Java Swings/Print version

Contents

- 1 Java Swing
  - 1.1 Success Stories
    - 1.1.1 NetBeans
  - 1.2 JFC
    - 1.2.1 Swing GUI Components
    - 1.2.2 Pluggable Look-and-Feel (PLAF) Support
    - 1.2.3 Accessibility API
    - 1.2.4 Java 2D API
    - 1.2.5 Internationalization
  - 1.3 Reference
- 2 Overview
  - 2.1 Overview
  - 2.2 Notes
  - 2.3 External links
- 3 AWT
  - 3.1 AWT Native Interface example walk-through
    - 3.1.1 Create the Java application
    - 3.1.2 Create the C++ header file
    - 3.1.3 Implement the C++ native code
    - 3.1.4 Run the example
  - 3.2 Native painting
- 4 Swings
  - 4.1 First GUI
    - 4.1.1 How it works?
  - 4.2 Diving Into Swing
    - 4.2.1 Frame with Title
    - 4.2.2 Adding Labels
    - 4.2.3 Label and Text
    - 4.2.4 Layout Manager
    - 4.2.5 Packing Frames
- 5 First Examples
  - 5.1 JFrames
  - 5.2 JLabels
  - 5.3 Summary
- 6 Swing Top Level Containers
  - 6.1 JFrame
  - 6.2 JOptionPane
  - 6.3 JApplet
- 7 Swing Layouts
  - 7.1 BorderLayout
  - 7.2 FlowLayout
  - 7.3 BoxLayout
  - 7.4 CardLayout
  - 7.5 GridLayout
  - 7.6 GridBagLayout
  - 7.7 SpringLayout
- 8 Event Handling
  - 8.1 Events
  - 8.2 JButtons
- 9 MVC
- 10 Responsive Swing Application

- 10.1 The Program
- 10.2 Multi-threading in Swing
- 11 Large Examples
  - 11.1 Calculator
    - 11.1.1 Main Method
    - 11.1.2 Constructor
    - 11.1.3 Listeners
    - 11.1.4 Building the buttons
    - 11.1.5 Building the panels
    - 11.1.6 Full code for the Calculator
- 12 Reference
  - 12.1 Website

Java Swing

Java Swing is a library/toolkit released by Oracle as a part of Java language which enables Java programmers to create GUIs and rich client applications.

Content

- Java Swing
- Overview
- AWT
- Dive into Swings
- First Examples
- Swing Top Level Containers
- Swing Layouts
- Event Handling
- MVC architecture
- Responsive Swing Application
- Large Examples

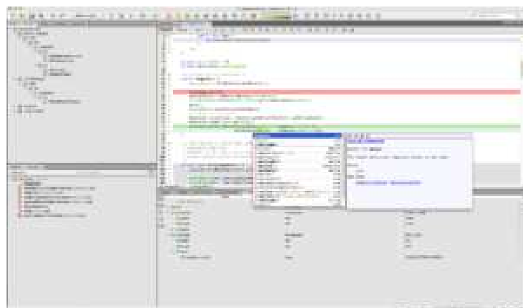
Appendix

- Reference

Success Stories

There are many applications written in Java, and a new one pops up every day. Here are some examples:

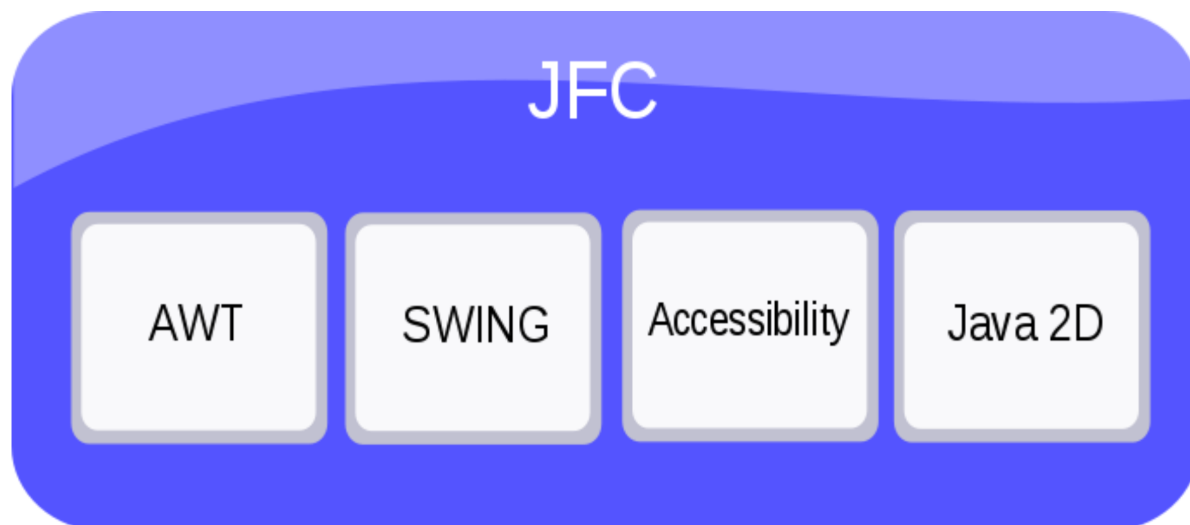
NetBeans



NetBeans is a Integrated Development Environment (IDE) written completely using Java. Because of Java's portability, it can run on any platform, as long as Java is installed. Initially it was only a Java IDE, but now it supports many languages like C, C++, Python, and PHP, to name a few.

JFC

Swing is a part of Java Foundation classes (JFC). JFC consists of the following modules



Swing GUI Components

Java Swing Components are components for the Graphical User Interfaces of Applications. Some examples of Swing GUI components are lists, combo-boxes, labels, text areas, editor panes, buttons, menus, etc. Many components have built-in features such as print, drag & drop, sort, open file, save file, etc.

Pluggable Look-and-Feel (PLAF) Support

The Look and Feel module defines how all of the components in the application look. Each Java Swing Application has an entirely separate L&F from the rest of the applications on the computer, including other Java Applications. Some examples of L&Fs are: Nimbus (really new), Metal, Aqua(Macs only), Windows Aero & Windows Classic (Windows only), and Motif(Highly customizable by the User of the Application). There are also many third-party Look and Feels available.

Accessibility API

Not all humans are perfect. There is a good chance that a disabled person might not be able to use your program/application, so your program must be accessible to him or her. Java Foundation Classes Accessibility API enables you to create programs that are accessible to disabled people by providing them with screen readers, braille displays, etc., so that they can get information from the User Interface.

Java 2D API

The Java 2D API provides a rich set of graphics classes and utilities for drawing 2D graphics. It can even generate output for printers.

Internationalization

English is not the only language on this planet. Many people don't know any other language than their native tongue. Hence, when a programmer writes an application he/she must consider how to release it in as many languages as possible. Java i18n (there are 18 letter's between 'i' and 'n' in Internationalization) helps programmers develop applications in various different languages, so that you can target a global audience.

Reference

http://docs.oracle.com/javase/tutorial/uiswing/start/about.html

Overview

Swing is a tool kit in Java which provides a way to build cross platform user interfaces. It is built on top of and designed as a replacement for AWT, the other UI toolkit built into Java. SWT is a third toolkit you may hear about. SWT is an open source

toolkit and is a full topic in of itself, see SWT's Homepage (http://www.eclipse.org/swt/) for more information.

Overview

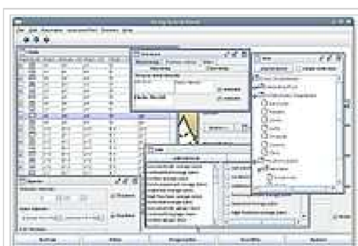
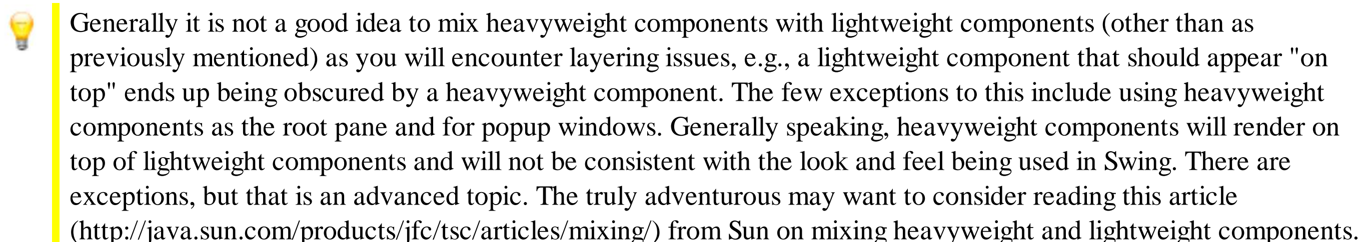


Figure 10.1: Example of a Swing application

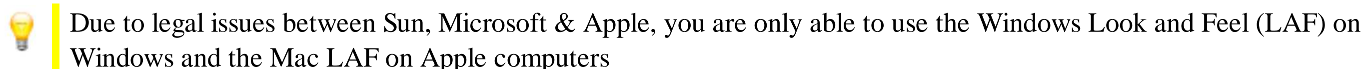
Swing[1] provides many controls and widgets to build user interfaces with. Swing class names typically begin with a J such as `JButton`, `JList`, `JFrame`. This is mainly to differentiate them from their AWT[2] counterparts and in general are one-to-one replacements. Swing is built on the concept of *Lightweight components* vs AWT and SWT's concept of *Heavyweight components*. The difference between the two is that the Lightweight components are rendered (drawn) using purely Java code, such as drawLine and drawImage, whereas Heavyweight components use the native operating system to render the components.

Some components in Swing are actually heavyweight components. The top-level classes and any derived from them are heavyweight as they extend the AWT versions. This is needed because at the root of the UI, the parent windows need to be provided by the OS. These top-level classes include `JWindow`, `JFrame`, `JDialog` and `JApplet`. All Swing components to be rendered to the screen must be able to trace their way to a root window or one of those classes.


Generally it is not a good idea to mix heavyweight components with lightweight components (other than as previously mentioned) as you will encounter layering issues, e.g., a lightweight component that should appear "on top" ends up being obscured by a heavyweight component. The few exceptions to this include using heavyweight components as the root pane and for popup windows. Generally speaking, heavyweight components will render on top of lightweight components and will not be consistent with the look and feel being used in Swing. There are exceptions, but that is an advanced topic. The truly adventurous may want to consider reading this article (http://java.sun.com/products/jfc/tsc/articles/mixing/) from Sun on mixing heavyweight and lightweight components.

So what does using Swing get you? So far we've only talked about components and rendering. Well, to start with you get the following.

- Controls: Buttons, Check Boxes, Lists, Trees, Tables, Combo boxes (dropdown list), Menus, Text fields
- Displays: Labels, Progress bars, Icons, Tool Tips
- Pluggable look and feels (PLAFs): Windows, CDE/Motif, Mac. Allows for "skinning" the application without changing any code


Due to legal issues between Sun, Microsoft & Apple, you are only able to use the Windows Look and Feel (LAF) on Windows and the Mac LAF on Apple computers

- Standard Top-Level Windows: Windows, Frames, Dialogs etc.
- Event Listener APIs
- Key bindings & mnemonics: Allow keystrokes to map to specific actions.

Notes

1. Swing is also referred to incorrectly as JFC (Java Foundation Classes), however the JFC includes APIs in addition to Swing, such as the Java 2D and Drag-n-Drop APIs.
2. AWT is the Abstract Windowing Toolkit, where the components are rendered by the native operating system

External links

- Swing Tutorial, interactive lessons based on examples (http://javalessons.com/cgi-bin/fun/java-tutorials-main.cgi?sub=gui&ses=ao789)
- The Java Tutorials: Graphic User Interfaces (http://java.sun.com/docs/books/tutorial/ui/index.html)

AWT

AWT stands for Abstract Windowing Toolkit. Prior to Swing, AWT was used to develop GUI and rich client interface, but AWT had one major problem. AWT was platform dependent, which means a program written in AWT behaves differently in different platforms. Hence it defeats **WORA** (Write Once, Run Anywhere) purpose which is the key Java philosophy.

Swing on the other hand is purely (100%) written in Java. A swing application developed on one platform behaves the same on any other platform in which Java is installed. Hence today almost all Java programmers prefer Swing over AWT for GUI development.

Java AWT Native Interface is an interface for the Java programming language that enables rendering libraries compiled to native code to draw directly to a Java Abstract Window Toolkit (AWT) object drawing surface.

The Java Native Interface (JNI) enabled developers to add platform-dependent functionality to Java applications. The JNI enables developers to add time-critical operations like mathematical calculations and 3D rendering. Previously, native 3D rendering was a problem because the native code didn't have access to the graphic context. The AWT Native Interface is designed to give developers access to an AWT `Canvas` for direct drawing by native code. In fact, the Java 3D API extension to the standard Java SE JDK relies heavily on the AWT Native Interface to render 3D objects in Java. The AWT Native Interface is very similar to the JNI, and, the steps are, in fact, the same as those of the JNI.

The AWT Native Interface was added to the Java platform with the Java Platform, Standard Edition J2SE 1.3 ("Kestrel") version.

AWT Native Interface example walk-through

Create the Java application

Type in this in a .java file named JavaSideCanvas and compile:

JavaSideCanvas.java

```
import java.awt.Canvas;
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class JavaSideCanvas extends Canvas {

    static {
        System.loadLibrary("NativeSideCanvas");
    }

    public native void paint(Graphics g);

    public static void main(String[] args) {
        Frame frame = new Frame();
        frame.setBounds(0, 0, 500, 500);
        JavaSideCanvas jsc = new JavaSideCanvas();
        frame.add(jsc);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent ev) {
                System.exit(0);
            }
        });
        frame.show();
    }
}
```

The `paint()` method will be simply invoked when the AWT event dispatching thread *repaints* the screen.

Create the C++ header file

Create the C++ header file as usual for JNI.

The header file looks like this now:

JavaSideCanvas.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JavaSideCanvas */

#ifndef _Included_JavaSideCanvas
#define _Included_JavaSideCanvas
#ifdef __cplusplus
extern "C" {
#endif
#undef JavaSideCanvas_FOCUS_TRAVERSABLE_UNKNOWN
#define JavaSideCanvas_FOCUS_TRAVERSABLE_UNKNOWN 0L
#undef JavaSideCanvas_FOCUS_TRAVERSABLE_DEFAULT
#define JavaSideCanvas_FOCUS_TRAVERSABLE_DEFAULT 1L
#undef JavaSideCanvas_FOCUS_TRAVERSABLE_SET
#define JavaSideCanvas_FOCUS_TRAVERSABLE_SET 2L
#undef JavaSideCanvas_TOP_ALIGNMENT
#define JavaSideCanvas_TOP_ALIGNMENT 0.0f
#undef JavaSideCanvas_CENTER_ALIGNMENT
#define JavaSideCanvas_CENTER_ALIGNMENT 0.5f
#undef JavaSideCanvas_BOTTOM_ALIGNMENT
#define JavaSideCanvas_BOTTOM_ALIGNMENT 1.0f
#undef JavaSideCanvas_LEFT_ALIGNMENT
#define JavaSideCanvas_LEFT_ALIGNMENT 0.0f
#undef JavaSideCanvas_RIGHT_ALIGNMENT
#define JavaSideCanvas_RIGHT_ALIGNMENT 1.0f
#undef JavaSideCanvas_serialVersionUID
#define JavaSideCanvas_serialVersionUID -7644114512714619750i64
#undef JavaSideCanvas_serialVersionUID
#define JavaSideCanvas_serialVersionUID -2284879212465893870i64
/*
 * Class:     JavaSideCanvas
 * Method:    paint
 * Signature: (Ljava/awt/Graphics;)V
 */
JNIEXPORT void JNICALL Java_JavaSideCanvas_paint
  (JNIEnv *, jobject, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Implement the C++ native code

Type this in a file named "NativeSideCanvas.cpp" and compile into a library.

(Microsoft) Don't forget to link this with "jawt.lib" and "gdi32.lib". These libraries are needed because the code draws a rectangle using routines from these libraries.

Microsoft C++:

NativeSideCanvas.cpp

```
#include "jawt_md.h"
#include <assert.h>
#include "JavaSideCanvas.h"
JNIEXPORT void JNICALL Java_JavaSideCanvas_paint(JNIEnv* env, jobject canvas, jobject graphics)
{
      JAWT awt;
      JAWT_DrawingSurface* ds;
```

```
    JAWT_DrawingSurfaceInfo* dsi;
    JAWT_Win32DrawingSurfaceInfo* dsi_win;
    jboolean result;
    jint lock;

    // Get the AWT
    awt.version = JAWT_VERSION_1_3;
    result = JAWT_GetAWT(env, &awt);
    assert(result != JNI_FALSE);

    // Get the drawing surface
    ds = awt.GetDrawingSurface(env, canvas);
    assert(ds != NULL);

    // Lock the drawing surface
    lock = ds->Lock(ds);
    assert((lock & JAWT_LOCK_ERROR) == 0);

    // Get the drawing surface info
    dsi = ds->GetDrawingSurfaceInfo(ds);

    // Get the platform-specific drawing info
    dsi_win = (JAWT_Win32DrawingSurfaceInfo*)dsi->platformInfo;

    //////////////////////////////
    // !!! DO PAINTING HERE !!! //
    //////////////////////////////

    //Simple paints a rectangle (GDI32)
    //It's a GDI API, Use Windows provided GDI library in order to compile the code.
    Rectangle(dsi_win->hdc, 50, 50, 200, 200);

    // Free the drawing surface info
    ds->FreeDrawingSurfaceInfo(dsi);

    // Unlock the drawing surface
    ds->Unlock(ds);

    // Free the drawing surface
    awt.FreeDrawingSurface(ds);
}
```

Run the example

Run the file as usual for JNI.

It's interesting to note that the AWT Native Interface requires the "jawt.dll" (or "jawt.so") to run with the application, so the easiest way to do that is copying the "jawt.dll" (should be in the .../jre/bin file path of the JDK's installation path.)

You should see a window with a rectangle drawn in it.

Congratulations! You have made your first AWT Native Application!

Native painting

As you can see, you can paint as if it is a native application. In Windows, the JVM will pass a HWND and other window information to your native application so that your application will "know" where to draw. In this example, it uses GDI to draw a Rectangle. The window information your native side need will be in a `JAWT_Win32DrawingSurfaceInfo` structure (depending on Operating System) which can be retrieved with this line:

`JAWT_Win32DrawingSurfaceInfo` structure

```
dsi_win = (JAWT_Win32DrawingSurfaceInfo*)dsi->platformInfo;
```

dsi_win has the information, look in the "jni.h" file for details.

Swings

First GUI

OK, its now time to dive into swings. Fire up your text editor or IDE and type the code as shown below.

HelloSwing.java

```
import javax.swing.JFrame;

public class HelloSwing {

    public static void main(String[] args) {

        JFrame frame = new JFrame(); //We create a new frame
        frame.setSize(200, 100); //Setting size of width=200px and height=100px
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //Frame must close when close button is pressed
        frame.setVisible(true);

    }
}
```

Output

Compile the code and run it. If you get the output as shown, you have done it all right.

How it works?

Now let's examine how it works. As in all Java programs, this program starts by creating a class, we have named this class ***HelloSwing***. This class contains the main method from where the program starts execution. The first line in this function is:

Code section

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

In the first line we create new Java Frame or JFrame. A frame is nothing but a Swing container that holds other components like labels, buttons, text fields, menu bars etc. Now since we have created a frame and before displaying it, we need to set its size. To set the size of the frame we use the following code:

Code section

```
frame.setSize(200, 100);
```

The **setSize(int *width*, int *height*)**, receives two arguments, the first one is the frame width and the second one is the frame height. We have assigned the frame size to be 200 pixels (px) wide and 100px in height. So the frame dimensions has been set successfully.

Now what will happen if someone clicks the close button in the frame? We want the program to exit and clear off the

memory when close button is clicked, hence to close the frame we use the function **setDefaultCloseOperation()**. We want the program to exit when the close button is pressed and hence we use the following code:

Code section

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

All has been done right, all we need to do now is to make the frame visible. For that we use the **setVisible(boolean *b*)** function. This function accepts one boolean input. If passed true, the frame is visible, else if passed false the frame is invisible. To make the frame visible, we use the following code:

Code section

```
frame.setVisible(true);
```

So we have created our HelloSwing program.

Diving Into Swing

Frame with Title

In the last example we saw how to create a frame that has nothing in it. This time we will create a frame that has a title. For that let's create a class `FrameWithTitle`. All the code is just the same as previous example with just one change. Notice the first statement in main method:

Code section

```
JFrame frame = new JFrame("Hello Swing");
```

Here the **new JFrame()** constructor gets a string argument. This argument forms the title of the frame. Simple!

FrameWithTitle.java

```
import javax.swing.JFrame;

public class FrameWithTitle {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Hello Swing"); // We create a new frame
        frame.setSize(200, 100); // Setting size of width=200px and height=100px
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Frame must close when closed button is presses
        frame.setVisible(true);

    }

}
```

Output

Compiling the FrameWithTitle code will give an output as shown above.

Adding Labels

We have created an empty frame, a frame with a title, now let us see how to add a simple component to a frame. Label is a simple component and we will add it to our frame. Read the program below carefully. It has a new statement when compared to the previous example.

AddingLabel.java

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class AddingLabel {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Hello Swing"); // We create a new frame
        frame.setSize(200, 100); // Setting size of width=200px and height=100px
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Frame must close when closed button is presses
        frame.add(new JLabel("Diving into swing!")); // Adding a label
        frame.setVisible(true);

    }
}
```

Output

Look at the line

Code section

```
frame.add(new JLabel("Diving into swing!"));
```

In this line we tell the computer to add a component or object to the frame using ***add()*** function. Since we want to add a label we pass a new label object using ***new JLabel()*** command.

The `new JLabel("Diving into swing!")` creates a new label. Once added, the frame is displayed using `frame.setVisible(true);`

If all had been done right, you must be getting output as shown above.

Label and Text

LabelnText.java

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class LabelnText {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Hello Swing"); // We create a new frame
        frame.setSize(200, 100); // Setting size of width=200px and height=100px
```

```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Frame must close when closed button is presses
        frame.add(new JLabel("Diving into swing!")); // Adding a label
        frame.add(new JTextField("Type something here"));// Adding text field
        frame.setVisible(true);

    }

}
```

Output

In the last example (AddingLabel.java) we have seen how to add a JLabel to a JFrame. Now we will increase our knowledge by adding more components. Lets add a text field (that's JTextField) along with label. The program shown above (LabelnText.java) has the same code of AddingLabel.java but with a minor difference. Look at the penultimate line `frame.add(new JTextField("Type something here"));`, in this line we have added a **JTextField** to frame using the **add()** method. To the add method we pass a new JTextField using **new JTextField("Type something here")** statement. The **JTextField()** constructor can accept a string argument which is set as text value of the text field.

So when we run the example we would expect the frame to have a label and a text field at the right of it. But look at the output. What's wrong? All we see is a text field that occupies the entire frame! Java has a bug? The answer is both label and text fields are present in the frame. The text field is drawn on top of the label.

Java Virtual Machine is dumb. You told to put an label, so it did put one onto the frame, next you told to put a text field so it did faithfully. But it put the text field on top of the label. To get a desired output like a label and text field **laid** next to each other we must use what is called a **Layout Manager** which you will be briefed about shortly.

Layout Manager

LayoutManagerDemo.java

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class LayoutManagerDemo {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Hello Swing"); // We create a new frame
        frame.setSize(200, 100); // Setting size of width=200px and height=100px
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Frame must close when closed button is presses
        frame.setLayout(new FlowLayout()); // Adding layout, flow layout in this case
        frame.add(new JLabel("Diving into swing!")); // Adding a label
        frame.add(new JTextField("Type something here")); // Adding text field
        frame.setVisible(true);

    }
}
```

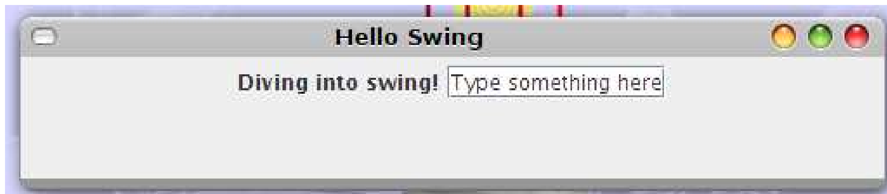
Output

Let's see what occurs if we change the frame dimension.

LayoutManagerDemo1.java

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class LayoutManagerDemo1 {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Hello Swing"); // We create a new frame
        frame.setSize(500, 100); // Setting size of width=500px and height=100px
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Frame must close when closed button is presses
        frame.setLayout(new FlowLayout()); // Adding layout, flow layout in this case
        frame.add(new JLabel("Diving into swing!")); // Adding a label
        frame.add(new JTextField("Type something here")); // Adding text field
        frame.setVisible(true);

    }
}
```

Output



Packing Frames

Packing optimize the size of the frame.

PackingFrames.java

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class PackingFrames {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Hello Swing"); // We create a new frame
        //frame.setSize(500, 100); // Setting size of width=200px and height=100px
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Frame must close when closed button is presses
        frame.setLayout(new FlowLayout());
        frame.add(new JLabel("Diving into swing!")); // Adding a label
        frame.add(new JTextField("Type something here")); // Adding text field
        frame.pack(); // Packing components into frames. This is like auto sizing of frames
        frame.setVisible(true);

```

```
    }
}
```

Output

First Examples

On this page, we'll get started with swing. We'll look at the classes `JFrame` and `JLabel`, and build a basic swing "hello world" application.

JFrames

JFrames are swing components. A swing component is a part of a graphic user interface (GUI). Frames, windows, text boxes, buttons, switches and many other parts of a GUI application are all components. The root class of all swing components is the `JComponent` class, and other swing components, including `JFrame`, are subclasses of `JComponent`. `JComponent` is an abstract class, so it cannot be instantiated directly, but can be subclassed, which is useful if you want to write your own custom GUI component.

A JFrame is a top-level component, meaning that it contains other components, but is not contained itself. It's onscreen appearance is dictated by the platform, but generally it is a window that the user can resize, move, maximize, minimize, and has its own title bar.

Probably the best way to get familiar with a `JFrame`, as well as components in general, is to run a short example program that creates one. The following program will do the trick.

A simple Swing application

```
import javax.swing.JFrame;  // All swing components live
                    // in the javax.swing package


// A simple class to test a JFrame
public class JFrameTest {

    public static void main(String[] args) {

        // create the frame.  The constructor optionally takes
                // a string as an argument that's used as the title.
        JFrame frame = new JFrame("This is a test!");

        // set the size -- 300 by 300 is good
        frame.setSize(300, 300);

        // this next line is currently commented.  If you
        // uncomment it, the user will not be able to resize
        // the JFrame

        // frame.setResizable(false);

        // has the application exit when the frame is closed
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // makes the frame visible.  The frame is invisible
```

```
        // until this method is called.
        frame.setVisible(true);

        // these lines are also commented, but you can uncomment
        // them to see their effects.  The first will center the
        // JFrame in the center of the screen.  The second will
        // make it the front, focused window.

        // frame.setLocationRelativeTo(null);
        // frame.toFront();
    }
}
```

The code creates an empty JFrame. When you run it, you should see a window appear on your screen. The code also illustrates some methods that can be used with JFrames, some of which are commented out. I would encourage you to experiment a bit with commenting and uncommenting these, and seeing the effect.

JLabels

An empty frame is only so exciting, so for this section, we're going to look at JLabels, and put one in a JFrame. A `JLabel` is a GUI component that can hold an image, some text, or both.

The following code creates a GUI version of the "Hello World!" application.

Hello world in Swing

```
import javax.swing.*;  // All swing components live
               // in the javax.swing package

// A GUI hello world application
public class Hello {

    public static void main(String[] args) {

        // creates the label.  The JLabel constructor
        // takes an optional argument which sets the text
        // of the label.
        JLabel label = new JLabel("Hello, World!");

        // The next line is commented out.  If it is
        // uncommented, the text will be aligned with
        // the center of the frame, otherwise it will
        // align on the left.

        // label.setHorizontalAlignment(SwingConstants.CENTER);

        JFrame frame = new JFrame("Hello");
        frame.add(label);

        frame.setSize(300, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
        frame.setLocationRelativeTo(null);
        frame.toFront();
    }
}
```

This program is pretty much like the previous one, except that it creates a `JLabel` with the text "Hello, World!" in addition to the `JFrame`, and uses the `JFrame add()` method to add it to the frame before making it visible. If you compile and run the program as-is, the "Hello, world" text will be aligned on the left side of the window.

By uncommenting the line `label.setHorizontalAlignment(SwingConstants.CENTER);` you can move the text to the center.

Summary

We've seen how to create a JFrame, a top level container for a swing GUI application, as well as a few `JFrame` methods that can be used to change its settings. We also looked at the `JLabel` class, and used the `JFrame`'s `add()` method to add a `JLabel` to the frame in order to create a simple application.

Looking at these examples, you might be wondering what other methods exist for JFrames, JLabels, and other components. Although we'll continue to look at more components and more methods. Fortunately, Oracle provides an invaluable online reference to all of the core Java classes, including swing, which can be viewed at the following link:

http://docs.oracle.com/javase/7/docs/api/index.html

Swing Top Level Containers

JFrame

JFrameEx.java

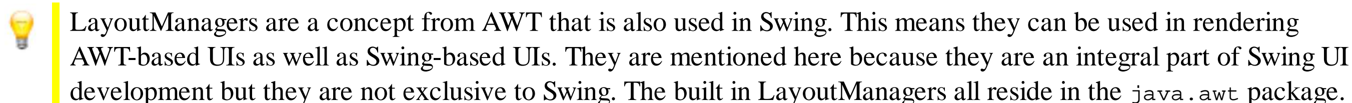
```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class JFrameEx extends JFrame {

    JFrameEx() {
        JLabel label1 = new JLabel("My Label");
        add(label1);
        pack();
        setVisible(true);
    }

    public static void main(String args[]) {
        JFrameEx e1 = new JFrameEx();
    }
}
```

JOptionPane

JApplet

Swing Layouts

The layout of components is dictated by the layout manager used. There are 7 layout managers built into Java. Most UIs are built using some combination of them, typically by nesting layout managers. The most commonly used layouts are FlowLayout, BorderLayout and BoxLayout.



LayoutManagers are a concept from AWT that is also used in Swing. This means they can be used in rendering AWT-based UIs as well as Swing-based UIs. They are mentioned here because they are an integral part of Swing UI development but they are not exclusive to Swing. The built in LayoutManagers all reside in the `java.awt` package.

BorderLayout

- Used to arrange components around a center panel.
- Sizes components to use all available space for their region.

- North / South components will be as tall as their preferred height and as wide as the width of the component they're in.
- East / West components will be as wide as their preferred width and as tall as the height of the component they're in minus the preferred heights of the North and South components.
- Center will use all remaining space in the component after the North / South / East / West components have been sized.

BorderLayout

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import java.awt.BorderLayout;
import java.awt.Container;

public class BorderLayoutDemo {
    public static void main(String[] args) {
/*
        Swing events run in the EventDispatchThread. All swing components
        and models must be modified only from the EventDispatchThread.
        SwingUtilities.invokeLater(...) and SwingUtilities.invokeAndWait(...)
        both will execute a Runnable in the EDT.
*/
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JFrame borderLayoutFrame = new JFrame("Border Layout");
/*
                You shouldn't add components directly to a JFrame. As of Java 5 (1.5?)
                you're allowed to but it won't be backwards compatible. Add to the
                contentPane instead.
*/
                Container contentPane = borderLayoutFrame.getContentPane();
                contentPane.setLayout(new BorderLayout());
                contentPane.add(new JButton("East Button"), BorderLayout.EAST);
                contentPane.add(new JButton("West Button"), BorderLayout.WEST);
                contentPane.add(new JButton("North Button"), BorderLayout.NORTH);
                contentPane.add(new JButton("South Button"), BorderLayout.SOUTH);
                contentPane.add(new JButton("Center"), BorderLayout.CENTER);

                borderLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                borderLayoutFrame.setSize(400, 200);
                borderLayoutFrame.setVisible(true);
            }
        });
    }
}
```

FlowLayout

- Used to arrange components in a straight horizontal line.
- If there is not enough space for all the components to fit, they'll be moved to the next line.
- Components are set at their preferred sizes.

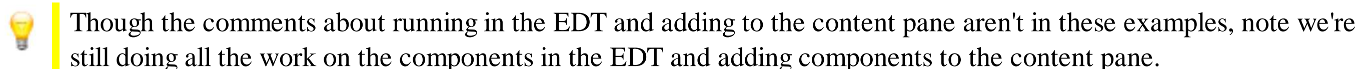
FlowLayout

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.FlowLayout;

public class FlowLayoutDemo {
    public static void main(String[] args) {
/*
```

```
        Swing events run in the EventDispatchThread. All swing components
        and models must be modified only from the EventDispatchThread.
        SwingUtilities.invokeLater(...) and SwingUtilities.invokeAndWait(...)
        both will execute a Runnable in the EDT.
*/
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                JFrame flowLayoutFrame = new JFrame("Flow Layout");
/*
                You shouldn't add components directly to a JFrame. As of Java 5 (1.5?)
                you're allowed to but it won't be backwards compatible. Add to the
                contentPane instead.
*/
                Container contentPane = flowLayoutFrame.getContentPane();
                contentPane.setLayout(new FlowLayout());
                contentPane.add(new JButton("Button A"));

                // buttonB will not be resized smaller than 200x40 pixels
                JButton buttonB = new JButton("Button B");
                buttonB.setPreferredSize(new Dimension(200, 40));
                contentPane.add(buttonB);
                contentPane.add(new JButton("Button C"));
                contentPane.add(new JButton("Button D"));

                flowLayoutFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                flowLayoutFrame.setSize(400, 200);
                flowLayoutFrame.setVisible(true);
            }
        });
    }
}
```

BoxLayout

- Used to arrange components vertically or horizontally. `BoxLayout` is essentially a replacement for `FlowLayout` that is more powerful if thats what you're going for. However with great power comes great responsibility... or something like that. The trade off for having more power is you have to do a bit more work, but you can get things to look closer to what you really want. To demonstrate we'll go through several examples.



Though the comments about running in the EDT and adding to the content pane aren't in these examples, note we're still doing all the work on the components in the EDT and adding components to the content pane.

`BoxLayout`s come in two basic flavors. Vertical or horizontal. These are represented by the constants `BoxLayout.X_AXIS`, `BoxLayout.Y_AXIS`, `BoxLayout.LINE_AXIS`, `BoxLayout.PAGE_AXIS`. X/Y_AXIS specify to always layout the components along the X or Y axis of the container. The components are then arranged left to right (X_AXIS) or top to bottom (Y_AXIS). LINE/PAGE_AXIS will do exactly the same thing as X/Y_AXIS *assuming* the `ComponentOrientation` field of the parent container is left to right oriented. If not, LINE_AXIS will add components from right to left, and PAGE_AXIS will by default align components on the right side of the panel. If you are planning to support locale specific layouts, PAGE_AXIS is the one for you.

Talking about alignments is fun, but lets see a demo. The below demo will create a main window which will allow you to create child windows laid out with the proper alignments.

BoxLayout

```
import javax.swing.BoxLayout;
import javax.swing.ButtonGroup;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JRadioButton;
import javax.swing.JSeparator;
import javax.swing.SwingUtilities;
```

```
import java.awt.Component;
import java.awt.ComponentOrientation;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class BoxLayoutDemo {
    private int alignment;
    private ComponentOrientation orientation;

    private static final String X_AXIS = "X_AXIS";
    private static final String Y_AXIS = "Y_AXIS";
    private static final String LINE_AXIS = "LINE_AXIS";
    private static final String PAGE_AXIS = "PAGE_AXIS";

    private static final String LEFT_TO_RIGHT = "Left to Right";
    private static final String RIGHT_TO_LEFT = "Right to Left";
    private static final String DEFAULT_ORIENTATION = "Default Orientation";

    private ActionListener alignmentListener = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if (X_AXIS.equals(e.getActionCommand())) {
                alignment = BoxLayout.X_AXIS;
            } else if (Y_AXIS.equals(e.getActionCommand())) {
                alignment = BoxLayout.Y_AXIS;
            } else if (LINE_AXIS.equals(e.getActionCommand())) {
                alignment = BoxLayout.LINE_AXIS;
            } else if (PAGE_AXIS.equals(e.getActionCommand())) {
                alignment = BoxLayout.PAGE_AXIS;
            } else if (LEFT_TO_RIGHT.equals(e.getActionCommand())){
                orientation = ComponentOrientation.LEFT_TO_RIGHT;
            } else if (RIGHT_TO_LEFT.equals(e.getActionCommand())){
                orientation = ComponentOrientation.RIGHT_TO_LEFT;
            } else if (DEFAULT_ORIENTATION.equals(e.getActionCommand())){
                orientation = null;
            }
        }
    };

    public JFrame buildMainFrame(){
        JFrame boxLayoutDemoFrame = new JFrame("BoxLayoutFrame");
        boxLayoutDemoFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Container contentPane = boxLayoutDemoFrame.getContentPane();
        contentPane.setLayout(new BoxLayout(contentPane, BoxLayout.Y_AXIS));

        ButtonGroup alignmentGroup = new ButtonGroup();

        JRadioButton xAxisButton = buildAlignmentButton(alignmentGroup, X_AXIS);
        xAxisButton.setSelected(true);
        JRadioButton yAxisButton = buildAlignmentButton(alignmentGroup, Y_AXIS);
        JRadioButton lineAxisButton = buildAlignmentButton(alignmentGroup, LINE_AXIS);
        JRadioButton pageAxisButton = buildAlignmentButton(alignmentGroup, PAGE_AXIS);

        final ButtonGroup orientationGroup = new ButtonGroup();
        JRadioButton defaultOrientation = buildAlignmentButton(orientationGroup, DEFAULT_ORIENTATION);
        defaultOrientation.setSelected(true);
        JRadioButton leftToRight = buildAlignmentButton(orientationGroup, LEFT_TO_RIGHT);
        JRadioButton rightToLeft = buildAlignmentButton(orientationGroup, RIGHT_TO_LEFT);

        JButton openFrameButton = new JButton("Build window");
        openFrameButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JFrame demoFrame = new JFrame("Demo Frame");

                Container contentPane = demoFrame.getContentPane();
                contentPane.setLayout(new BoxLayout(contentPane, alignment));
                if(orientation != null){
                    contentPane.setComponentOrientation(orientation);
                }
```

```
                JButton buttonA;
                JButton buttonB;
                JButton buttonC;
                JButton buttonD;

                if(alignment == BoxLayout.PAGE_AXIS || alignment == BoxLayout.Y_AXIS){
                    buttonA = new JButton("Left Aligned");
                    buttonA.setAlignmentX(JComponent.LEFT_ALIGNMENT);
                    buttonB = new JButton("Right Aligned");
                    buttonB.setAlignmentX(JComponent.RIGHT_ALIGNMENT);
                    buttonC = new JButton("Center Aligned");
                    buttonC.setAlignmentX(JComponent.CENTER_ALIGNMENT);
                    buttonD = new JButton("Default Aligned");
                } else {
                    buttonA = new JButton("Top Aligned");
                    buttonA.setAlignmentY(Component.TOP_ALIGNMENT);
                    buttonB = new JButton("Bottom Aligned");
                    buttonB.setAlignmentY(Component.BOTTOM_ALIGNMENT);
                    buttonC = new JButton("Center Aligned");
                    buttonC.setAlignmentY(Component.CENTER_ALIGNMENT);
                    buttonD = new JButton("Default Aligned");
                }

                contentPane.add(buttonA);
                contentPane.add(buttonB);
                contentPane.add(buttonC);
                contentPane.add(buttonD);

                demoFrame.pack();
                demoFrame.setVisible(true);
            }
        });

        contentPane.add(new JLabel("Alignment Options"));
        contentPane.add(xAxisButton);
        contentPane.add(yAxisButton);
        contentPane.add(lineAxisButton);
        contentPane.add(pageAxisButton);

        contentPane.add(new JSeparator());
        contentPane.add(new JLabel("Orientation"));
        contentPane.add(defaultOrientation);
        contentPane.add(leftToRight);
        contentPane.add(rightToLeft);

        contentPane.add(new JSeparator());
        contentPane.add(openFrameButton);
        boxLayoutDemoFrame.setSize(250, 300);

        return boxLayoutDemoFrame;
    }

    private JRadioButton buildAlignmentButton(ButtonGroup alignmentGroup, String alignmentName) {
        JRadioButton xAxisButton = new JRadioButton(alignmentName, false);
        xAxisButton.setActionCommand(alignmentName);
        xAxisButton.addActionListener(alignmentListener);
        alignmentGroup.add(xAxisButton);
        return xAxisButton;
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                BoxLayoutDemo demo = new BoxLayoutDemo();
                JFrame boxLayoutDemoFrame = demo.buildMainFrame();
                boxLayoutDemoFrame.setVisible(true);
            }
        });
    }
}
```

CardLayout

- Used to show one component at a time from a set of components. It is like a deck of flash cards, where you show one of the cards at a time, hence the name. You can quickly change which component is shown at any time, e.g., when the user clicks a button.

GridLayout

- Used to arrange components in an evenly spaced grid.

GridLayout

```
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.SwingConstants;
import javax.swing.SwingUtilities;

public class GridLayoutDemo extends JFrame {

    private static final long serialVersionUID = 1L;

    public GridLayoutDemo() {
        /*
         * Calls the constructor in superclass.
         */
        super("GridLayout Demo");

        /*
         * Sets the close operation to DISPOSE_ON_CLOSE.
         */
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        /*
         * Creates a new GridLayout object. It has 2 rows and any number of
         * columns depending of the component count on container. The horizontal
         * and vertical gaps are 10 and 15, respectively.
         */
        GridLayout layout = new GridLayout(2, 0, 10, 15);

        /*
         * The default layout manager for the JFrame's content pane is
         * BorderLayout. So, we must to change it.
         */
        setLayout(layout);

        /*
         * Just add the new components.
         */
        add(new JButton("Button 1"));
        add(new JButton("Button 2"));
        add(new JLabel("Label 1", SwingConstants.CENTER));
        add(new JCheckBox("CheckBox 1"));
        add(new JLabel("Label 2", SwingConstants.CENTER));
        add(new JLabel("Label 3", SwingConstants.CENTER));
        add(new JButton("Button 3"));

        /*
         * Adjusts the windows size, centers it in screen and makes it visible.
         */
        pack();
        setLocationRelativeTo(null);
        setVisible(true);
    }
```

```
    public static void main(String[] args) {
        /*
         * Creates a new instance of GridLayoutDemo. Try to resize the window
         * and see the results.
         */
        SwingUtilities.invokeLater(new Runnable() {

            @Override
            public void run() {
                new GridLayoutDemo();
            }

        });

    }
}
```

GridBagLayout

SpringLayout

Event Handling

On the last page, we created an application that passively displays a "Hello, world" message. However, that gets boring quickly. In this chapter, we'll talk about swing events and look at the `JButton` class, and we'll build a small application that includes a `JButton` with a `JLabel`, and use it to introduce the concept of Layouts.

Events

Events are a fundamental part of how swing works. Various actions on the part of the user cause components to *fire* various types of events. Other Java objects can register as listeners of events with any components whose events they are interested in. In order to be a listener, the object must implement the appropriate listener interface (ie, `ActionListener` for action events, `FocusListener` for focus events, etc.), which contains methods which are called when the event is *fired*.

In Java, events themselves are objects, and instances of event classes. The event classes are located in the packages `java.awt.event` and `javax.swing.event`. A few common classes of events are:

- `ActionEvent` — a (somewhat generic) event fired by some objects, including JButtons
- `MouseEvent` — fired when the user does something with the mouse, such as a click, drag, or more general motion
- `KeyEvent` — fired when the user presses, releases, or more generally *types* a key on the keyboard

In order to see how events are used, we'll look at JButtons so that we can build an example.

JButtons

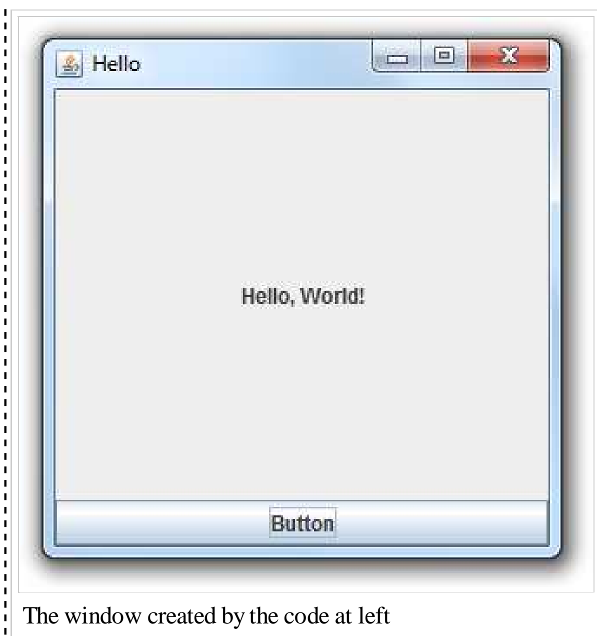
A JButton is a component that, like a JLabel, can hold text, images, or both. However, as we'd expect, it also has the property that the user can "push" it by clicking on it, which causes it to fire an event (specifically, an action event). In this example, we'll more or less redo our previous example, except that we'll add a JButton. In the next section, we'll add functionality so that pressing the button does something.

Here's the code that creates the previous "Hello, world" application, plus a button at the bottom of the screen:

A Swing application with buttons

```
import javax.swing.*;
import java.awt.BorderLayout; // The BorderLayout class
                  // lives in java.awt.  We
                  // use its constants to tell
```

```java
                  // the JFrame where we want the
                  // components to go.

// expanded version of the hello application,
// but with a button below.
public class HelloWithButton {

    public static void main(String[] args) {

        // construct the JLabel,
        // an overloaded version of the constructor
        // allows us to specify the alignment off the
        // bat
        JLabel label = new JLabel
            ("Hello, World!", SwingConstants.CENTER);

        // create the button. The text in the
        // constructor will be visible on the
        // button.
        JButton button = new JButton("Button");

        // create the frame
        JFrame frame = new JFrame("Hello");

        // add the label and the button to the
        // frame, using layout constants.
        frame.add(label, BorderLayout.CENTER);
        frame.add(button, BorderLayout.SOUTH);

        frame.setSize(300, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
        frame.setLocationRelativeTo(null);
        frame.toFront();
    }
}
```



The window created by the code at left

What you should see is a frame appear that contains the text "Hello, World!" in the center, along with a button labeled "Button" at the bottom. You can click on the button, but it doesn't do anything.

In this example, we not only added the components to the frame, but specified where they should go. Every component which contains other components has a `LayoutManager` object, which controls how the components are laid out, and different layout managers lay out their components differently. A few examples of layout managers are `GridLayout`, `CardLayout`, and `BorderLayout`. This last is the default layout manager of a JFrame (although we can change the layout manager of a component by using its `setLayout()` method). Although layout managers are covered in more detail in a later chapter, we'll say here that BorderLayout managers divide their space into five regions, and each is associated with a constant in the `BorderLayout` class. The constants (which explain the regions) are:

- `BorderLayout.NORTH`,
- `BorderLayout.SOUTH`,
- `BorderLayout.EAST`,
- `BorderLayout.WEST` and
- `BorderLayout.CENTER`.

If you're adding a component to a container that is using a BorderLayout, you can pass one of these constants as an optional second parameter to the `add()` method to specify where the component should be placed. If no constant is specified, the layout manager by default places the new component in the center. I would encourage you to experiment with the previous example and change the layout parameters, and observe the effect.

MVC

This chapter explains how to separate Swing applications into a more maintainable triad of components: the domain model, the view of the user interface, and the controller that oversees these. A minimal class structure is shown below.

600px

The main application class is the controller: it creates a model object (which is Observable), and a view object (which is an Observer of the model). The controller adds the view as an observer of the model.

When the model changes, it calls `setChanged()` and then `notifyObservers()`. This means that the model can react to user input.

The controller doesn't touch the view (apart from instantiating it and giving it the model).

The view has access to the model, including to change it.

Content

- Java Swing
- Overview
- AWT
- Dive into Swings
- First Examples
- Swing Top Level Containers
- Swing Layouts
- Event Handling
- MVC architecture
- Responsive Swing Application
- Large Examples

Appendix

- Reference

Responsive Swing Application

In this section, we'll create an application that responds to a user action, namely the pressing of a button.

The Program

Last section, we wrote a program that displays a text label along with a button. In this section, we'll add functionality to the application so that the text changes when the user presses the button. In order to accomplish this, we'll write a subclass of `JLabel` that can change its text, and implements the `ActionListener` interface. Because a JButton generates action events when pushed, we can connect our specialized JLabel to the JButton by registering it as a listener. Here's the code:

A new listener

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.SwingConstants;
import javax.swing.JLabel;

// ActionEvent and ActionListener
// live in the java.awt.event
// package

class ChangingLabel extends JLabel implements ActionListener {

    // String constants which define the text that's displayed.
    private static final String TEXT1 = "Hello, World!";
    private static final String TEXT2 = "Hello again!";

    // Instance field to keep track of the text currently
    // displayed
    private boolean usingText1;

    // Constructor.  Sets alignment and initial text, and
    // sets the usingText1 field appropriately.
    public ChangingLabel() {
        this.setHorizontalAlignment(SwingConstants.CENTER);
        this.setText(TEXT1);
        usingText1 = true;
    }

    // A method to change the label text.  If the
    // current text is TEXT1, changes it to TEXT2,
```

```java
    // and vice-versa.  The method is private here,
    // because it is never called directly.
    private void changeLabel() {
        if(usingText1)
            this.setText(TEXT2);
        else
            this.setText(TEXT1);

                usingText1 = !usingText1;
    }

    // This method implements the ActionListener interface.
    // Any time that an action is performed by a component
    // that this object is registered with, this method
    // is called.  We can analyze the event object received
    // here for more information if we want to, but it's not
    // necessary in this application.
    public void actionPerformed(ActionEvent e) {
        this.changeLabel();
    }
}
```

A Swing application with an active button

```java
import java.awt.BorderLayout;

import javax.swing.JButton;
import javax.swing.JFrame;

// Application with a changing text field
public class HelloWithButton2 {

        public static void main(String[] args) {

                // Constructs the ChangingLabel.  All the
            // intialization is done be the default
            // constructor, defined in the ChangingLabel
            // class below.
                ChangingLabel changingLabel = new ChangingLabel();

                // Create the button
                JButton button = new JButton("Button");

                // Register the ChangingLabel as an action
                // listener to the button.  Whenever the
                // button is pressed, its ActionEvent will
                // be sent the ChangingLabel's
                // actionPerformed() method, and the code
                // there will be executed.
                button.addActionListener(changingLabel);

                // Create the frame
                JFrame frame = new JFrame("Hello");

                // Add the label and the button to the
                // frame, using layout constants.
                frame.add(changingLabel, BorderLayout.CENTER);
                frame.add(button, BorderLayout.SOUTH);

                frame.setSize(300, 300);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.setVisible(true);
                frame.setLocationRelativeTo(null);
                frame.toFront();
        }
}
```

In this application, class `ChangingLabel` defines a special type of label that is an action listener (ie, that can receive ActionEvents) by implementing the `ActionListener` interface. It also has the ability to change its text through the private

method `changeLabel()`. The `actionPerformed()` method just calls `changeLabel()` when it receives an ActionEvent.

The main program code in `HelloWithButton2` is similar to the code in the last section, but uses a `ChangeLabel` instead of an ordinary `JLabel` object. It also registers the ChangeLabel as an `ActionListener` of the button with the button's `addActionListener()` method. We can do this because JButtons create ActionEvents, and ChangeLabel is an ActionListener (ie, implements the `ActionListener` interface).

There's one thing to notice in the above example. Two classes are defined, one to contain the program code in the static main() method, and another that defines a type used in the program. However, there's no reason that we can't put the `main()` method directly in our new type's class. In fact, putting the program's `main()` method in a customized subclass of a component is a common idiom in programming Swing applications. When reorganized this way, the program looks like this:

The merged class

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

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

// ActionEvent and ActionListener
// live in the java.awt.event
// package

// Application with a changing text field
public class ChangingLabel extends JLabel implements ActionListener {

        // String constants which define the text that's displayed.
        private static final String TEXT1 = "Hello, World!";
        private static final String TEXT2 = "Hello again!";

        // Instance field to keep track of the text currently
        // displayed
        private boolean usingText1;

        // Constructor.  Sets alignment and initial text, and
        // sets the usingText1 field appropriately.
        public ChangingLabel() {
                this.setHorizontalAlignment(SwingConstants.CENTER);
                this.setText(TEXT1);
                usingText1 = true;
        }

        // A method to change the label text.  If the
        // current text is TEXT1, changes it to TEXT2,
        // and vice-versa.  The method is private here,
        // because it is never called directly.
        private void changeLabel() {
                if(usingText1) {
                        this.setText(TEXT2);
                        usingText1 = false;
                } else {
                        this.setText(TEXT1);
                        usingText1 = true;
                }
        }

        // This method implements the ActionListener interface.
        // Any time that an action is performed by a component
        // that this object is registered with, this method
        // is called.  We can analyze the event object received
        // here for more information if we want to, but it's not
        // necessary in this application.
        public void actionPerformed(ActionEvent e) {
                this.changeLabel();
        }
```

```java
        // Main method.  This is the code that is executed when the
        // program is run
        public static void main(String[] args) {

            // Constructs the ChangingLabel.  All the
            // intialization is done be the default
            // constructor, defined in the ChangingLabel
            // class below.
            ChangingLabel changingLabel = new ChangingLabel();

            // Create the button
            JButton button = new JButton("Button");

            // Register the ChangingLabel as an action
            // listener to the button.  Whenever the
            // button is pressed, its ActionEvent will
            // be sent the ChangingLabel's
            // actionPerformed() method, and the code
            // there will be executed.
            button.addActionListener(changingLabel);

            // Create the frame
            JFrame frame = new JFrame("Hello");

            // Add the label and the button to the
            // frame, using layout constants.
            frame.add(changingLabel, BorderLayout.CENTER);
            frame.add(button, BorderLayout.SOUTH);

            frame.setSize(300, 300);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
            frame.setLocationRelativeTo(null);
            frame.toFront();
    }
}
```

This program does exactly the same thing as the first, but organizes all the code into one class.

Multi-threading in Swing

The thread that performs actions inside the action, change, mouse and other listeners is called a *Swing thread*. These methods must execute and return quickly, as the Swing thread that must be available for the Swing framework to keep the application responsive. In other words, if you perform some time-consuming calculation or other activities, you must do this in a separate thread, not in the Swing thread. However then you access Swing components from the non-swing thread you initiated yourself, they may hang due race conditions if manipulated by the Swing thread at the same time. Because of that, Swing component from the non-Swing thread must be accessed through intermediate Runnable, using the `SwingUtilities.invokeAndWait()` method:

Multi-threading

```java
String myMessage = "abc";
final String message = myMessage; // Making final allows to access from inner class here.
SwingUtilities.invokeAndWait(new Runnable {
  public void run() {
    myLabel.setText(message);
  }
})
```

This call will fire *setText(message)* in the appropriate time, when the Swing thread will be ready to respond. This approach is also usual when implementing the moving progress bar that only works properly if the slow process itself is implemented outside the Swing thread.

Large Examples

Calculator

The example below builds a user interface using multiple nested layout managers, applies listeners and client properties to components, and demonstrates how to create components from the Event Dispatch Thread (EDT).

Main Method

The main method is pretty straight forward.

Main method

```
public static void main(String[] args) {
    // Remember, all swing components must be accessed from
    // the event dispatch thread.
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            Calculator calc = new Calculator();
            calc.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            calc.setVisible(true);
        }
    });
}
```

It creates a new calculator, sets it to exit the program when its closed and then makes it visible. The interesting bit about this code is that it doesn't happen in the main thread. It happens in the event dispatch thread. While in a simple application like this it's enough to simply wrap the whole component creation in an `invokeLater()` to push it into the EDT, larger applications need to be careful that they only access UI components from the EDT. Not doing so can cause mysterious deadlocks in applications.

Constructor

The constructor is where the calculator is laid out.

The constructor

```
public Calculator() {
    super("Calculator");

    JPanel mainPanel = new JPanel(new BorderLayout());
    JPanel numberPanel = buildNumberPanel();
    JPanel operatorPanel = buildOperatorPanel();
    JPanel clearPanel = buildClearPanel();
    lcdDisplay = new JTextArea();
    mainPanel.add(clearPanel, BorderLayout.SOUTH);
    mainPanel.add(numberPanel, BorderLayout.CENTER);
    mainPanel.add(operatorPanel, BorderLayout.EAST);
    mainPanel.add(lcdDisplay, BorderLayout.NORTH);

    errorDisplay = new JLabel(" ");
    errorDisplay.setFont(new Font("Dialog", Font.BOLD, 12));

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add(mainPanel, BorderLayout.CENTER);
    getContentPane().add(errorDisplay, BorderLayout.SOUTH);

    pack();
    resetState();
}
```

This approach uses inheritance for defining the calculator. Another way of doing it would have been to have a `getComponent()` method which would return a panel laid out as desired. The calculator would then be a controller dictating the behavior of the panels and own the panel all the components were on. While more complex, that method also scales more easily to larger, more complicated UIs. Inheritance has the problem of child classes having the ability to override methods

that are currently being called in the constructor of the parent class. If the overridden methods required a field in the child class to be already initialized, you're in trouble. At the time that the method would be called in the parent, the child hasn't been fully initialized yet.

The constructor uses two BorderLayouts to do the layout. The first is for the main panel, the second is to compose the main panel with the error message panel. Also, each of the subpanels in the main panel are built with their own layout managers. This is a good example of building a more complicated UI as a composite of multiple simpler sub-panels.

Listeners

Here we start getting into the fun bits. For each of the buttons we want something to listen for the pushed button. The `ActionListener` class does just that.

The listeners

```
private final ActionListener numberListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JComponent source = (JComponent)e.getSource();
        Integer number = (Integer) source.getClientProperty(NUMBER_PROPERTY);
        if (number == null){
            throw new IllegalStateException("No NUMBER_PROPERTY on component");
        }

        numberButtonPressed(number.intValue());
    }
};

private final ActionListener decimalListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        decimalButtonPressed();
    }
};

private final ActionListener operatorListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JComponent source = (JComponent) e.getSource();
        Integer opCode = (Integer) source.getClientProperty(OPERATOR_PROPERTY);
        if (opCode == null) {
            throw new IllegalStateException("No OPERATOR_PROPERTY on component");
        }

        operatorButtonPressed(opCode);
    }
};

private final ActionListener clearListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        resetState();
    }
};
```

We setup a separate listener for each type of button. Numbers, Decimals, Operators and Clear buttons. Each one processes the message associated with that class of button and only that class of button. While that's fine, how do you determine which button in that class was pressed? Well, Swing provides a way of setting arbitrary properties on objects. This is accomplished by the `getClientProperty putClientProperty` pair. Below, we tell the difference between the number buttons by which number is associated with the `clientProperty` of the button. Similarly the `operatorListener` does the same, but pulling out its own property.

Building the buttons

Here we can see how the listeners get assigned to the buttons.

The buttons

```
private JButton buildNumberButton(int number) {
    JButton button = new JButton(Integer.toString(number));
    button.putClientProperty(NUMBER_PROPERTY, Integer.valueOf(number));
    button.addActionListener(numberListener);
    return button;
}

private JButton buildOperatorButton(String symbol, int opType) {
    JButton plus = new JButton(symbol);
```

```
    plus.putClientProperty(OPERATOR_PROPERTY, Integer.valueOf(opType));
    plus.addActionListener(operatorListener);
    return plus;
}
```

First we set the label for the buttons by passing a `String` to the constructor of the JButton, then we set the property value that we want using `putClientProperty` then we add the appropriate action listener to the button. Action listeners are activated when the component takes its intended action. For a `JButton` that means being pressed.

Building the panels

Here we show building the number panel.

The panels

```
public JPanel buildNumberPanel() {
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(4, 3));

    panel.add(buildNumberButton(1));
    panel.add(buildNumberButton(2));
    panel.add(buildNumberButton(3));
    panel.add(buildNumberButton(4));
    panel.add(buildNumberButton(5));
    panel.add(buildNumberButton(6));
    panel.add(buildNumberButton(7));
    panel.add(buildNumberButton(8));
    panel.add(buildNumberButton(9));

    JButton buttonDec = new JButton(".");
    buttonDec.addActionListener(decimalListener);
    panel.add(buttonDec);

    panel.add(buildNumberButton(0));

    // Exit button is to close the calculator and terminate the program.
    JButton buttonExit = new JButton("EXIT");
    buttonExit.setMnemonic(KeyEvent.VK_C);
    buttonExit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            System.exit(0);
        }
    });
    panel.add(buttonExit);
    return panel;

}
```

We want an even grid for the number keys and want them all to be sized the same so we use a `GridLayout` as our LayoutManager. This grid is 4 rows deep and 3 columns wide. Building the operator panel and the clear panel are both done similarly.

Full code for the Calculator

The Calculator

```
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;
import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;

public class Calculator extends JFrame {
```

```
    private static final String NUMBER_PROPERTY = "NUMBER_PROPERTY";
    private static final String OPERATOR_PROPERTY = "OPERATOR_PROPERTY";
    private static final String FIRST = "FIRST";
    private static final String VALID = "VALID";

    // These would be much better if placed in an enum,
    // but enums are only available starting in Java 5.
    // Code using them isn't back portable.
    private static interface Operator{
        static final int EQUALS = 0;
        static final int PLUS = 1;
        static final int MINUS = 2;
        static final int MULTIPLY = 3;
        static final int DIVIDE = 4;
    }

    private String status;
    private int previousOperation;
    private double lastValue;
    private JTextArea lcdDisplay;
    private JLabel errorDisplay;

    public static void main(String[] args) {
        // Remember, all swing components must be accessed from
        // the event dispatch thread.
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                Calculator calc = new Calculator();
                calc.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                calc.setVisible(true);
            }
        });
    }

    public Calculator() {
        super("Calculator");

        JPanel mainPanel = new JPanel(new BorderLayout());
        JPanel numberPanel = buildNumberPanel();
        JPanel operatorPanel = buildOperatorPanel();
        JPanel clearPanel = buildClearPanel();
        lcdDisplay = new JTextArea();
        lcdDisplay.setFont(new Font("Dialog", Font.BOLD, 18));
        mainPanel.add(clearPanel, BorderLayout.SOUTH);
        mainPanel.add(numberPanel, BorderLayout.CENTER);
        mainPanel.add(operatorPanel, BorderLayout.EAST);
        mainPanel.add(lcdDisplay, BorderLayout.NORTH);

        errorDisplay = new JLabel(" ");
        errorDisplay.setFont(new Font("Dialog", Font.BOLD, 12));

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(mainPanel, BorderLayout.CENTER);
        getContentPane().add(errorDisplay, BorderLayout.SOUTH);

        pack();
        resetState();
    }

    private final ActionListener numberListener = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JComponent source = (JComponent)e.getSource();
            Integer number = (Integer) source.getClientProperty(NUMBER_PROPERTY);
            if(number == null){
                throw new IllegalStateException("No NUMBER_PROPERTY on component");
            }

            numberButtonPressed(number.intValue());
        }
    };

    private final ActionListener decimalListener = new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e) {
            decimalButtonPressed();
        }
    };

    private final ActionListener operatorListener = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            JComponent source = (JComponent) e.getSource();
            Integer opCode = (Integer) source.getClientProperty(OPERATOR_PROPERTY);
            if (opCode == null) {
                throw new IllegalStateException("No OPERATOR_PROPERTY on component");
            }

            operatorButtonPressed(opCode);
        }
    };

    private final ActionListener clearListener = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            resetState();
        }
    };

    private JButton buildNumberButton(int number) {
        JButton button = new JButton(Integer.toString(number));
        button.putClientProperty(NUMBER_PROPERTY, Integer.valueOf(number));
        button.addActionListener(numberListener);
        return button;
    }

    private JButton buildOperatorButton(String symbol, int opType) {
        JButton plus = new JButton(symbol);
        plus.putClientProperty(OPERATOR_PROPERTY, Integer.valueOf(opType));
        plus.addActionListener(operatorListener);
        return plus;
    }

    public JPanel buildNumberPanel() {
        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(4, 3));

        panel.add(buildNumberButton(7));
        panel.add(buildNumberButton(8));
        panel.add(buildNumberButton(9));
        panel.add(buildNumberButton(4));
        panel.add(buildNumberButton(5));
        panel.add(buildNumberButton(6));
        panel.add(buildNumberButton(1));
        panel.add(buildNumberButton(2));
        panel.add(buildNumberButton(3));

        JButton buttonDec = new JButton(".");
        buttonDec.addActionListener(decimalListener);
        panel.add(buttonDec);

        panel.add(buildNumberButton(0));

        // Exit button is to close the calculator and terminate the program.
        JButton buttonExit = new JButton("EXIT");
        buttonExit.setMnemonic(KeyEvent.VK_C);
        buttonExit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
        panel.add(buttonExit);
        return panel;

    }

    public JPanel buildOperatorPanel() {
        JPanel panel = new JPanel();
```

```
        panel.setLayout(new GridLayout(4, 1));

        panel.add(buildOperatorButton("+", Operator.PLUS));
        panel.add(buildOperatorButton("-", Operator.MINUS));
        panel.add(buildOperatorButton("*", Operator.MULTIPLY));
        panel.add(buildOperatorButton("/", Operator.DIVIDE));
        return panel;
    }

    public JPanel buildClearPanel() {
        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(1, 3));

        JButton clear = new JButton("C");
        clear.addActionListener(clearListener);
        panel.add(clear);

        JButton CEntry = new JButton("CE");
        CEntry.addActionListener(clearListener);
        panel.add(CEntry);

        panel.add(buildOperatorButton("=", Operator.EQUALS));

        return panel;
    }

    public void numberButtonPressed(int i) {
        String displayText = lcdDisplay.getText();
        String valueString = Integer.toString(i);

        if (("0".equals(displayText)) || (FIRST.equals(status))) {
            displayText = "";
        }

        int maxLength = (displayText.indexOf(".") >= 0) ? 21 : 20;
        if(displayText.length() + valueString.length() <= maxLength){
            displayText += valueString;
            clearError();
        } else {
            setError("Reached the 20 digit max");
        }

        lcdDisplay.setText(displayText);
        status = VALID;
    }

    public void operatorButtonPressed(int newOperation) {
        Double displayValue = Double.valueOf(lcdDisplay.getText());

//         if(newOperation == Operator.EQUALS && previousOperation != //Operator.EQUALS){
//             operatorButtonPressed(previousOperation);
//         } else {
            switch (previousOperation) {
                case Operator.PLUS:
                    displayValue = lastValue + displayValue;
                    commitOperation(newOperation, displayValue);
                    break;
                case Operator.MINUS:
                    displayValue = lastValue - displayValue;
                    commitOperation(newOperation, displayValue);
                    break;
                case Operator.MULTIPLY:
                    displayValue = lastValue * displayValue;
                    commitOperation(newOperation, displayValue);
                    break;
                case Operator.DIVIDE:
                    if (displayValue == 0) {
                        setError("ERROR: Division by Zero");
                    } else {
                        displayValue = lastValue / displayValue;
                        commitOperation(newOperation, displayValue);
                    }
```

```
                    break;
                case Operator.EQUALS:
                    commitOperation(newOperation, displayValue);
//            }
        }
    }

    public void decimalButtonPressed() {
        String displayText = lcdDisplay.getText();
        if (FIRST.equals(status)) {
            displayText = "0";
        }

        if(!displayText.contains(".")){
            displayText = displayText + ".";
        }
        lcdDisplay.setText(displayText);
        status = VALID;
    }

    private void setError(String errorMessage) {
        if(errorMessage.trim().equals("")){
            errorMessage = " ";
        }
        errorDisplay.setText(errorMessage);
    }

    private void clearError(){
        status = FIRST;
        errorDisplay.setText(" ");
    }

    private void commitOperation(int operation, double result) {
        status = FIRST;
        lastValue = result;
        previousOperation = operation;
        lcdDisplay.setText(String.valueOf(result));
    }

    /**
     * Resets the program state.
     */
    void resetState() {
        clearError();
        lastValue = 0;
        previousOperation = Operator.EQUALS;

        lcdDisplay.setText("0");
    }
}
```

In this "calculator": 4 * 6 = 36, 4 * 9 = 81, 4 * 3 = 9. Error1 in strings:

```
                case Operator.MULTIPLY:
                    displayValue *= displayValue;
                // This is Error1, must be: displayValue *= lastValue;
```

After correcting Error1 in this "calculator": 1 / 3 = 3, 5 / 2 = 0.4, 8 / 4 = 0.5. Error2 in strings:

```
                case Operator.DIVIDE:
                if (displayValue == 0) {
                        setError("ERROR: Division by Zero");
                    } else {
                        displayValue /= lastValue;
                // This is Error2, must be: displayValue = lastValue / displayValue;
```

After correcting Error2 in this "calculator": 9 - 6 = -3, 6 - 3 = -3, 8 - 4 = -4. Error3 in strings:

```
                case Operator.MINUS:
```

```
                displayValue -= lastValue;
            // This is Error3, must be: displayValue = lastValue - displayValue;
```

Error4 in strings:

```
        if(newOperation == Operator.EQUALS && previousOperation !=
            Operator.EQUALS){
            operatorButtonPressed(previousOperation);
        } else {
```

After removing this strings calculator work correctly.
Here (http://ultrastudio.org/en/Software_calculator) you can find the similar calculator running on a web like Java applet, with source code and code review available.

Reference

Website

1. http://java.sun.com/docs/books/tutorial/uiswing/start/about.html

2. http://javapassion.com

Retrieved from "https://en.wikibooks.org/w/index.php?title=Java_Swings/Print_version&oldid=3040512"

- This page was last modified on 25 January 2016, at 03:01.
- Text is available under the Creative Commons Attribution-ShareAlike License.; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.