterGUI constructor create a Counter instance.

• Modify the CounterGUI constructor to create a Counter instance.

Next we must associate the button with its listener, an instance of SimpleButtonListener. To do this, we can add the following line:

```java
button.addActionListener(new SimpleButtonListener(...));
```

where the arguments to the constructor are the Counter and the JLabel.

• Complete the modification of CounterGUI, compile, and test.

Using an anonymous class to listen to the button:

As we did with the window listener in Show, we can use an anonymous class instance to listen to the button.

• Make a new version of CounterGUI in which you use anonymous class instance to listen to the JButton.

Post-lab:

Submit the following, as directed by your lab instructor:

• a listing of the SimpleButtonListener and the modified CounterGUI;
• a listing of the new version of CounterGUI using an anonymous class.