Intro to Swing

Java's first package to create Graphical User Interfaces (GUIs) was the AWT, or Abstract Window Toolkit. Swing is an improved version of the AWT and is based on objects and event-driven programming. We will start by describing events and how to handle them.

Events and Listeners

An **event** is an object that is a signal to another object known as a **listener** that does something, or **handles**, the event. Sending the event is known as **firing** the event. For example, when a button is clicked, the button fires the clicked event and a method may listen for this event and do something in response.

Exception handling is one form of event-driven programming where the listener is the catch block for the event.

In a Swing GUI, our events will be things like a mouse click, mouse drag, keypress, etc. A listener object will have methods that specify what happens when the events are received. These methods are called **event handlers**.

The main difference between event-driven programming and the programming you have done so far is the sequence that things happen. Previously everything started in main and ran in sequential order. In event-driven programming, you create listeners for events, and the events determine the order. We have seen some of these already in the graphics programming section. We end up writing methods that we don't actually call ourselves, but might be called when some event occurs.

Consider the following program, which pops up a window and lets us close it:

```
import javax.swing.JFrame;
/**
 *
 * @author Kenrick
 */
public class JavaApplication18 extends JFrame
{
    public JavaApplication18()
    {
        super();
        setSize(810,640);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        JavaApplication18 app = new JavaApplication18();
    }
}
```

Let’s give an example of listening firing off an event when the user clicks a button. To add a button the window we can use the JButton object:

```
import javax.swing.JFrame;
import javax.swing.JButton;
/**
 *
 * @author Kenrick
 */
public class JavaApplication18 extends JFrame
{
    public JavaApplication18()
    {
        super();
        setSize(810,640);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JButton clickme = new JButton("Click Me");
        this.add(clickme);

        setVisible(true);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        JavaApplication18 app = new JavaApplication18();
    }
}
```

This creates a button that takes up the entire JFrame (we’ll see shortly how to control this better) that the user can click, but at this point nothing happens because there is no listener to handle the button click event.

We can make a listener by creating a class that implements the **ActionListener** interface. This interface expects us to have an **actionPerformed** method:

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println(e);
    }
}
```

```
    public JavaApplication18()
    {
        super();
        setSize(810,640);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JButton clickme = new JButton("Click Me");
        clickme.addActionListener(new ButtonListener());
        this.add(clickme);

        setVisible(true);
    }
```

We now get output every time the button is clicked! We could of course do something else inside the actionPerformed method.

Listeners as Inner Classes

If the listener is only used inside the JFrame, then it makes sense to define the listener as an inner class. In this case the program would now look like this:

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class JavaApplication18 extends JFrame
{
    private class ButtonListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            System.out.println(e);
        }
    }

    public JavaApplication18()
    {
        super();
        setSize(810,640);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        JButton clickme = new JButton("Click Me");
        clickme.addActionListener(new ButtonListener());
        this.add(clickme);

        setVisible(true);
    }
```

```
        /**
         * @param args the command line arguments
         */
        public static void main(String[] args) {
            JavaApplication18 app = new JavaApplication18();
        }
    }
```

Anonymous Inner Class

If we are only going to use the inner class in one place, such as the button click event, then we can create an anonymous inner class right in the spot where it is used! This format looks a little weird at first, but in the end it can make the classes easier to read because the class is defined where it is referenced.

```
    public class JavaApplication18 extends JFrame
    {
        public JavaApplication18()
        {
            super();
            setSize(810,640);
            setDefaultCloseOperation(EXIT_ON_CLOSE);

            JButton clickme = new JButton("Click Me");
            clickme.addActionListener(new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        System.out.println("You clicked me");
                    }
                }
            );
            this.add(clickme);

            setVisible(true);
        }
```

Later we will see a method using lambda expressions that is an even shorter way of doing this.

Setting Color

To set the color of a Swing element, we can use the method setBackground(color) or setForground(color). You will likely need to import java.awt.Color:

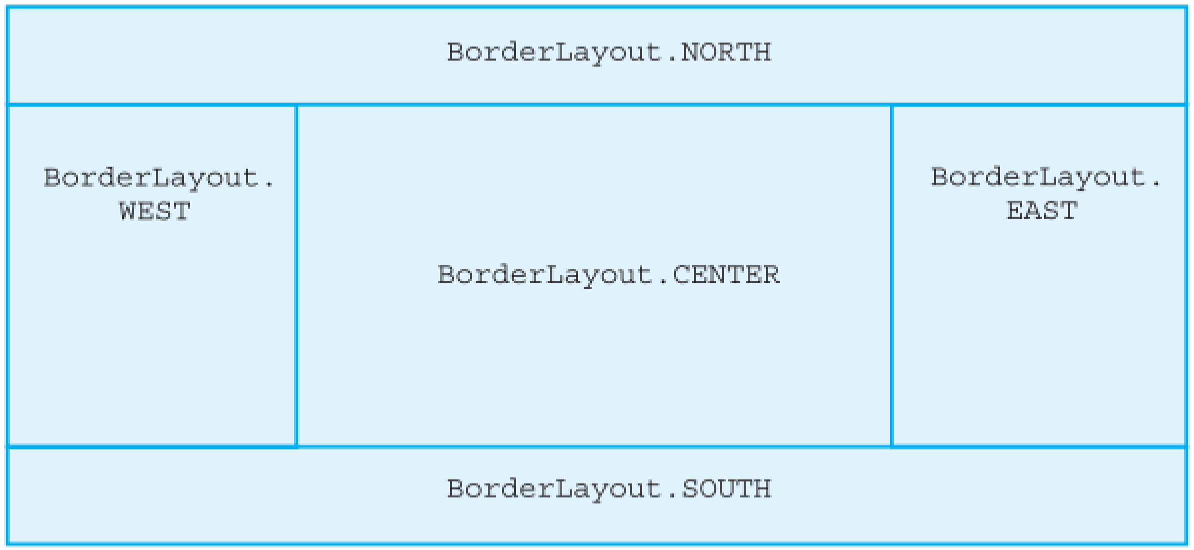
```
            clickme.setBackground(Color.red);
            clickme.setForeground(new Color(0,255,255));
```

Border Layout Manager

If you do not specify a layout, then Java will use a **BorderLayout** by default. A layout manager specifies how items should be added to a JFrame. To specify the layout type we add to the constructor:

setLayout(new BorderLayout());

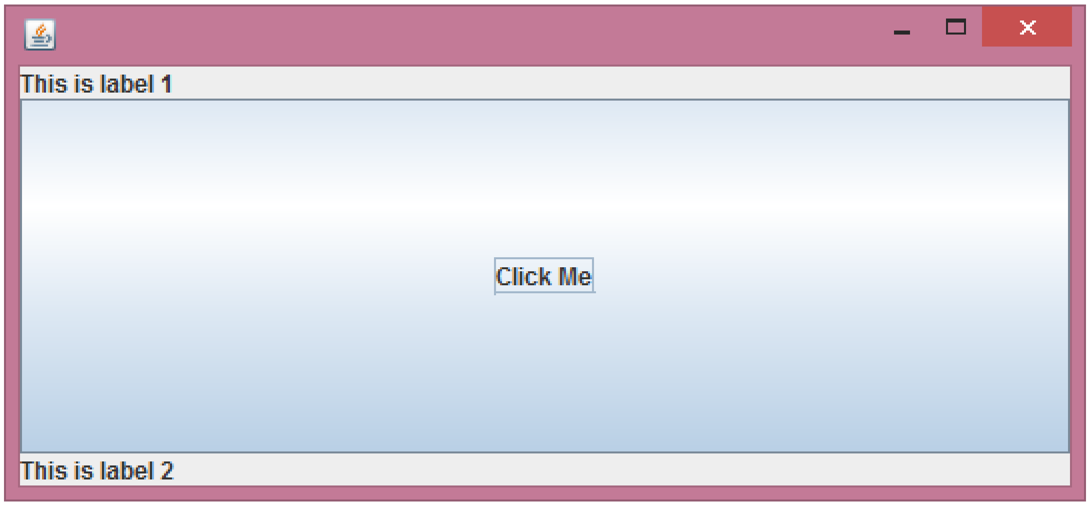
We need to import java.awt.BorderLayout. In a Border layout, we can add items to the center, north, west, east, or south of the frame:



Here is an example where we add a button the middle, and labels to the north and south. A Label is just a piece of text.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;
import javax.swing.JLabel;
import java.awt.BorderLayout;

public class JavaApplication18 extends JFrame
{
    public JavaApplication18()
    {
        super();
        setSize(810,640);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new BorderLayout());

        JButton clickme = new JButton("Click Me");
        clickme.addActionListener(new ActionListener()
            {
```

```
                        public void actionPerformed(ActionEvent e)
                        {
                            System.out.println("You clicked me");
                        }
                    }
                );
                this.add(clickme, BorderLayout.CENTER);

                JLabel label1 = new JLabel("This is label 1");
                this.add(label1, BorderLayout.NORTH);

                JLabel label2 = new JLabel("This is label 2");
                this.add(label2, BorderLayout.SOUTH);

                setVisible(true);
            }
            /**
             * @param args the command line arguments
             */
            public static void main(String[] args) {
                JavaApplication18 app = new JavaApplication18();
            }
        }
```



Note that the window resizes the center based on the size of the labels. If we add a component to the east or west then the button will shrink accordingly:

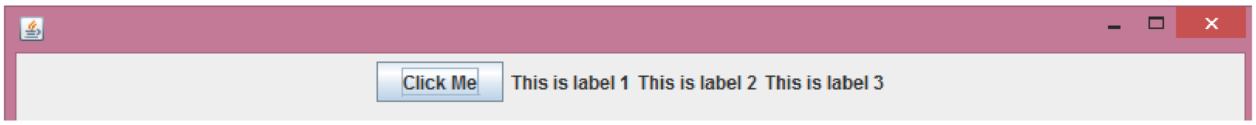
```
                JLabel label3 = new JLabel("This is label 3");
                this.add(label3, BorderLayout.WEST);
```

If you try to put two components into the same spot then you will only get the last component. However, we will see shortly that we can add a panel as a component, then put multiple items into the panel.

Flow Layout Manager

The flow layout manager is the simplest layout manager. It arranges components one after the other, left to right, in the order you add them. This can be useful for testing purposes as it is quick and easy to implement.

```
        public JavaApplication18()
          {
               super();
               setSize(810,640);
               setDefaultCloseOperation(EXIT_ON_CLOSE);
               setLayout(new FlowLayout());

               JButton clickme = new JButton("Click Me");
               clickme.addActionListener(new ActionListener()
                   {
                       public void actionPerformed(ActionEvent e)
                       {
                           System.out.println("You clicked me");
                       }
                   }
               );
               this.add(clickme);

               JLabel label1 = new JLabel("This is label 1");
               this.add(label1);

               JLabel label2 = new JLabel("This is label 2");
               this.add(label2);

               JLabel label3 = new JLabel("This is label 3");
               this.add(label3);

               setVisible(true);
          }
```

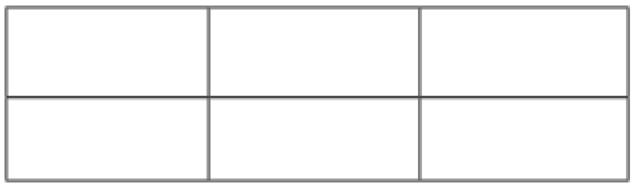


GridLayout Manager

The grid layout manager arranges components in a two-d grid with some number of rows and columns. With a GridLayotu manager, each entry is the same size. For example:

```
            setLayout(new GridLayout(2,3));
```

will specify two rows and three columns within the container:

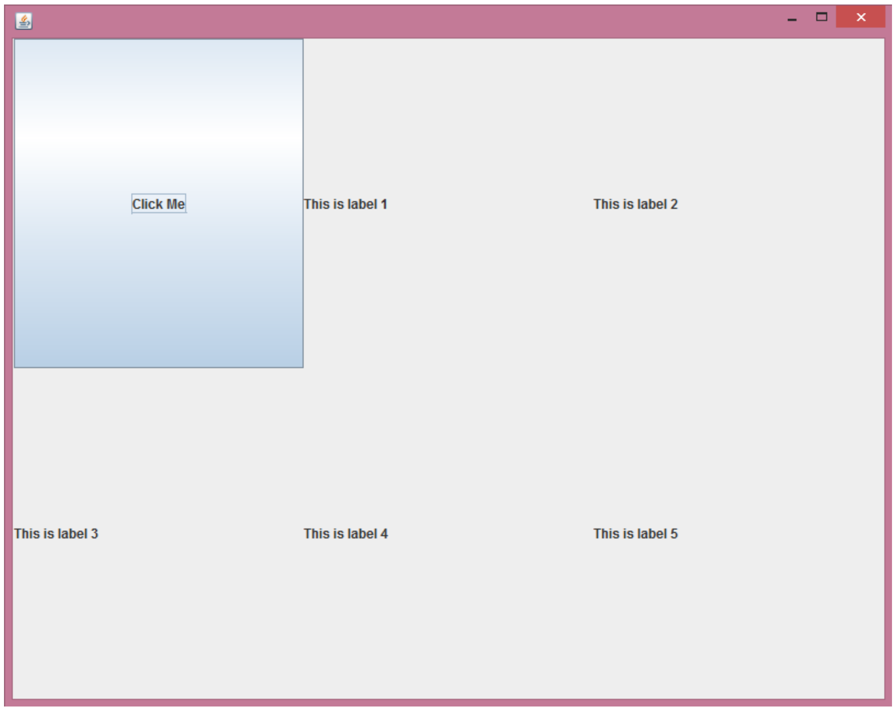


The lines will not be visible unless you do something special to make them seen. Each component is stretched so that it completely fills its grid position.

Note that if you add more items than fit in the grid, then a new column is added to fit the items. If you add fewer items than specified in the gird, then there will be two rows and a reduced number of columns.

When using add, items are placed in the grid left to right, filling the top row, then the second row, and so forth. You are not allowed to skip any grid positions although you could add something that does not show and so gives the illusion of skipping a grid position.

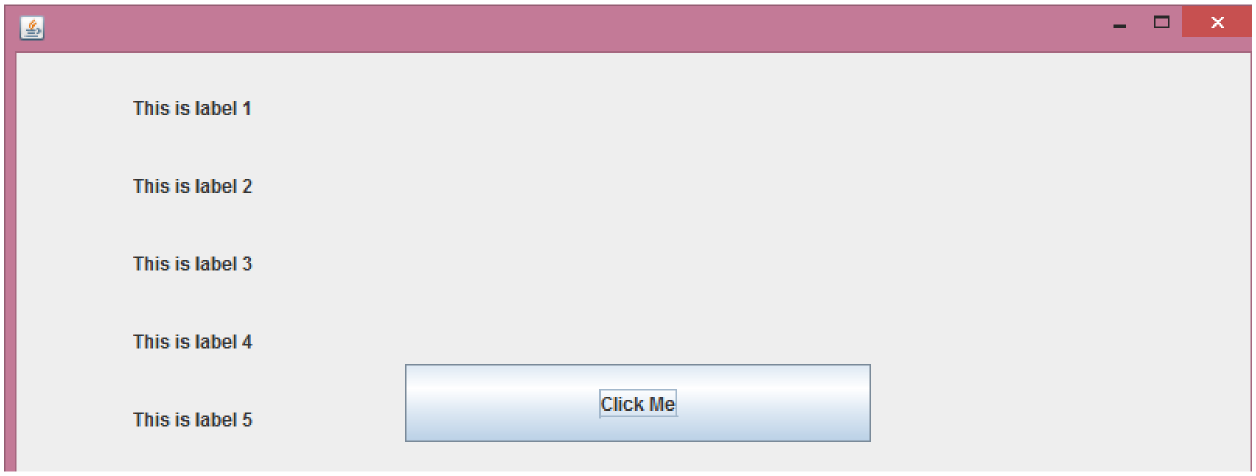
```
public JavaApplication18()
{
    super();
    setSize(810,640);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new GridLayout(2,3));

    JButton clickme = new JButton("Click Me");
    clickme.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                System.out.println("You clicked me");
            }
        }
    );
    this.add(clickme);

    for (int i = 0; i < 5; i++)
    {
        JLabel lbl = new JLabel("This is label " + (i+1));
        this.add(lbl);
    }

    setVisible(true);
}
```



Null Layout

Finally, we can use the null layout. In the null layout, we specify exact coordinates for each component. If we go this route, typically you would use an IDE to graphically build the GUI.

```
super();
setSize(810,640);
setDefaultCloseOperation(EXIT_ON_CLOSE);
setLayout(null);

JButton clickme = new JButton("Click Me");
clickme.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            System.out.println("You clicked me");
        }
    }
);
this.add(clickme);
clickme.setBounds(250, 200, 300, 50);

for (int i = 0; i < 5; i++)
{
    JLabel lbl = new JLabel("This is label " + (i+1));
    this.add(lbl);
    lbl.setBounds(75, 20 + i*50, 200, 30);
}
setVisible(true);
```



Panels

A GUI is often organized hierarchically with containers known as **panels** inside other containers. A panel is an object of the class JPanel, which is a simple container that just groups together objects. A JPanel object is analogous to the curly braces we use to group code. It groups smaller objects like buttons and labels into a larger component (the JPanel). You can then put the JPanel object into a JFrame. Thus, a JPanel is used to subdivide a JFrame (or other container) into different areas.

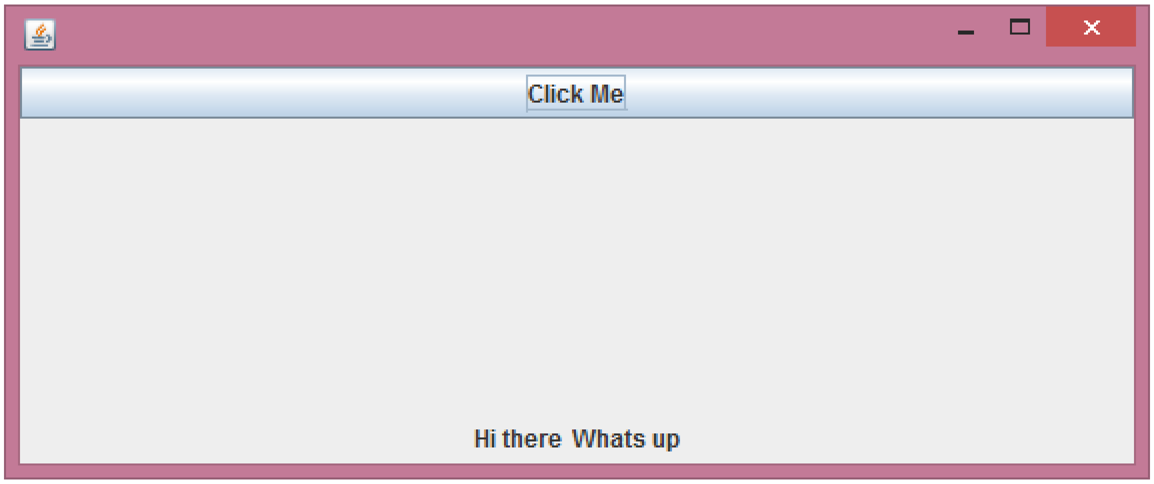
Import a panel with javax.swing.JPanel.

We can apply a layout to a panel.

In the example below we use a BorderLayout with a panel containing two labels in the SOUTH using a flow layout, then button in the NORTH of the border:

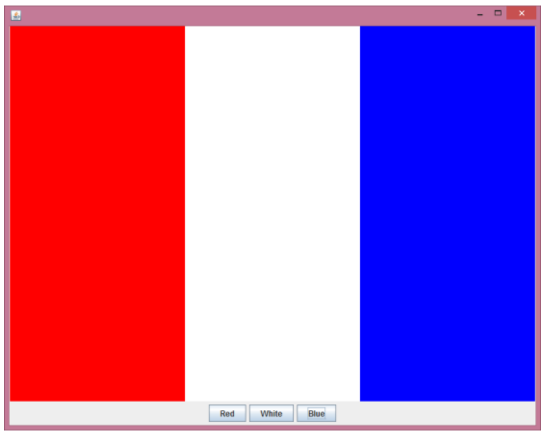
```
public JavaApplication18()
{
    super();
    setSize(810,640);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new BorderLayout());

    JButton clickme = new JButton("Click Me");
    clickme.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                System.out.println("You clicked me");
            }
        }
    );
    this.add(clickme, BorderLayout.NORTH);

    JPanel panel = new JPanel();
```

```
            panel.setLayout(new FlowLayout());
            this.add(panel, BorderLayout.SOUTH);

            JLabel label1 = new JLabel("Hi there");
            panel.add(label1);
            JLabel label2 = new JLabel("Whats up");
            panel.add(label2);

            setVisible(true);
        }
```



Here is an example where we make a panel containing three panels (grid layout) in the center, then a panel in the south containing three buttons. Each button changes the color of one of the panels in the center.

```
        public JavaApplication18()
        {
            super();
            setSize(810,640);
            setDefaultCloseOperation(EXIT_ON_CLOSE);
            setLayout(new BorderLayout());

            // Master panel in the middle
            JPanel middle = new JPanel();
            middle.setLayout(new GridLayout(1,3));
            this.add(middle, BorderLayout.CENTER);

            // Three panels inside the middle panel
            JPanel redPanel = new JPanel();
            JPanel whitePanel = new JPanel();
            JPanel bluePanel = new JPanel();
            middle.add(redPanel);
            middle.add(whitePanel);
            middle.add(bluePanel);
            JPanel bottom = new JPanel();
            bottom.setLayout(new FlowLayout());
            this.add(bottom, BorderLayout.SOUTH);

            JButton redButton = new JButton("Red");
            redButton.addActionListener(new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        redPanel.setBackground(Color.red);
                    }
                }
            );
            bottom.add(redButton);
            JButton whiteButton = new JButton("White");
            whiteButton.addActionListener(new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        whitePanel.setBackground(Color.white);
                    }
                }
            );
            bottom.add(whiteButton);
            JButton blueButton = new JButton("Blue");
            blueButton.addActionListener(new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        bluePanel.setBackground(Color.blue);
                    }
                }
            );
            bottom.add(blueButton);

            setVisible(true);
        }
```

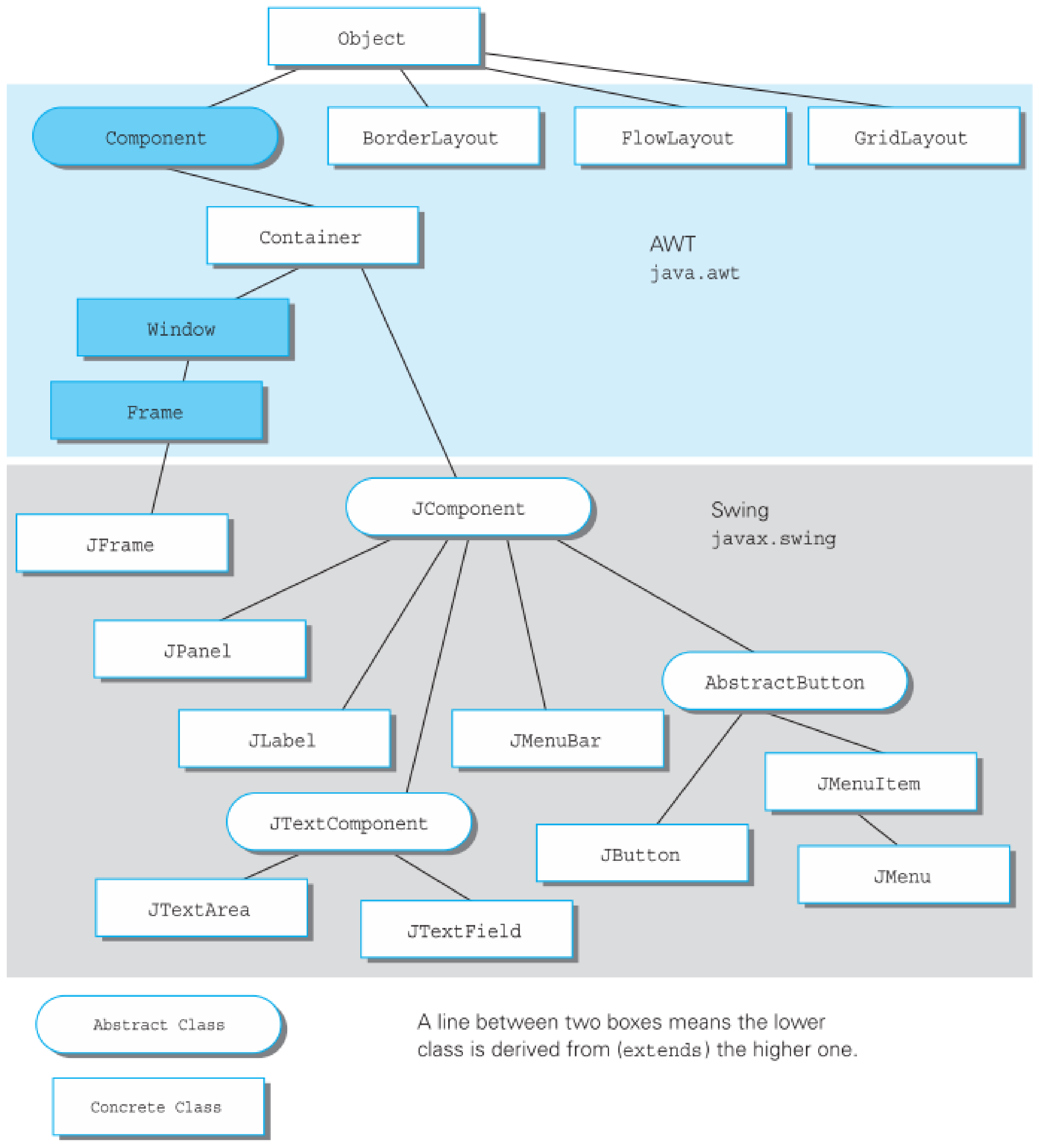


There is some repetition creating the button click event. If we like, we could share the same listener for all three buttons. The event argument sends the text of the button clicked, so we could distinguish which button was clicked. This method is not necessarily better, it will depend on your application.

If using this technique, the JFrame class itself typically implements ActionListener. We add the actionPerformed method to the same class and move the JPanels to private class variables so we can access them from the action listener.

```
public class JavaApplication18 extends JFrame implements ActionListener
{
    private JPanel redPanel;
    private JPanel whitePanel;
    private JPanel bluePanel;

    @Override
    public void actionPerformed(ActionEvent e)
    {
        String cmd = e.getActionCommand();
        if (cmd.equals("Red"))
        {
            redPanel.setBackground(Color.red);
        }
        else if (cmd.equals("White"))
        {
            whitePanel.setBackground(Color.white);
        }
        else if (cmd.equals("Blue"))
        {
            bluePanel.setBackground(Color.blue);
        }
    }

    public JavaApplication18()
    {
        super();
        setSize(810,640);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLayout(new BorderLayout());

        // Master panel in the middle
        JPanel middle = new JPanel();
        middle.setLayout(new GridLayout(1,3));
        this.add(middle, BorderLayout.CENTER);

        // Three panels inside the middle panel
        redPanel = new JPanel();
        whitePanel = new JPanel();
        bluePanel = new JPanel();
        middle.add(redPanel);
        middle.add(whitePanel);
        middle.add(bluePanel);

        JPanel bottom = new JPanel();
        bottom.setLayout(new FlowLayout());
        this.add(bottom, BorderLayout.SOUTH);

        JButton redButton = new JButton("Red");
        redButton.addActionListener(this);
        bottom.add(redButton);
        JButton whiteButton = new JButton("White");
        whiteButton.addActionListener(this);
        bottom.add(whiteButton);
        JButton blueButton = new JButton("Blue");
        blueButton.addActionListener(this);
        bottom.add(blueButton);

        setVisible(true);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        JavaApplication18 app = new JavaApplication18();
    }
}
```

The following diagram shows the relationship between AWT and Swing for some common classes.


Object
Component
BorderLayout
FlowLayout
GridLayout
Container
AWT
java.awt
Window
Frame
JComponent
Swing
javax.swing
JFrame
JPanel
AbstractButton
JLabel
JMenuBar
JMenuItem
JTextComponent
JButton
JMenu
JTextArea
JTextField
Abstract Class
Concrete Class
A line between two boxes means the lower class is derived from (extends) the higher one.