

About Swing

Swing library is an official Java GUI toolkit released by Sun Microsystems. It is used to create Graphical user interfaces with Java.

The Swing API has 18 public packages:

- javax.accessibility
- javax.swing
- javax.swing.border
- javax.swing.colorchooser
- javax.swing.event
- javax.swing.filechooser
- javax.swing.plaf
- javax.swing.plaf.basic
- javax.swing.plaf.metal
- javax.swing.plaf.multi
- javax.swing.plaf.synth
- javax.swing.table
- javax.swing.text
- javax.swing.text.html
- javax.swing.text.html.parser
- javax.swing.text.rtf
- javax.swing.tree
- javax.swing.undo

Swing is an advanced GUI toolkit. It has a rich set of widgets. From basic widgets like buttons, labels, scrollbars to advanced widgets like trees and tables. Swing itself is written in Java.

Swing is a part of JFC, Java Foundation Classes. It is a collection of packages for creating full featured desktop applications. JFC consists of AWT, Swing, Accessibility, Java 2D, and Drag and Drop. Swing was released in 1997 with JDK 1.2. It is a mature toolkit.

Java Swing first programs

In this chapter, we will program our first programs in Swing toolkit. The examples are going to be very simple. We will cover some basic functionality.

First example

In our first example, we will show a basic window on the screen.

```
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class SimpleExample extends JFrame {
```

```

public SimpleExample() {

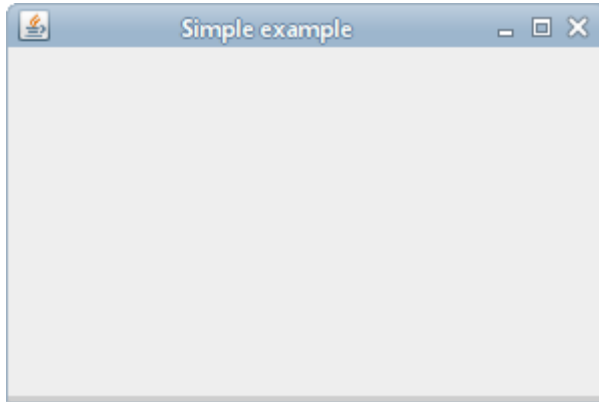
    setTitle("Simple example");
    setSize(300, 200);
    setLocationRelativeTo(null);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public static void main(String[] args) {

    SimpleExample ex = new SimpleExample();
    ex.setVisible(true);
}
}

```

This code will result in this:



Let's explain some of the lines in the code:

```
public class Example extends JFrame {
```

The Example class inherits from the `JFrame` widget. `JFrame` is a toplevel container. In the container, we put other widgets. Think of it as the window you are creating. By having our class inherit from `JFrame`, we get all the methods `JFrame` has built in.

```
    setTitle("Simple example");
```

Here we set the title of the window using the `setTitle()` method.

```
    setSize(300, 200);
```

This code will resize the window to be 300px wide and 200px tall.

```
    setLocationRelativeTo(null);
```

This line will center the window on the screen.

```
    setDefaultCloseOperation(EXIT_ON_CLOSE);
```

This method will close the window, if we click on the close button of the titlebar. By default nothing happens.

```
SimpleExample ex = new SimpleExample();
ex.setVisible(true);
```

We create an instance of our code example and make it visible on the screen.

Note: if we really wanted to do this correctly, you'd do the following:

```
SwingUtilities.invokeLater(new Runnable() {
    @Override
    public void run() {
        SimpleExample ex = new SimpleExample();
        ex.setVisible(true);
    }
});
```

This is to prevent your GUI from hanging. It's a safer way to run your GUI.

Quit button

Now we'll add a quit button. When we click on the button, the application terminates.

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

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class QuitButtonExample extends JFrame { // must extend JFrame

    public QuitButtonExample() {

        initUI(); // we do this so that all the UI stuff is together
    }

    private void initUI() {

        JPanel panel = new JPanel(); //create a JPanel - panels hold things
        getContentPane().add(panel); // add the panel to the JFrame

        panel.setLayout(null); /** see below

        JButton quitButton = new JButton("Quit");
            //create a button with the word quit on it
        quitButton.setBounds(50, 60, 80, 30); /** see below

        quitButton.addActionListener(new ActionListener() { /** see below
            public void actionPerformed(ActionEvent event) {
```

```

        System.exit(0);
    }
});

panel.add(quitButton); // add the button to the panel widget (object)

setTitle("Quit button");
setSize(300, 200);
setLocationRelativeTo(null);
setDefaultCloseOperation(EXIT_ON_CLOSE);
}

public static void main(String[] args) {
    QuitButtonExample ex = new QuitButtonExample();
    ex.setVisible(true);
}
}

```

The above adds a quit button to a panel and positions the quit button, and then adds the panel to the JFrame. We will get this:

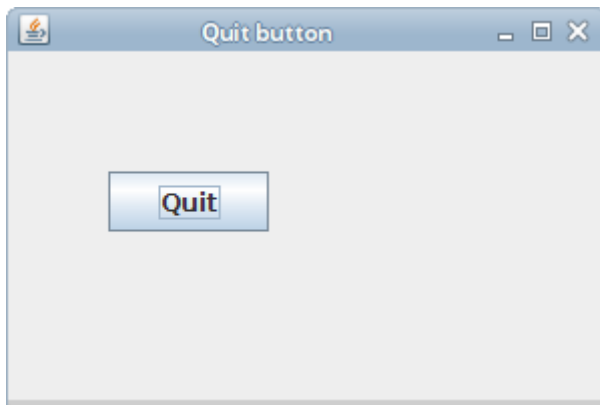


Figure: Quit button

**** More information:**

```
panel.setLayout(null);
```

By default, the JPanel has a `FlowLayout` manager. The layout manager is used to place widgets onto the containers. If we call `setLayout(null)` we can position our components absolutely. For this, we use the `setBounds()` method.

```
JButton quitButton = new JButton("Quit");
quitButton.setBounds(50, 60, 80, 30);
```

Here we create a button. We position it by calling the `setBounds()` method.

**** More info about adding a listener:**

```
quitButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.exit(0);
    }
});
```

Buttons have a built-in method called `addActionListener`, that adds an `ActionListener`, and must define the `actionPerformed` method. In this case we defined the method `actionPerformed` to exit the system. So now if we click on the button. The click will terminate the application.

The Swing layout management

The Java Swing toolkit has two kind of components. Containers and children. The containers group children into suitable layouts. To create layouts, we use **layout managers**. Layout managers are one of the most difficult parts of modern GUI programming.

No manager

We can use no layout manager, if we want. There might be situations, where we might not need a layout manager. But to create truly portable, complex applications, we need layout managers.

Without layout manager, we position components using absolute values.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class AbsoluteExample extends JFrame {

    public AbsoluteExample() {

        initUI();
    }

    public final void initUI() {

        setLayout(null); // means we're positioning absolutely

        JButton ok = new JButton("OK");

        ok.setBounds(50, 50, 80, 25);
        // The parameters are the x and y location and then the width and the
        // height of the button

        JButton close = new JButton("Close");
        close.setBounds(150, 50, 80, 25);

        add(ok); // add the ok button to the JFrame
        add(close); // add the close button to the JFrame
    }
}
```

```

        setTitle("Absolute positioning");
        setSize(300, 250);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }

    public static void main(String[] args) {
        AbsoluteExample ex = new AbsoluteExample();
        ex.setVisible(true);
    }
}

```

This simple example shows two buttons.

FlowLayout manager

This is the simplest layout manager. It is mainly used in combination with other layout managers. When calculating the size of things, a flow layout lets each component assume its natural (preferred) size.

The manager puts components into a row in the order in which they were added. If they do not fit into one row, they go into the next one. The components can be added from the right to the left or vice versa. The manager aligns the components. Implicitly, the components are centered and there is 5px space among components.

```

FlowLayout()
FlowLayout(int align)
FlowLayout(int align, int hgap, int vgap)

```

There are three constructors available for the FlowLayout manager.

The first one creates a manager with implicit values, centered with 5px horizontal and vertical spaces.

The others allow to specify those parametes.

```

import java.awt.Dimension;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.JTree;
import javax.swing.SwingUtilities;

public class FlowLayoutExample extends JFrame {

    public FlowLayoutExample() {

        initUI();
    }
}

```

```

}

public final void initUI() {

    JPanel panel = new JPanel();
    // The implicit layout manager of a JPanel is FlowLayout

    JTextArea area = new JTextArea("text area"); // creates a text area
    area.setPreferredSize(new Dimension(150, 20)); //dimensions: 150x20

    JButton button = new JButton("button1");
    panel.add(button);
    JButton button2 = new JButton("button2");
    panel.add(button2); //add everything to the panel

    panel.add(area);

    add(panel); // add the panel to the JFrame

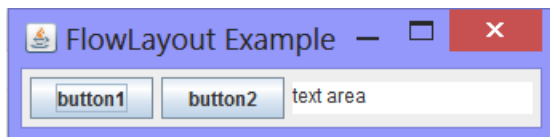
    pack();//Makes the window fit around the components you put inside it

    setTitle("FlowLayout Example");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
}

public static void main(String[] args) {
    FlowLayoutExample ex = new FlowLayoutExample();
    ex.setVisible(true);
}
}

```

The example shows two buttons, and a text area component in the window



```

JTextArea area = new JTextArea("text area");
area.setPreferredSize(new Dimension(100, 100));

```

The flow layout manager sets a **preferred** size for its components. This means, that in our case, the area component will have 150x20px. If we didn't set the preferred size, the component would have a size of its text. No more, no less. Without any text, the component would not be visible at all. As you write and delete some text in the area component the text area will grow and shrink accordingly.

GridLayout

The `GridLayout` layout manager lays out components in a rectangular grid. The container is divided into equally sized rectangles. One component is placed in each rectangle.

```
import java.awt.GridLayout;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class GridLayoutExample extends JFrame {
    String[] buttons; // Note this is a field.  Yep, we can have fields

    public GridLayoutExample() {

        initUI();
        buttons = {"Cls", "Bck", "", "Close", "7", "8", "9", "/", "4",
            "5", "6", "*", "1", "2", "3", "-", "0", ".", "=", "+"};
    }

    public final void initUI() {

        JPanel panel = new JPanel();

        panel.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));
        // setting a border empty, and 5 from the top, left, bottom, and right
        panel.setLayout(new GridLayout(5, 4, 5, 5));
        // number of rows, columns, horizontal, and vertical gap.
        /** see below for more info

        for (int i = 0; i < buttons.length; i++) {

            if (i == 2)
                panel.add(new JLabel(buttons[i])); // making a label
                /** difference between label and button explained below
            else
                panel.add(new JButton(buttons[i]));
        }

        add(panel);

        setTitle("GridLayout");
        setSize(350, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }

    public static void main(String[] args) {
```

```
        GridLayoutExample ex = new GridLayoutExample();
        ex.setVisible(true);
    }
}
```

The example shows a skeleton of a simple calculator tool. It is an ideal example for this layout manager. We put 19 buttons and one label into the manager.

**** More Info:**

```
panel.setLayout(new GridLayout(5, 4, 5, 5));
```

Here we set the grid layout manager for the panel component. The layout manager takes four parameters. The number of rows, the number of columns and the horizontal and vertical gaps between components. Note that the number of rows wins. In other words, if you said the gridlayout should have 5 rows and 4 columns, and then you made 10 buttons, you'd have 5 rows and 2 columns.

****Difference between a label and a button:**

A label is something you can't click on or interact with.

A button is something you can interact with. It has a built-in ActionListener, in which you must fill in the code to specify what should happen when the button is clicked on in the actionPerformed method.



Figure: GridLayout manager

BorderLayout

A `BorderLayout` manager is a very handy layout manager. It divides the space into five regions. North, West, South, East and Centre. Each region can have only one component. If we need to put more components into a region, we can simply put a panel there with a layout manager of our choice. The components in N, W, S, E regions get their **preferred** size. The component in the center takes up space left over.

We want to put space between each of the components usually. To create a border, we either create a new instance of an `EmptyBorder` class (which doesn't actually write a border, just creates space around the edges) or we use a `BorderFactory`. There are a whole bunch of different types of borders. We've seen the `emptyBorder`, above. We'll work with it again.

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Insets;

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.border.EmptyBorder;

public class BorderExample extends JFrame {

    public BorderExample() {

        initUI();
    }

    public final void initUI() {

        JPanel panel = new JPanel(new BorderLayout()); // here we're using
        // the BorderLayout, which lets us put a panel inside a panel
        JPanel top = new JPanel(); // we're creating a second panel

        top.setBackground(Color.gray); // setting the background color
        top.setPreferredSize(new Dimension(250, 150));
        panel.add(top); //adding the top panel to the first panel

        panel.setBorder(new EmptyBorder(new Insets(20, 20, 20, 20)));
        // creating an invisible border 20 pixels wide

        add(panel); // adding the panel to the frame

        pack(); // making the window fit around the elements in the window

        setTitle("Border Example");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }
}
```

```
public static void main(String[] args) {  
    BorderExample ex = new BorderExample();  
    ex.setVisible(true);  
}  
}
```

The example will display a gray panel and border around it.

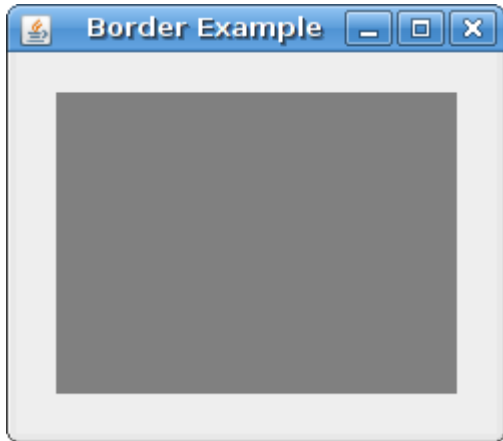


Figure: Border example

BoxLayout

BoxLayout puts components into a row or into a column. It enables nesting, a powerful feature, that makes this manager very flexible. It means, that we can put a box layout into another box layout.

The box layout manager is often used with the `Box` class. This class creates several invisible components, which affect the final layout.

- glue
- strut
- rigid area

Let's say, we want to put two buttons into the right bottom corner of the window. We will use the box layout managers to accomplish this.

When we use a `BoxLayout` manager, we can set a rigid area among our components.

```
import java.awt.Dimension;  
import java.awt.Insets;  
  
import javax.swing.Box;  
import javax.swing.BoxLayout;  
import javax.swing.JButton;  
import javax.swing.JFrame;
```

```

import javax.swing.JPanel;
import javax.swing.SwingUtilities;
import javax.swing.border.EmptyBorder;

public class RigidAreaExample extends JFrame {

    public RigidAreaExample() {

        initUI();
    }

    public final void initUI() {

        JPanel panel = new JPanel();
        panel.setLayout(new BorderLayout(panel, BorderLayout.Y_AXIS));

        panel.setBorder(new EmptyBorder(new Insets(40, 60, 40, 60)));

        panel.add(new JButton("Button"));
        panel.add(Box.createRigidArea(new Dimension(0, 5)));
        panel.add(new JButton("Button"));
        panel.add(Box.createRigidArea(new Dimension(0, 5)));
        panel.add(new JButton("Button"));
        panel.add(Box.createRigidArea(new Dimension(0, 5)));
        panel.add(new JButton("Button"));

        add(panel);

        pack();

        setTitle("RigidArea");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }

    public static void main(String[] args) {
        RigidAreaExample ex = new RigidAreaExample();
        ex.setVisible(true);
    }
}

```

In this example, we display four buttons. By default, there is no space among the buttons. To put some space among them, we add some rigid area.

```
panel.setLayout(new BorderLayout(panel, BorderLayout.Y_AXIS));
```

We use a vertical `BoxLayout` manager for our panel.

```
panel.add(new JButton("Button"));
panel.add(Box.createRigidArea(new Dimension(0, 5)));
panel.add(new JButton("Button"));
```

We add buttons and create a rigid area in between them.



Figure: Rigid area

Next we will create two panels. The basic panel has a vertical box layout. The bottom panel has a horizontal one. We will put a bottom panel into the basic panel. We will right align the bottom panel. The space between the top of the window and the bottom panel is expandable. It is done by the vertical glue.

```
import java.awt.Dimension;

import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class TwoButtonsExample extends JFrame {

    public TwoButtonsExample() {
        initUI();
    }

    public final void initUI() {

        JPanel basic = new JPanel();
        basic.setLayout(new BoxLayout(basic, BoxLayout.Y_AXIS));
        // means the components will be lined up vertically
        add(basic);

        basic.add(Box.createVerticalGlue());
        // makes the space between the boxes expandable

        JPanel bottom = new JPanel();
        bottom.setAlignmentX(1f); // sets a right alignment
        bottom.setLayout(new BoxLayout(bottom, BoxLayout.X_AXIS));
        // horizontal layout

        JButton ok = new JButton("OK");
        JButton close = new JButton("Close");
```

```

        bottom.add(ok);
        bottom.add(Box.createRigidArea(new Dimension(5, 0)));
        // adds rigid space to horizontal flow
        bottom.add(close);
        bottom.add(Box.createRigidArea(new Dimension(15, 0)));
        // adds more horizontal space

        basic.add(bottom);
        basic.add(Box.createRigidArea(new Dimension(0, 15)));
        // adds vertical space between bottom and actual bottom of JFrame

        setTitle("Two Buttons");
        setSize(300, 150);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
    }

    public static void main(String[] args) {
        TwoButtonsExample ex = new TwoButtonsExample();
        ex.setVisible(true);
    }
}

```

Figure: Two buttons The glue is expandable

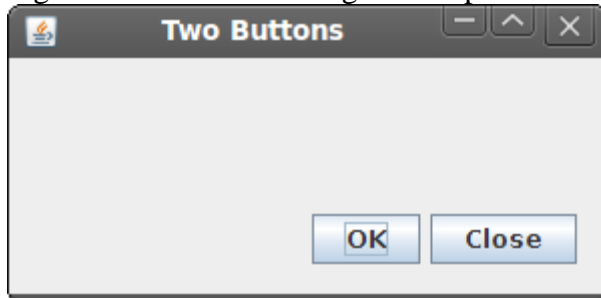


Figure: Two buttons

Buttons, Labels, and TextAreas:

So far we've created buttons with text in them, labels, and textareas using their default values. Buttons are interactive, meaning you can click on them and have something happen. Labels, however, are not. We'll discuss how to make buttons do something shortly. However, there are some methods associated with all three of these used to style the font and background, as well as to set and get the text associated with these widgets.

So far we've created buttons and labels using text. We have more options. We can use images. If we are going to use an image, we have to first create an `ImageIcon` object. We do that as follows:

- `ImageIcon icon = new ImageIcon("middle.gif");` // middle.gif is the name of the image

Then we can include the image in the label:

- `JLabel label1 = new JLabel("Image and Text", icon, JLabel.CENTER);`

This creates a new label. The text for the label is "ImageandText", the image will be icon (middle.gif), and the label will be centered.

- `JLabel label2 = new JLabel("Text-Only Label");`

The above label is only text.

- `JLabel label3 = new JLabel(icon);`

And this label is only the icon.

The above code will give you this:



You can also set the vertical and horizontal position of the text in a label, e.g.,

- `label1.setVerticalTextPosition(JLabel.BOTTOM);`
`//The vertical alignment is at the bottom of the label`
- `label1.setHorizontalTextPosition(JLabel.CENTER);`
`// The horizontal position of the text is centered.`

Other methods associated with labels include:

- `label1.setText(String)`
- `label1.setIcon(icon)`
- `label1.getText()`
- `label1.getIcon()`

This allows you to change the label's text after the label has been created, and to get the label's text (this one is more useful with buttons). You can do the same with the label's icon (image).

- `label1.setBackground(Color.red);`
- `Font font=new Font("Impact",Font.BOLD,16);`
- `label.setFont(font);`

This sets the background color to red (using a color object). And it sets the font on the label to Impact, bold, and 16pixels.

All of the above methods can be used with **buttons** as well.

You can also style a text area using the above methods. In addition, you can set the dimensions of your text area either when it's being created or using:

```
Textarea.setSize(100, 1);
```

Note that `setText` may come in useful in the near future.

ActionEvents:

Finally, buttons have available to them action events. When you click on a button, we can make something happen. To do that we must attach to the button an `ActionListener` that defines the `actionPerformed` method. We've seen this before in the first example.

```
JButton closeButton = new JButton("Close");
```

The button is the **event source**. It will generate events.

```
closeButton.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent event) {  
        System.exit(0);  
        // the action that happens when you click on the button  
        // you can place any code in here that you want to happen  
    }  
});
```

Here we **register** an action listener with the button. This way, the events are sent to the **event target**. The event target in our case is the `ActionListener` class. In other words, there is a “listener” waiting for the button to be clicked. When it is clicked, the listener calls the `actionPerformed` method with the action event (the click). The code associated with the `actionPerformed` method is executed.

Your Job:

Create a JFrame that looks like this, with a text area at the top, then a grid of number buttons, with a label in the bottom left corner and a submit button in the bottom right corner.



Now add an `ActionEvent` to each button so that when you click on the button, it adds to the text area. So, for instance, if I clicked on 3, then 6, then 4, then 7, I should have:



(Easiest way to do this- make the textarea a field of the class, and then use the setText method associated with the textarea.)

Choose one of the versions below to turn in:

Version 1: (15 pts total):

Modify the code so that when the submit button is clicked on, the text area is reset to blank.

Version 2: (18 pts total):

Modify the screen so that the label in the bottom left corner is a plus sign. Now when you hit submit, you should add all the numbers together and display the total.

(For this you'll want to use `str.split('+')`, which splits a string into an array of strings, separated by the `+`.)

Version 3: (23 pts total):

Modify the code so that, if I click on the submit button, the text in the text area should be changed to the letter equivalent of the number string (remember lab 4?) So you should have:

