

Graphical User Interface (an Introduction)

(CISS-110 - William Jojo)

Overview

The first time we used the GUI was back in the *Objects, Input & Output* section. Recall that the `javax.swing` package was used to gain access to the `JOptionPane` class. This class allowed us to use the `showInputDialog()` method for prompted input returned in the form of a string and the `showMessageDialog()` method for displaying output.

We will now expand our knowledge of the GUI with the use of `JFrame`, `JLabel`, `JTextField` and `JButton`. These new objects will be presented in complete programs to demonstrate their use. In addition, the topics of event handling and inheritance will be introduced.

Also to be introduced is the `ActionListener` interface. This will be used to associate work to be performed to a particular event such as a button-press.

Windows and JFrame

Let us begin with a window. The window itself can be thought of as an empty canvas. Nothing exists within the window until we place it there – except a title bar along the top and some basic window controls such as minimize, maximize and close. In fact, many aspects of the window, in general, are not defined until we explicitly set values to manage, adjust or even make visible some particular window.

The `JFrame` class is what we use to create our windows. Some basic methods of `JFrame` are described in *Table 1*. Like `JOptionPane`, the `JFrame` class is part of the `javax.swing` package and is derived from several classes within the Abstract Window Toolkit (AWT). That is to say that `JOptionPane` and `JFrame` inherit methods from classes within the AWT. We can import these classes from the `java.awt` package.

One way to have an application create a window is to declare an object of type `JFrame`. This is demonstrated in *Example 1*.

JFrame methods and description	
public JFrame()	Constructor for JFrame that does not set a value for the title on the newly created window.
public JFrame(String title)	Constructor for JFrame that sets the window title to title on the newly created window.
public void setSize(int width, int height)	Method to set the size of the windows to width pixels wide by height pixels high. [inherited from java.awt.Component class]
public void setTitle(String title)	Method to set the title of a window to title . [inherited from java.awt.Frame class]
public void setVisible(boolean visible)	Method to make a window visible or hidden based on the boolean value of visible . [inherited from the java.awt.Component class]
public void setDefaultCloseOperation(int operation)	Method to set the action to be taken when the user closes the window using the x button. This values for operation are EXIT_ON_CLOSE, HIDE_ON_CLOSE, DISPOSE_ON_CLOSE and DO_NOTHING_ON_CLOSE. The value EXIT_ON_CLOSE is defined within the JFrame class and all others are in the javax.swing.WindowConstants class.
public void addWindowListener(WindowEvent e)	Method to register a windows listener object to a JFrame. [inherited from java.awt.Window class]

Table 1: Some basic methods of the JFrame class.

The first thing to note about the code in *Example 1* is that we have imported the `java.swing` package. A `JFrame` object has been declared as `win` and has been instantiated. All of the methods of the `JFrame` class are called on the `JFrame` instance named `win`. The `EXIT_ON_CLOSE` named constant is defined within the `JFrame` class.

Note that the window title was set as a result of passing a string to the `JFrame` constructor. If we needed to change the title, we could simply call the `setTitle()` method and pass along a new title as a string.

It is essential that we set a window size, the default close operation and make the window visible to the user. The steps are done with the methods `setSize()`, `setDefaultCloseOperation()` and `setVisible()` respectively.

Now although this particular program does work, it doesn't approach the method of programming from an object oriented standpoint. What we need is to alter our thinking slightly and not think of our program as creating a window, but rather that our program *is* the window. *Example 2* shows what we are trying to accomplish.

```

import javax.swing.*;

public class window1
{
    public static void main(String[] args)
    {
        JFrame win = new JFrame("New Window");

        win.setSize(300,200);
        win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        win.setVisible(true);
    }
}

```

Example 1: Sample program creating window object with a reference variable.

```

import javax.swing.*;

public class window2 extends JFrame
{
    // Constructor for window2 class
    public window2()
    {
        setTitle("New Window");
        setSize(300,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args)
    {
        window2 win = new window2();
    }
}

```

Example 2: Sample program that extends JFrame demonstrating inheritance.

Before we describe the details of *Example 2*, know that both of the prior examples produce the windows shown in *Figure 1*.

Take a close look at *Example 2*. By using the `extends` reserved word, we are telling the Java compiler that we are extending the `JFrame` class. This is the first step to understanding *inheritance*.

The act of extending an existing class means that we are *inheriting* all of the capabilities of the class being extended. The class being extended is known as the *superclass* and the class doing the extending is the *subclass*. Therefore, `JFrame` is our superclass and `window2` is the subclass. In addition `window2` inherits all of the definitions of the superclass including methods and named constants.



Figure 1: New window created by programs in Examples 1 and 2.

A review of the code in *Example 2* shows some important details:

- The `main()` method is not doing all of the window work it did in *Example 1*. Instead the main method is invoking the `window2()` constructor by instantiating a `window2` object named `win`.
- The `window2()` constructor is performing all the work once done by `main()`. Recall that constructors are special methods that are automatically called when we instantiate objects.
- Note that all of the set methods are called as if they were static. This is because we have direct access to the `JFrame` methods because of inheritance.
- We also have direct access to `EXIT_ON_CLOSE`, again due to inheritance.

It may seem a bit strange at first to picture writing a program this way. Remember that we are creating classes and classes inherit other classes. One goal is to get into the habit of *reusing* classes rather than reinventing them. Also keep in mind that if we are creating a window, rather than treating the object as a disjoint appendage of some `main()` method, it is far better to treat the window as a complete entity and enjoy the benefits of inheritance and simplified programming.

Adding Objects to JFrame

By default we cannot add content after the window is created. This is because we have not created a pathway into the window. Within the frame of the window is the *content pane*. The content pane is where we would add labels, fields and buttons. This is where all of the objects of a window get together to get the required work done.

The first thing we must do is gain access to the content pane. We do this by creating a reference variable of type `Container`. The `Container` class is available in the `java.awt` package and will need to be imported. After that, we will call the `getContentPane()` method of the `JFrame` class. One line of code can do this as demonstrated here:

```
Container pane = getContentPane();
```

This defines the `pane` reference variable of type `Container` and assigns the object returned by `getContentPane()` to `pane`. Some methods available to the `Container` class are listed in *Table 2*.

Container class methods
public void add(Object obj) Method to add an object specified by <code>obj</code> to the pane.
public void setLayout(Object obj) Method to set a layout specified by <code>obj</code> for a pane.

Table 2: Some methods of the Container class.

The content pane allows objects to be placed in the window based on a *layout*. The default layout of the content pane is the `Border` layout. There are many layout managers. We will discuss the `BorderLayout`, `GridLayout` and `FlowLayout` layouts.

Layouts (an introduction)

With the window created and access to the content pane established, we can begin to think about how we want to layout the window contents. This step is important as we need to know where our window components will be placed and whether their placement makes sense when used by us or an end user. As a result the selection of a layout manager is somewhat crucial, but perhaps not at first. There is nothing wrong with selecting a `FlowLayout` to just drop objects onto for testing purposes and then selecting perhaps a `Border`- or `GridLayout` once development is underway.

The simplest layout to work with is the `GridLayout`. This layout takes the content pane and divides up the pixel space into even chunks based on the number of rows and columns specified. If we select a `GridLayout` of 3x5, then we would have 3 rows of 5 columns of equal size areas to place objects. It is essential to know the dimensions of the window when starting to imagine the layout. For example, a 400 pixel wide by 200 pixel high window loses a bit for the frame and title bar and what is left is divided up among the 15 grid locations.

Example 3 shows a 3x2 `GridLayout` using some labels. The resulting window is shown in *Figure 2*.

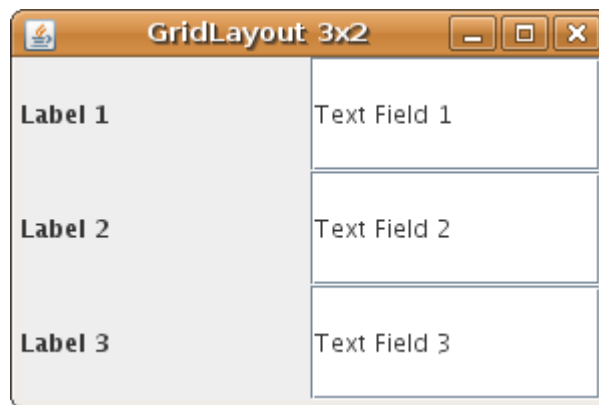


Figure 2: Window using grid layout created by program in Example 3.

Labels and Text Fields (JLabel, JTextField)

Fortunately, *Example 3* has set the stage for the next part of our discussion which involves the use of labels and text fields. As is shown in the code for *Example 3*, we have added three labels and three text fields to the content pane. The objects have been arranged from left to right and from top to bottom as each object was added to the content pane. This is the expected behavior when using the `GridLayout`.

Each `JLabel` and `JTextField` was explicitly declared with `private instance` variables of the class `grid` and then objects were instantiated and assigned to each variable. These particular statements in *Example 3* have been highlighted in bold text. The remainder of the program is reminiscent of *Example 2*.

A string value was specified for each `JLabel` at the time of instantiation. The particular `JTextField` constructor used specified an initializer for the text field, but not a size.

Tables 3 and 4 show some methods available when using `JLabel` and `JTextField`.

JLabel class methods
public JLabel(String label) Constructor for <code>JLabel</code> that assigns left-aligned text specified in <code>label</code> .
public JLabel(String label, int align) Constructor for <code>JLabel</code> that assigns text specified in <code>label</code> and whose alignment is specified in <code>align</code> . The values of <code>align</code> are <code>SwingConstants.LEFT</code> , <code>SwingConstants.RIGHT</code> or <code>SwingConstants.CENTER</code> .
public JLabel(String label, Icon icon, int align) Constructor for <code>JLabel</code> that assigns text and an icon to the label. The icon appears to the left of the text. The text is aligned based on the value of <code>align</code> .
public JLabel(Icon icon) Constructor for <code>JLabel</code> that creates a label with an icon.

Table 3: Some methods of the JLabel class.

```

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

public class grid extends JFrame
{
    private JLabel l1, l2, l3;
    private JTextField tf1, tf2, tf3;

    // Constructor for grid
    public grid()
    {
        Container pane = getContentPane();

        pane.setLayout(new GridLayout(3,2));

        l1 = new JLabel("Label 1");
        l2 = new JLabel("Label 2");
        l3 = new JLabel("Label 3");
        tf1 = new JTextField("Text Field 1");
        tf2 = new JTextField("Text Field 2");
        tf3 = new JTextField("Text Field 3");

        pane.add(l1);
        pane.add(tf1);
        pane.add(l2);
        pane.add(tf2);
        pane.add(l3);
        pane.add(tf3);

        setTitle("GridLayout 3x2");
        setSize(300,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args)
    {
        grid win = new grid();
    }
}

```

Example 3: Sample program to add labels to window using GridLayout.

JTextField class methods.	
public JTextField(int columns)	Constructor to set the text field size to <code>columns</code> .
public JTextField(String s)	Constructor to initialize the value of the text field to <code>s</code> .
public JTextField(String s, int columns)	Constructor to initialize the value of the text field to <code>s</code> and set the size to <code>columns</code> .
public void setText(String s)	Method to change the value of the text field to <code>s</code> .
public String getText()	Method to access the current value of the text field as a <code>String</code> .
public void setEditable(boolean edit)	Method to control the editable nature of the text field. If <code>edit</code> is false, the text field is protected.
public void addActionListener(ActionListener obj)	Method to register a listener object specified by <code>obj</code> to the text field.

Table 4: Some methods of the *JTextField* class.

Buttons (JButton)

Buttons are a natural progression from labels and text fields. We are often required to fill out forms and submit the contents to some remote data receptacle. Buttons are usually available to perform the submit or perhaps to clear a particular form. *Example 4* demonstrates using two labels, two text fields and two button with event handlers to move some data from one field to another and to clear the fields. The use of *JButton* completes this form quite nicely.

The Reset button is intended to clear both text fields and the Submit button will copy form the input text field to the display text field. The form created by the program in *Example 4* is shown in *Figure 3*. Although this is a nice layout, there is only one very important thing wrong – there is no action taken for a button press.

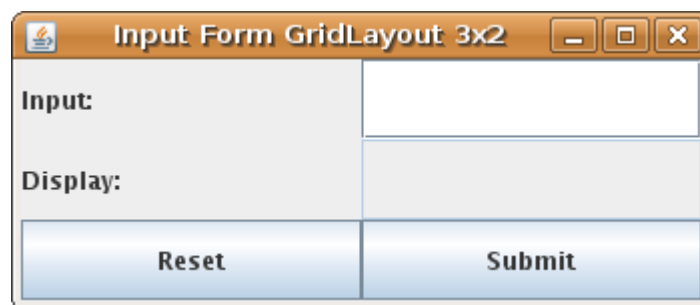


Figure 3: Form created by the program in *Example 4*.

```

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

public class form extends JFrame
{
    private JLabel inputL, displayL;
    private JTextField inputTF, displayTF;
    private JButton resetB, submitB;

    public form()
    {
        Container pane = getContentPane();

        pane.setLayout(new GridLayout(3,2));

        inputL = new JLabel("Input:");
        displayL = new JLabel("Display:");
        inputTF = new JTextField("");
        displayTF = new JTextField("");
        resetB = new JButton("Reset");
        submitB = new JButton("Submit");

        pane.add(inputL);
        pane.add(inputTF);
        pane.add(displayL);
        pane.add(displayTF);
        displayTF.setEditable(false);
        pane.add(resetB);
        pane.add(submitB);

        setTitle("Input Form GridLayout 3x2");
        setSize(350,150);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args)
    {
        form win = new form();
    }
}

```

Example 4: Sample program to create form complete with buttons.

JButton class methods
public JButton(Icon icon) Constructor to set the value of a button to some icon specified by <code>icon</code> .
public JButton(String s) Constructor to set the value of the button to the text in <code>s</code> .
public JButton(String s, Icon icon) Constructor that assigns <code>s</code> and <code>icon</code> to a button. The icon appears to the left of the text.
public void setText(String s) Method to assign text to a button specified by <code>s</code> .
public String getText() Method to read the current text of a button and return it as a <code>String</code> .
public void addActionListener(ActionListener obj) Method to register a listener object to the button specified by <code>obj</code> .

Table 5: Some methods available to the `JButton` class.

Event Handling

In order for the Reset button to clear the two text fields and in order for the Submit button to copy the text from input text field to the display text field, we need a mechanism to notify us that the button has been pressed.

Event handling allows us to assign special listener methods to objects that can be used to cause work to be done when an event is triggered by a user. Now these types of events can be anything from text fields being filled-in to a mouse passing over an object in a given window. We're going to simply get our two buttons to perform work when pressed.

Let us start with creating a handler for an event. Clicking on a button creates an *action event*. This action event sends a message to an *action listener*. Our handler will be our action listener. These listeners will be in the form of additional classes that we will create. A class will be created to handle the Reset button and another class to handle the Submit button. Consider the following code:

```
private class ResetButtonHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        inputTF.setText("");
        displayTF.setText("");
    }
}
```

```
private class SubmitButtonHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        displayTF.setText(inputTF.getText());
    }
}
```

Now we have a couple of classes for button handlers namely `ResetButtonHandler` and `SubmitButtonHandler`. Each of these classes *implements* the `ActionListener` interface which provides an empty method `actionPerformed()`. This interface is a kind of class that only defines method headings or *prototypes*. As a result, you cannot instantiate an interface, rather we create classes that implement the interface. It is now our responsibility to provide the method body to complete the picture for the compiler. The `ActionListener` interface is shown below:

```
public interface ActionListener
{
    public void actionPerformed(ActionEvent e);
}
```

As you can see from the `ActionListener` interface definition, the `actionPerformed()` method has no body – it is open ended. We need it to be open ended since we will be providing the work to be done for each action event for which we intend to provide a handler.

To use the `actionListener` interface we must import the `java.awt.event` package.

The `ResetButtonHandler` and `SubmitButtonHandler` classes are implementing the `ActionListener` interface and they each have an `actionPerformed()` method that defines the work to be done. Specifically the `ResetButtonHandler` will call the `setText()` method to clear the two text fields and the `SubmitButtonHandler` will call `setText()` of the display textfield using the data retrieved from `getText()` of the input text field.

All that is left to do is to *register* the action listeners with their specific buttons. The following declarations create reference variables to hold the objects that represent our handlers. They will allow us to call the `addActionListener()` method of each button.

```
private ResetButtonHandler rbHandler;
private SubmitButtonHandler sbHandler;
```

```

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

public class formevent extends JFrame
{
    private JLabel inputL, displayL;
    private JTextField inputTF, displayTF;
    private JButton resetB, submitB;
    private ResetButtonHandler rbHandler;
    private SubmitButtonHandler sbHandler;

    public formevent()
    {
        Container pane = getContentPane();

        pane.setLayout(new GridLayout(3,2));

        inputL = new JLabel("Input:");
        displayL = new JLabel("Display:");
        inputTF = new JTextField("");
        displayTF = new JTextField("");
        resetB = new JButton("Reset");
        submitB = new JButton("Submit");

        rbHandler = new ResetButtonHandler();
        resetB.addActionListener(rbHandler);

        sbHandler = new SubmitButtonHandler();
        submitB.addActionListener(sbHandler);

        pane.add(inputL);
        pane.add(inputTF);
        pane.add(displayL);
        pane.add(displayTF);
        displayTF.setEditable(false);
        pane.add(resetB);
        pane.add(submitB);
    }
}

```

Example 5: Final program for the event driven form.

```

        setTitle("Input Form GridLayout 3x2");
        setSize(350,150);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    private class ResetButtonHandler implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            inputTF.setText("");
            displayTF.setText("");
        }
    }

    private class SubmitButtonHandler implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            displayTF.setText(inputTF.getText());
        }
    }

    public static void main(String[] args)
    {
        formevent win = new formevent();
    }
}

```

Example 5: Continued - Final program for the event driven form.

Now we can instantiate the handlers and add them as action listeners for each button.

```

rbHandler = new ResetButtonHandler();
resetB.addActionListener(rbHandler);

sbHandler = new SubmitButtonHandler();
submitB.addActionListener(sbHandler);

```

It can be quite tedious to set up a GUI that has many labels, field and buttons. The real tedium comes in creating action listeners for the events that will take place and making sure you have considered all the possibilities as a result of interacting with the user through event.

A final version of our program is presented in Example 5.

Additional Layouts

We have already seen the `GridLayout` and as we mentioned earlier, there are some other layout managers available. This section will touch on the `FlowLayout` and `BorderLayout` as they are of some interest in designing a GUI. Keep in mind there are many other layout managers and some are more difficult to use or require expert knowledge to set up effectively.

Recall that the default layout of the content pane is the `BorderLayout`. The `BorderLayout` maps the content pane into the four compass points north, south, east and west. In addition, there is a center. *Example 6* shows a program with the `BorderLayout` and buttons in each section. The buttons do not function, but are simply used to fill out the region to accentuate the zones of the `BorderLayout`. *Figure 4* shows the window that results from the program in *Example 6*.

As you can see from the picture in *Figure 4*, the objects placed in the `BorderLayout` occupy the complete section. Note that in the code for *Example 6*, the buttons are placed in their respective zones by use of named constants `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER` defined in the `BorderLayout` class. If the window size is changed by using the mouse, the buttons will adjust to the changes and will likely cause the text in the button to shrink or otherwise disappear.

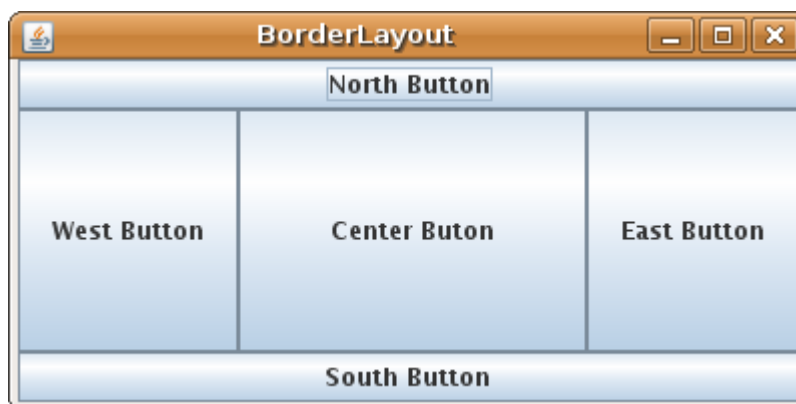


Figure 4: Buttons in the BorderLayout from Example 6.

```

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

public class border extends JFrame
{
    private JButton nb, sb, eb, wb, cb;

    public border()
    {
        Container pane = getContentPane();

        pane.setLayout(new BorderLayout());

        nb = new JButton("North Button");
        sb = new JButton("South Button");
        eb = new JButton("East Button");
        wb = new JButton("West Button");
        cb = new JButton("Center Buton");

        pane.add(nb, BorderLayout.NORTH);
        pane.add(sb, BorderLayout.SOUTH);
        pane.add(eb, BorderLayout.EAST);
        pane.add(wb, BorderLayout.WEST);
        pane.add(cb, BorderLayout.CENTER);

        setTitle("BorderLayout");
        setSize(400,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args)
    {
        border win = new border();
    }
}

```

Example 6: Sample program showing the use of the BorderLayout.

```

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

public class flow extends JFrame
{
    private JButton b1, b2, b3;
    private JLabel l1;
    private JTextField tf1;

    public flow()
    {
        Container pane = getContentPane();

        pane.setLayout(new FlowLayout());

        b1 = new JButton("Button 1");
        b2 = new JButton("Button 2");
        b3 = new JButton("Button 3");
        l1 = new JLabel("Label 1");
        tf1 = new JTextField("Text Field 1 - 30 Columns", 30);

        pane.add(l1);
        pane.add(tf1);
        pane.add(b1);
        pane.add(b2);
        pane.add(b3);

        setTitle("FlowLayout");
        setSize(400,200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String[] args)
    {
        flow win = new flow();
    }
}

```

Example 7: Sample program demonstrating the FlowLayout.

The `FlowLayout` allows you to place objects in the content pane without placement restrictions. The objects are placed in the content pane from left to right in the top line (or row) and when no more objects can fit, the objects will be placed on the next line. What is interesting about the `FlowLayout` is how the objects are repositioned as the window is adjusted.

The program in *Example 7* uses the `FlowLayout` to arrange a label, text field and three buttons. *Figure 5* shows the initial window view of the window based on the size and the order of object placement. Using the mouse, resize the window and see how the objects reposition themselves. A very narrow window will cause the objects to be stacked and a very wide window will make them appear on a single line.

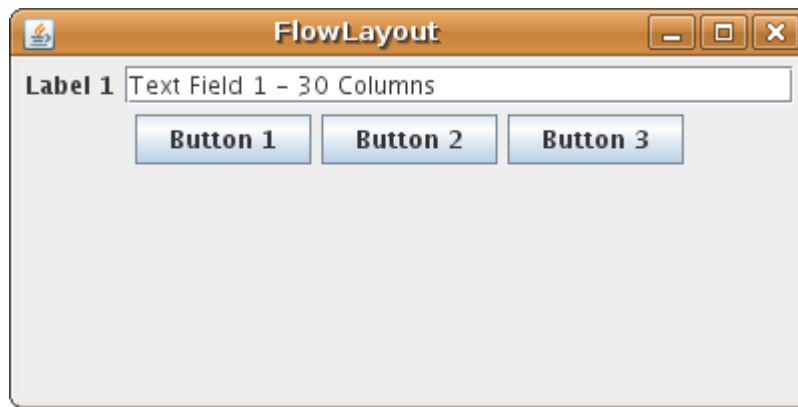


Figure 5: Initial object position in the FlowLayout based on the program from Example 7.