1. Swing

Note: Most of the stuff stolen from or from the jdk documentation. Most programs are modified or written by me.

This section explains the various elements of the graphical user interface, i.e., most of the classes of the Swing using a set of very primitive applications. The classes are part of the package.

Note: "Swing" was the codename of the project that developed the new components. Although it’s an unofficial name, it’s frequently used to refer to the new components and related API. It’s immortalized in the package names for the Swing API, which begin with javax.swing.
1.1. JFC

JFC stands for Java Foundation Classes, which encompass a group of features to help people build graphical user interfaces (GUIs). The JFC was first announced at the 1997 JavaOne developer conference and is defined as containing the following features:

The Swing Components
Include everything from buttons to split panes to tables. You can see mugshots of all the components in A Visual Index to the Swing Components.

Pluggable Look & Feel Support
Gives any program that uses Swing components a choice of look and feel. For example, the same program can use either the Java Look & Feel or the Windows Look & Feel. We expect many more look-and-feel packages -- including some that use sound instead of a visual "look" -- to become available from various sources.

Accessibility API
Enables assistive technologies such as screen readers and Braille displays to get information from the user interface.

Java 2D API (JDK 1.{234} only)
Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and in applets.

Drag and Drop Support (JDK 1.{234} only)
Provides the ability to drag and drop between a Java application and a native application.

The first three JFC features were implemented without any native code, relying only on the API defined in JDK 1.1. As a result, they could and did become available as an extension to JDK 1.1. This extension was released as JFC 1.1, which is sometimes called "the Swing release." The API in JFC 1.1 is often called the Swing API.

Please see also
1.2. SwingSet 3

Download it from:

% cd /usr/local/jdk/demo/jfc/SwingSet3 && java -jar SwingSet2.jar
% appletviewer SwingSet2.html

- SwingSet.html:

```html
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
<head>
  <title>SwingSet demo</title>
</head>
<body>
  <h1>SwingSet demo</h1>
  <applet code=SwingSet2Applet
          codebase="." archive="SwingSet3.jar"
          width=695 height=525>
  </applet>
</body>
</html>
```

1.3. Top-Level Containers

The components at the top of any Swing containment hierarchy.

It exists mainly to provide a place for other Swing components to paint themselves. The commonly used top-level containers are frames (JFrame), dialogs (JDialog), and applets (JApplet).

The following statement can be found here: but it does not work in MacOS 10/Sun OS 5.9.

To view the containment hierarchy for any frame or dialog, click its border to select it, and then press Control-Shift-F1. A list of the containment hierarchy will be written to the standard output stream.

A visual index to the swing components can be found at:

1.4. General-Purpose Containers

Intermediate containers that can be used under many different circumstances.
1.5. Special-Purpose Containers

Intermediate containers that play specific roles in the UI.
1.6. Basic Controls
Atomic components that exist primarily to get input from the user; they generally also display some simple state.
1.7. Uneditable Information Displays
Atomic components that exist solely to give the user information.
1.8. Editable Displays of Formatted Information

Atomic components that display highly formatted information that (if you choose) can be edited by the user.
1.9. Swing versus AWT

The AWT components are those provided by the JDK 1.0 and 1.1 platforms. Although JDK 1.3 still supports the AWT components, SUN strongly encourages to use Swing components instead. You can identify Swing components because their names start with J.

The AWT button class, for example, is named Button, while the Swing button class is named JButton.

Additionally, the AWT components are in the java/awt package, while the Swing components are in the javax.swing package.

The biggest difference between the AWT components and Swing components is that the Swing components are implemented with absolutely no native code. Since Swing components aren’t restricted to the least common denominator — the features that are present on every platform — they can have more functionality than AWT components. Because the Swing components have no native code, they can be shipped as an add-on to JDK 1.1, in addition to being part of JDK 1.3.
Even the simplest Swing components have capabilities far beyond what the AWT components offer:

- Swing buttons and labels can display images instead of, or in addition to, text.
- You can easily add or change the borders drawn around most Swing components. For example, it’s easy to put a box around the outside of a container or label.
- You can easily change the behavior or appearance of a Swing component by either invoking methods on it or creating a subclass of it.
- Swing components don’t have to be rectangular. Buttons, for example, can be round.
- Assistive technologies such as screen readers can easily get information from Swing components. For example, a tool can easily get the text that’s displayed on a button or label.
Swing lets you specify which look and feel your program’s GUI uses. By contrast, AWT components always have the look and feel of the native platform.

CDE:

Java:
Windows:

Another interesting feature is that Swing components with state use models to keep the state. A JSlider, for instance, uses a BoundedRangeModel object to hold its current value and range of legal values. Models are set up automatically, so you don’t have to deal with them unless you want to take advantage of the power they can give you.

Swing components aren’t thread safe. If you modify a visible Swing component -- invoking its setText method, for example -- from anywhere but an event handler, then you need to take special steps to make the modification execute on the event-dispatching thread. This isn’t an issue for many Swing programs, since component-modifying code is typically in event handlers.

The containment hierarchy for any window or applet that contains Swing components must have a Swing top-level container at the root of the hierarchy. For example, a main window should be implemented as a JFrame instance rather than as a Frame instance.

You don’t add components directly to a top-level container such as a JFrame. Instead, you add components to a container (called the content pane) that is itself contained by the JFrame.
1.10. The First Swing Program

```java
import javax.swing.*; //This is the final package name.
import java.awt.*;
import java.awt.event.*;

public class SwingApplication {
    private static String labelPrefix = "So many clicks: ";
    private int numClicks = 0;

    public Component createComponents() {
        final JLabel label = new JLabel(labelPrefix + "0 ");

        JButton button = new JButton("Swing button!");
        button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    numClicks++;
                    label.setText(labelPrefix + numClicks);
                    System.out.println("getActionCommand: "+
                        e.getActionCommand());
                    System.out.println("getModifiers: "+
                        e.getModifiers());
                    System.out.println("paramString: "+
                        e.paramString());
                }
            }
        );
        label.setLabelFor(button);

        /* An easy way to put space between a top-level container
         * and its contents is to put the contents in a JPanel
         * that has an "empty" border.
         */
        JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createEmptyBorder(
            30, //top
            30, //left
            10, //bottom
            30) //right
        );

        pane.setLayout(new FlowLayout());
        pane.add(button);
        pane.add( new JFileChooser()) ;
        pane.add(new JCheckBox("not me", true) );
        pane.add(label);

        return pane;
    }
}
public static void main(String[] args) {
    String lookAndFeel;

    lookAndFeel = UIManager.getCrossPlatformLookAndFeelClassName();
    if (args.length == 1)
    {
        if (args[0].equals("motif"))
            lookAndFeel = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
        else if (args[0].equals("metal"))
            lookAndFeel = "javax.swing.plaf.metal.MetalLookAndFeel";
        else if (args[0].equals("system"))
            lookAndFeel = UIManager.getSystemLookAndFeelClassName();

        try {
            UIManager.setLookAndFeel(lookAndFeel);
        } catch (Exception e) { }

        //Create the top-level container and add contents to it.
        JFrame frame = new JFrame("SwingApplication");
        SwingApplication app = new SwingApplication();
        Component contents = app.createComponent();
        frame.getContentPane().add(contents);

        //Finish setting up the frame, and show it.
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        frame.pack();
        frame.setVisible(true);
    }
}

Source Code: Src/12/SwingApplication.java

Output:
getActionCommand: Swing button!
getModifiers: 0
paramString: ACTION_PERFORMED,cmd=Swing button!
getActionCommand: Swing button!
getModifiers: 0
paramString: ACTION_PERFORMED,cmd=Swing button!
getActionCommand: Swing button!
getModifiers: 0
...

Default:
Motif:
1.11. Layout Management

Layout management is the process of determining the size and position of components. By default, each container has a layout manager -- an object that performs layout management for the components within the container. Components can provide size and alignment hints to layout managers, but layout managers have the final say on the size and position of those components.

The Java platform supplies five commonly used layout managers: BorderLayout, BoxLayout, FlowLayout, GridBagLayout, and GridLayout. These layout managers are designed for displaying multiple components at once. A sixth provided class, CardLayout, is a special-purpose layout manager used in combination with other layout managers.

Whenever you use the add method to put a component in a container, you must take the container’s layout manager into account. Some layout managers, such as BorderLayout, require you to specify the component’s relative position in the container, using an additional argument with the add method. Occasionally, a layout manager such as GridBagLayout requires elaborate setup procedures. Many layout managers, however, simply place components based on the order they were added to their container.

1.12. Layout Manager

See also

1.13. GridLayout Manager

The GridLayout class is a layout manager that lays out a container’s components in a rectangular grid. The container is divided into equal-sized rectangles, and one component is placed in each rectangle.

1.14. GridBagLayout Manager

GridBagLayout is the most sophisticated, flexible layout manager the Java platform provides. It aligns components by placing them within a grid of cells, allowing some components to span more than one cell. The rows in the grid aren’t necessarily all the same height; similarly, grid columns can have different widths.

1.15. FlowLayout Manager

A flow layout arranges components in a left-to-right flow, much like lines of text in a paragraph. Flow layouts are typically used to arrange buttons in a panel. It will arrange buttons left to right until no more buttons fit on the same line. Each line is centered.

1.16. Viewport Manager

The default layout manager for JViewport. JViewportLayout defines a policy for layout that should be useful for most applications. The viewport makes its view the same size as the viewport, however it will not make the view smaller than its minimum size. As the viewport grows the view is kept bottom justified until the entire view is visible, subsequently the view is kept top justified.
1.17. ScrollPaneLayout Manager

The layout manager used by JScrollPane. JScrollPaneLayout is responsible for nine components: a viewport, two scrollbars, a row header, a column header, and four "corner" components.
1.18. A Calculator Layout

```java
import javax.swing.*; //This is the final package name.
import java.awt.*;
import java.awt.event.*;

public class Calculator {
    private JLabel label = null;
    private int numClicks = 0;

    public Component createLabel() {
        final JLabel label = new JLabel("0");
        label.setText("0");
        label.setHorizontalAlignment(JLabel.RIGHT);
        this.label = label;
        return label;
    }

    public Component createButtons() {
        JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createLoweredBevelBorder());
        pane.setLayout(new GridLayout(5, 2));
        for (int index = 0; index < 10; index++) {
            JButton button = new JButton(new Integer(index).toString());
            button.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    label.setText(e.getActionCommand());
                }
            });
            pane.add(button);
        }
        return pane;
    }

    public Component createOps() {
        JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createLoweredBevelBorder());
        pane.setLayout(new GridLayout(2, 2));
        pane.add(new JButton("+"));
        pane.add(new JButton("-"));
        pane.add(new JButton("*"));
        pane.add(new JButton("/"));
        return pane;
    }

    public Component createComponents() {
        JPanel pane = new JPanel();
        pane.setBorder(BorderFactory.createLoweredBevelBorder());
        return pane;
    }
}
```
pane.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5));
pane.add(createLabel());
pane.add(createButtons());
pane.add(createOps());

return pane;
}

public static void main(String[] args) {
    String lookAndFeel;

    lookAndFeel=UIManager.getCrossPlatformLookAndFeelClassName();
    if (args.length == 1)
    {
        if (args[0].equals("motif") )
            lookAndFeel =
                "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
        if (args[0].equals("metal") )
            lookAndFeel =
                "javax.swing.plaf.metal.MetalLookAndFeel";
        else if (args[0].equals("system") )
            lookAndFeel =
                UIManager.getSystemLookAndFeelClassName();
    }
    try {
        UIManager.setLookAndFeel(lookAndFeel);
    } catch (Exception e) {} 

    //Create the top-level container and add contents to it.
    JFrame frame = new JFrame("Calculator");
    Calculator app = new Calculator();
    Component contents = app.createComponent();
    frame.getContentPane().add(contents);
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    frame.pack();
    frame.setVisible(true);
}

Source Code: Src/12/Calculator.java

1.19. GridBagLayout

Essentially, GridBagLayout places components in rectangles (cells) in a grid, and then uses the compo-

nents’ preferred sizes to determine how big the cells should be.
import javax.swing.*; //This is the final package name.
import java.awt.*;
import java.awt.event.*;

public class GB {

    public Component createComponents() {
        JButton button;
        JPanel contentPane = new JPanel();
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints c = new GridBagConstraints();
        contentPane.setLayout(gridbag);
        c.fill = GridBagConstraints.HORIZONTAL;

        button = new JButton("Button 1");
        c.weightx = 0.5;
        c.gridx = 0;
        c.gridy = 0;
        gridbag.setConstraints(button, c);
        contentPane.add(button);

        button = new JButton("2");
        c.weightx = 4;
        c.gridx = 1;
        c.gridy = 0;
        gridbag.setConstraints(button, c);
        contentPane.add(button);

        button = new JButton("Button 3");
        c.weightx = 8;
        c.gridx = 2;
        c.gridy = 0;
        gridbag.setConstraints(button, c);
        contentPane.add(button);

        button = new JButton("Long-Named Button 4");
        c.ipady = 40; //make this component tall
        c.weightx = 0.0;
        c.insets = new Insets(20,0,0,0); //top padding
        c.gridy = 1;
        gridbag.setConstraints(button, c);
        contentPane.add(button);

        button = new JButton("Button 5");
        c.ipady = 0; //reset to default
        c.weighty = 1.0; //request any extra vertical space
        c.anchor = GridBagConstraints.SOUTH; //bottom of space
        c.insets = new Insets(10,0,0,0); //top padding
        c.gridx = 0;
        c.gridy = 1;
        gridbag.setConstraints(button, c);
        contentPane.add(button);
    }
}
c.gridx = 1; //aligned with button 2
  c.gridwidth = 2; //2 columns wide
  c.gridy = 2; //third row
  gridbag.setConstraints(button, c);
  contentPane.add(button);
  return contentPane;
}

public static void main(String[] args) {
  String lookAndFeel = UIManager.getCrossPlatformLookAndFeelClassName();
  try {
    UIManager.setLookAndFeel(lookAndFeel);
  } catch (Exception e) {
  }
  JFrame frame = new JFrame("GB");
  GB app = new GB();
  Component contents = app.createComponent();
  frame.getContentPane().add(contents);
  //Finish setting up the frame, and show it.
  frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  });
  frame.pack();
  frame.setVisible(true);
}

Source Code: Src/12/GB.java

When you enlarge the window the program brings up, the columns grow proportionately. This is because each component in the first row, where each component is one column wide, has

weightx = 1.0.

The actual value of these components’ weightx is unimportant. What matters is that all the components, and consequently, all the columns, have an equal weight that is greater than 0. If no component managed by the GridBagLayout had weightx set, then when the components’ container was made wider, the components would stay clumped together in the center of the container.
1.20. A Second Try
import javax.swing.*;  //This is the final package name.
import java.awt.*;
import java.awt.event.*;

public class Lotto {

    public void createButtons( GridBagLayout gridbag,
            JPanel contentPane,
            int gridy) {
        int number = 1;

        GridBagConstraints c = new GridBagConstraints();
        c.gridwidth = 1;
        c.weightx = 1.0;
        c.insets = new Insets(5, 3, 3, 5);
        c.fill = GridBagConstraints.BOTH;
        JButton button;
        JButton o;

        for (int col = 1; col <= 7; col++) {
            for (int row = 1; row <= 7; row++) {
                button = new JButton(
                        new Integer(number++).toString());
                button.setBackground(Color.yellow);
                button.setMinimumSize(
                        button.getSize());
                button.addActionListener(
                        new ActionListener() {
                            public void actionPerformed(ActionEvent e) {
                                o = (JButton)e.getSource();  // will this compile
                                System.out.println("getText = " +
                                        o.getText());
                                if (o.getBackground() == Color.red)
                                        o.setBackground(Color.yellow);
                                else
                                        o.setBackground(Color.red);
                            }
                        });
                c.gridx = row;
                c.gridy = col + gridy;
                gridbag.setConstraints(button, c);
                contentPane.add(button);
            }
        }
    }

    public void createSeparator( GridBagLayout gridbag,
            JPanel contentPane,
            int gridy) {
        GridBagConstraints c = new GridBagConstraints();
        public void actionPerformed(ActionEvent e) {
            o = (JButton)e.getSource();  // will this compile
            System.out.println("getText = " +
                    o.getText());
            if (o.getBackground() == Color.red)
                    o.setBackground(Color.yellow);
            else
                    o.setBackground(Color.red);
        };
        c.gridx = row;
        c.gridy = col + gridy;
        gridbag.setConstraints(button, c);
        contentPane.add(button);
    }

    public void createSeparator( GridBagLayout gridbag,
            JPanel contentPane,
            int gridy) {
        GridBagConstraints c = new GridBagConstraints();
    }
JSeparator sep = new JSeparator(SwingConstants.HORIZONTAL);
c.fill = GridBagConstraints.BOTH;
c.gridx = 0;
c.gridy = gridy;
c.gridwidth = 8;
c.insets = new Insets(5,3,3,5);
gridbag.setConstraints(sep, c);
contentPane.add(sep);

public Component createComponents() {
    JSeparator sep;
    JButton button;
    JLabel label;
    JPanel contentPane = new JPanel();
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints c = new GridBagConstraints();
    contentPane.setLayout(gridbag);

    label = new JLabel("0");
c.weightx = 0.5;
c.gridx = 0;
c.gridy = 0;
c.gridwidth = 8;
c.ipady = 10;
    gridbag.setConstraints(label, c);
    contentPane.add(label);

    createSeparator(gridbag, contentPane, 1);
    createButtons(gridbag, contentPane, 3);
    createSeparator(gridbag, contentPane, 11);

    JLabel label_1 = new JLabel("0 correct");
c.weightx = 0.5;
c.gridx = 0;
c.gridy = 12;
c.gridwidth = 4;
c.ipady = 10;
    gridbag.setConstraints(label_1, c);
    contentPane.add(label_1);

    JTextField tField = new JTextField("0", 5);
tField.setHorizontalAlignment(JTextField.RIGHT);
tField.setEditable(false);
c.weightx = 0.5;
c.gridx = 4;
c.gridy = 12;
c.gridwidth = 4;
c.ipady = 10;
    gridbag.setConstraints(tField, c);
    contentPane.add(tField);
    createSeparator(gridbag, contentPane, 13);
```java
public static void main(String[] args) {
    String lookAndFeel = UIManager.getCrossPlatformLookAndFeelClassName();
    try {
        UIManager.setLookAndFeel(lookAndFeel);
    } catch (Exception e) { }
    JFrame frame = new JFrame("Lotto");
    Lotto app = new Lotto();
    Component contents = app.createComponent();
    frame.getContentPane().add(contents);
    // Finish setting up the frame, and show it.
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    frame.pack();
    frame.setVisible(true);
}
```

Source Code: Src/12/Lotto.java

### 1.21. General Ideas for Selecting a Layout Manager

- Divide the area in logical pieces
- Do the objects form a regular/grid pattern?
- What should happen if there is a change in size?

### 1.22. How to Choose a Layout Manager

Stolen from

- You need to display a component in as much space as it can get.

  Consider using BorderLayout or GridBagLayout. If you use BorderLayout, you’ll need to put the space-hungry component in the center. With GridBagLayout, you’ll need to set the constraints for the component so that fill=GridBagConstraints.BOTH. Another possibility is to use BoxLayout, making the space-hungry component specify very large preferred and maximum sizes.

- You need to display a few components in a compact row at their natural size.

  Consider using a JPanel to group the components and using either the JPanel’s default FlowLayout manager or the BoxLayout manager.
• You need to display a few components of the same size in rows and columns.

GridLayout is perfect for this.

• You need to display a few components in a row or column, possibly with varying amounts of space between them, custom alignment, or custom component sizes.

BoxLayout is perfect for this.

• You have a complex layout with many components. Consider either using GridBagLayout or grouping the components into one or more JPanels to simplify layout. Each JPanel might use a different layout manager.

1.23. Creating a Custom Layout Manager
Instead of using one of the Java platform’s layout managers, you can write your own. Layout managers must implement the LayoutManager interface, which specifies the five methods every layout manager must define. Optionally, layout managers can implement LayoutManager2, which is a subinterface of LayoutManager.