UNIT IV

APPLETS

An **applet** is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following −

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet

Four methods in the Applet class gives you the framework on which you build any serious applet −

- **init** − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start** − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop** − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy** − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint** − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet

Following is a simple applet named HelloWorldApplet.java −

```
import java.applet.*;
import java.awt.*;

public class HelloWorldApplet extends Applet {
   public void paint (Graphics g) {
      g.drawString ("Hello World", 25, 50);
   }
}
```

These import statements bring the classes into the scope of our applet class −

- java.applet.Applet
- java.awt.Graphics

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet Class

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following −

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may −

- Request information about the author, version, and copyright of the applet
- Request a description of the parameters the applet recognizes
- Initialize the applet
- Destroy the applet
- Start the applet's execution
- Stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

An applet is a Java™ program designed to be included in an HTML Web document. You can write your Java applet and include it in an HTML page, much in the same way an image is included. When you use a Java-enabled browser to view an HTML page that contains an applet, the applet's code is transferred to your system and is run by the browser's Java virtual machine.

The HTML document contains tags, which specify the name of the Java applet and its Uniform Resource Locator (URL). The URL is the location at which the applet bytecodes reside on the Internet. When an HTML document containing a Java applet tag is displayed, a Java-enabled Web browser downloads the Java bytecodes from the Internet and uses the Java virtual machine to process the code from within the Web document. These Java applets are what enable Web pages to contain animated graphics or interactive content.

You can also write a Java application that does not require the use of a Web browser.

For more information, see Writing Applets, Sun Microsystems' tutorial for Java applets. It includes an overview of applets, directions for writing applets, and some common applet problems.

Applications are stand-alone programs that do not require the use of a browser. Java applications run by starting the Java interpreter from the command line and by specifying the file that contains the compiled application. Applications usually reside on the system on which they are deployed. Applications access resources on the system, and are restricted by the Java security model.

Parent topic:

Java platform

Related concepts:

Java virtual machine

Java JAR and class files

Java threads

Java Development Kit

Graphics in Applets

The idea behind the graphics of an applet is that you paint the pixels of the rectangular part of the screen that is controlled by the applet. As Java is an object-oriented language, a *graphics object* is attached to the applet, and there is a paint method that will take care of the coloring of the pixels.

The pixels are counted *downward and to the right* and the upper left corner is the origin. This direction is convention in computer graphics, and is the same in which you read a book. Note it is not the mathematical convention of graphics, where the origin is in the lower left corner.

Graphics Object

There is not only a graphics object for the applet, there is a graphics object attached to any component. Often it is called the *graphics context*. It contains several instance variables, that determine the effect of drawing routines like drawText and drawLine:

1. Color, with methods setColor and getColor
2. Font, with methods setFont and getFont
3. Clip region: this is a restriction of the area of pixels that are affected by the paint routine. Outside the region the pixels will not change. Default clip region is the complete component, the associated methods are clipRect and getClipRect.

Most other methods of the Graphics class are graphics routines like drawLine, fillRect, and drawText. Note that there is also a method drawImage for displaying any GIF- or JPEG images. In xxx there is more on image loading.

There are two points to note:

- The graphics object does not contain the lines, ovals, etc. as objects, neither does it contain the pixels as objects. It just represents the screen area and paints it according to the paint routine. After a paint call, it forgets about the state of pixels.
- You implement the paint method for a generic Graphics object

  public void paint (Graphics g) { ... }

  but you hardly ever explicitly call it for the graphics object that is attached to the applet in your code (then it should read something like this.paint(getGraphics())). Instead the paint method is called automatically by the *Screen Updater* of the applet when a screen refresh is needed. This is an example of a "callback routine".

Class java.awt.AWTException

```
java.lang.Object
  |
  +----java.lang.Throwable
          |
          +----java.lang.Exception
                  |
                  +----java.awt.AWTException
```

public class **AWTException**

extends Exception

Signals that an Absract Window Toolkit exception has occurred.

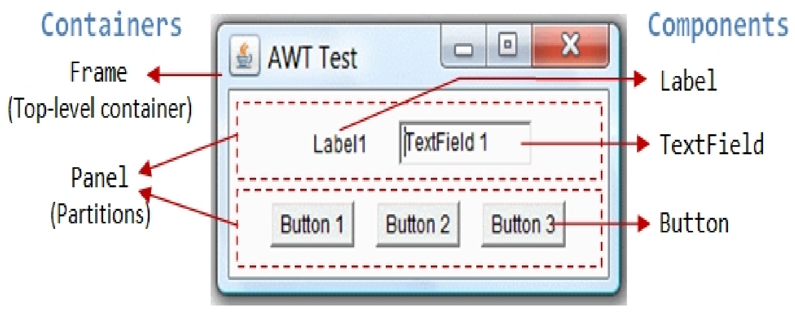
AWT Packages

AWT is huge! It consists of 12 packages of 370 classes (Swing is even bigger, with 18 packages of 737 classes as of JDK 8). Fortunately, only 2 packages - java.awt and java.awt.event - are commonly-used.

1. The java.awt package contains the *core* AWT graphics classes:
   - GUI Component classes, such as Button, TextField, and Label.
   - GUI Container classes, such as Frame and Panel.
   - Layout managers, such as FlowLayout, BorderLayout and GridLayout.
   - Custom graphics classes, such as Graphics, Color and Font.
2. The java.awt.event package supports event handling:
   - Event classes, such as ActionEvent, MouseEvent, KeyEvent and WindowEvent,
   - Event Listener Interfaces, such as ActionListener, MouseListener, MouseMotionListener, KeyListener and WindowListener,
   - Event Listener Adapter classes, such as MouseAdapter, KeyAdapter, and WindowAdapter.

AWT provides a *platform-independent* and *device-independent* interface to develop graphic programs that runs on all platforms, including Windows, Mac OS X, and Unixes.

2.2 Containers and Components



There are two types of GUI elements:

1. *Component*: Components are elementary GUI entities, such as Button, Label, and TextField.
2. *Container*: Containers, such as Frame and Panel, are used to *hold components in a specific layout* (such as FlowLayout or GridLayout). A container can also hold sub-containers.

In the above figure, there are three containers: a Frame and two Panels. A Frame is the *top-level container* of an AWT program. A Frame has a title bar (containing an icon, a title, and the minimize/maximize/close buttons), an optional menu bar and the content display area. A Panel is a *rectangular area* used to group related GUI components in a certain layout. In the above figure, the top-level Frame contains two Panels. There are five components: a Label (providing description), a TextField (for users to enter text), and three Buttons (for user to trigger certain programmed actions).

In a GUI program, a component must be kept in a container. You need to identify a container to hold the components. Every container has a method called add(Component c). A container (say c) can invoke c.add(aComponent) to add aComponent into itself. For example,

Introduction

So far, we have covered the basic programming constructs (such as variables, data types, decision, loop, array and method) and introduced the important concept of Object-Oriented Programming (OOP). As discussed, OOP permits higher level of abstraction than traditional Procedural-Oriented languages (such as C and Pascal). You can create high-level abstract data types called *classes* to mimic real-life things. These classes are self-contained and are *reusable*.

In this article, I shall show you how you can *reuse* the graphics classes provided in JDK for constructing your own Graphical User Interface (GUI) applications. Writing your own graphics classes (and re-inventing the wheels) is mission impossible! These graphics classes, developed by expert programmers, are highly complex and involve many advanced *design patterns*. However, re-using them are not so difficult, if you follow the API documentation, samples and templates provided.

I shall assume that you have a good grasp of OOP, including composition, inheritance, polymorphism, abstract class and interface; otherwise, read the earlier articles. I will describe another important OO concept called *nested class* (or *inner class*) in this article.

There are current three sets of Java APIs for graphics programming: AWT (Abstract Windowing Toolkit), Swing and JavaFX.

1. AWT API was introduced in JDK 1.0. Most of the AWT components have become

obsolete and should be replaced by newer Swing components.

2. Swing API, a much more comprehensive set of graphics libraries that enhances the AWT, was introduced as part of Java Foundation Classes (JFC) after the release of JDK 1.1. JFC consists of Swing, Java2D, Accessibility, Internationalization, and Pluggable Look-and-Feel Support APIs. JFC has been integrated into core Java since JDK 1.2.

3. The latest JavaFX, which was integrated into JDK 8, is meant to replace Swing.

Other than AWT/Swing/JavaFX graphics APIs provided in JDK, other organizations/vendors have also provided graphics APIs that work with Java, such as Eclipse's Standard Widget Toolkit (SWT) (used in Eclipse), Google Web Toolkit (GWT) (used in Android), 3D Graphics API such as Java bindings for OpenGL (JOGL) and Java3D.

Programming GUI with AWT

I shall start with the AWT before moving into Swing to give you a complete picture of Java Graphics.

AWT Classes

The AWT classes are contained in the **java.awt** package. It is one of Java’s largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table 25-1 lists some of the many AWT classes.

AWT Classes

- The **AWT classes** are contained in the **java**. ...
- AWTEvent : Encapsulates **AWT** events.
- AWTEventMulticaster : Dispatches events to multiple listeners.
- BorderLayout : The border layout manager. ...
- Button : Creates a push button control.
- Canvas : A blank, semantics-free window.
- CardLayout : The card layout manager.

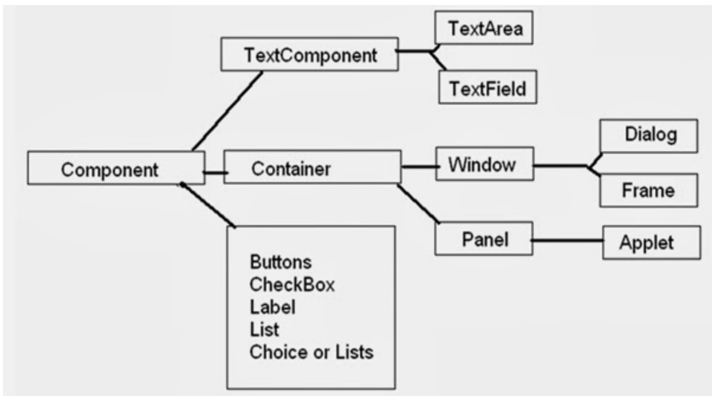
WINDOW FUNDAMENTALS

Java Abstract window tool kit package is used for displaying the data within a GUI Environment. Features of AW T Package are as Followings:

1. It provides us a set of user interface components including windows buttons text fields scrolling list etc.
2. It provides us the way to laying out these above components.
3. It provides to create the events upon these components.

The main purpose for using the AWT is using for all the components displaying on the screen. Awt defines all the windows according to a class hierarchy those are useful at a specific level or we can say arranged according to their functionality.

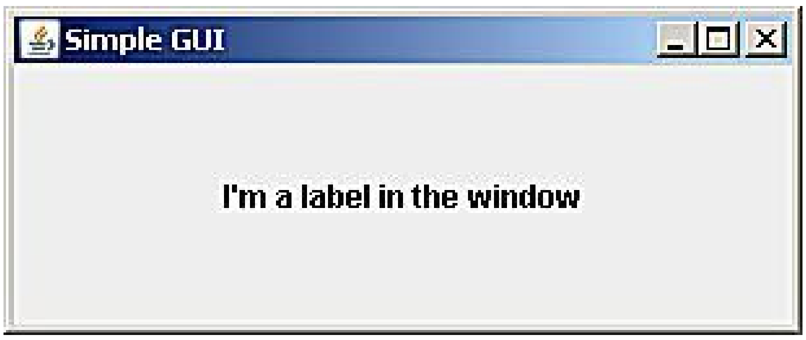
The most commonly used interface is the panels those are used by applets and those are derived from frame which creates a standard window. The hierarchy of this Awt is:-



Creating Window

A graphical user interface starts with a top-level container which provides a home for the other components of the interface, and dictates the overall feel of the application. In this tutorial, we introduce the JFrame class, which is used to create a simple top-level window for a Java application.

Import the Graphical Components



Microsoft product screen shot(s) reprinted with permission from Microsoft Corporation.

Open your text editor to start a new text file, and type in the following:

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

Java comes with a set of code libraries designed to help programmers quickly create applications. They provide access to classes that perform specific functions, to save you the bother of having to write them yourself. The two import statements above let the compiler know that the application needs access to some of the pre-built functionality contained within the "AWT" and "Swing" code libraries.

AWT stands for “Abstract Window Toolkit.” It contains classes that programmers can use to make graphical components such as buttons, labels and frames. Swing is built on top of AWT, and provides an additional set of more sophisticated graphical interface components. With just two lines of code, we gain access to these graphical components, and can use them in our Java application.

Create a Message Dialog Box

Dialog class is used to create a top-level container Dialog window which contains a set of components.

There are two kinds of Dialog windows -

- **Modal Dialog window**

  When a modal dialog window is active, all the user inputs are directed to it and all the other parts of application are inaccessible until this model dialog is closed.
- **Modeless Dialog window**

  When a modeless dialog window is active, the other parts of application are still accessible as normal and inputs can be directed to them, without needing to close this modeless dialog window.

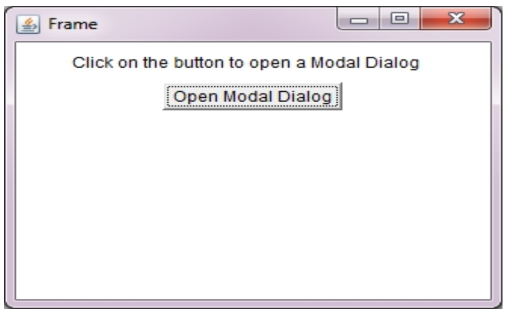
Constructors of Dialog

Constructor	**Description**
public Dialog()	Creates a modeless Dialog window without a Frame owner or title..
public Dialog(Dialog owner, String title)	Creates a modeless Dialog window with a Frame owner and a title.
public Dialog(Dialog owner, String title, boolean modal	Creates a Dialog window with a Frame owner, title and let's you define modality of Dialog window.

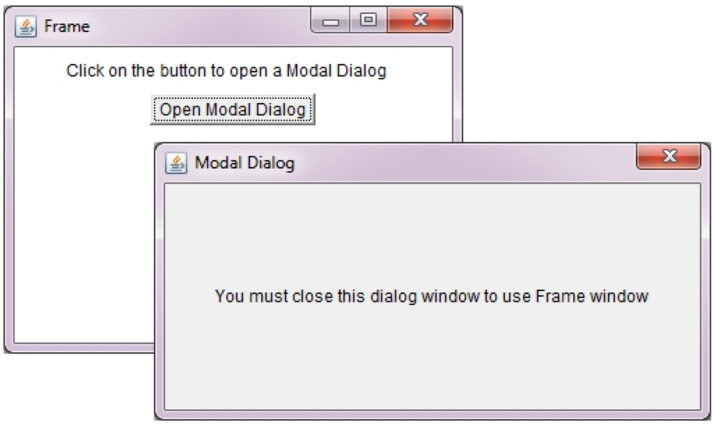
An example of Modal Dialog window.

```
import java.awt.*;
import java.awt.event.*;
public class DialogEx2 extends WindowAdapter implements ActionListener
{
Frame frame;
Label label1;
TextField field1;
Button button1, button2, button3;
Dialog d1, d2, d3;
DialogEx2()
{
frame = new Frame("Frame");
button1 = new Button("Open Modal Dialog");
label1 = new Label("Click on the button to open a Modal Dialog");
frame.add(label1);
frame.add(button1);
button1.addActionListener(this);
frame.pack();
frame.setLayout(new FlowLayout());
frame.setSize(330,250);
frame.setVisible(true);
}
public void actionPerformed(ActionEvent ae)
{

if(ae.getActionCommand().equals("Open Modal Dialog"))
{
//Creating a non-modeless blocking Dialog
d1= new Dialog(frame,"Modal Dialog",true);
Label label= new Label("You must close this dialog window to use Frame
window",Label.CENTER);
d1.add(label);

d1.addWindowListener(this);
d1.pack();
d1.setLocationRelativeTo(frame);
d1.setLocation(new Point(100,100));
d1.setSize(400,200);
d1.setVisible(true);
}
}
public void windowClosing(WindowEvent we)
{
d1.setVisible(false);
}

public static void main(String...ar)
{
new DialogEx2();
}

}
```

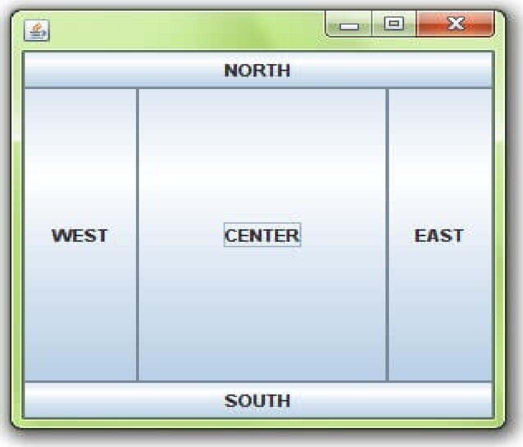
When you run the code, you are presented a window that shows a button asking you to click on it to open a Model Dialog window, shown in Figure 1.



When you click on the this button, you are presented with a Modal Dialog window which should be closed in order to continue with the program and go back to Frame window. Hence, a model Dialog window blocks all the inputs from user to the window in its background, until it is not closed.


Frame
Click on the button to open a Modal Dialog
Open Modal Dialog
Modal Dialog
You must close this dialog window to use Frame window

Example of BorderLayout class:



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

public class Border {
JFrame f;
Border(){
    f=new JFrame();
    JButton b1=new JButton("NORTH");;
```

```java
    JButton b2=new JButton("SOUTH");;
    JButton b3=new JButton("EAST");;
    JButton b4=new JButton("WEST");;
    JButton b5=new JButton("CENTER");;
    f.add(b1,BorderLayout.NORTH);
    f.add(b2,BorderLayout.SOUTH);
    f.add(b3,BorderLayout.EAST);
    f.add(b4,BorderLayout.WEST);
    f.add(b5,BorderLayout.CENTER);
    f.setSize(300,300);
    f.setVisible(true);
}
public static void main(String[] args) {
    new Border();
}
}
```

Java LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

Java BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window. The BorderLayout provides five constants for each region:

1. **public static final int NORTH**

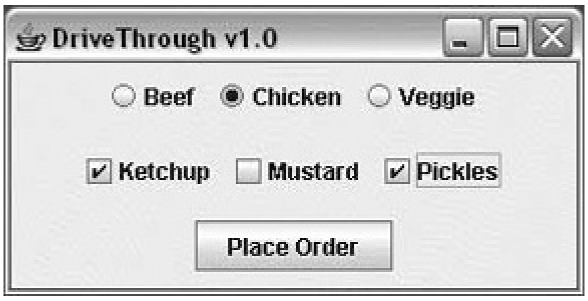
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**
6. **Checkboxes and Radio Buttons**

A checkbox is a labeled toggle switch. Each time the user clicks it, its state toggles between checked and unchecked. Swing implements the checkbox as a special kind of button. Radio buttons are similar to checkboxes, but they are normally used in groups. Clicking on one radio button in the group automatically turns the others off. They are named for the mechanical preset buttons on old car radios (like some of us had in high school).

Checkboxes and radio buttons are represented by instances of JCheckBox and JRadioButton, respectively. Radio buttons can be tethered together using an instance of another class called ButtonGroup . By now you're probably well into the swing of things (no pun intended) and could easily master these classes on your own. We'll use an example to illustrate a different way of dealing with the state of components and to show off a few more things about containers.

A JCheckBox sends ItemEvents when it's pushed. Because a checkbox is a kind of button, it also fires ActionEvents when checked. For something like a checkbox, we might want to be lazy and check on the state of the buttons only at some later time, such as when the user commits an action. For example, when filling out a form you may only care about the user's choices when the submit button is finally pressed.

The next application, DriveThrough, lets us check off selections on a fast food menu, as shown in Figure 17-3.

7. 

How to Use Borders

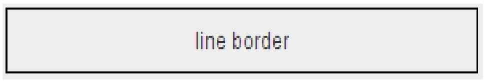
Every JComponent can have one or more borders. Borders are incredibly useful objects that, while not themselves components, know how to draw the edges of Swing components. Borders are useful not only for drawing lines and fancy edges, but also for providing titles and empty space around components.

Our examples set borders on JPanels, JLabels, and custom subclasses of JComponent. Although technically you can set the border on any object that inherits from JComponent, the look and feel implementation of many standard Swing components doesn't work well with user-set borders. In general, when you want to set a border on a standard Swing component other than JPanel or JLabel, we recommend that you put the component in a JPanel and set the border on the JPanel.

To put a border around a JComponent, you use its setBorder method. You can use the BorderFactory class to create most of the borders that Swing provides. If you need a reference to a border — say, because you want to use it in multiple components — you can save it in a variable of type Border. Here is an example of code that creates a bordered container:

```
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createLineBorder(Color.black));
```

Here's a picture of the container, which contains a label component. The black line drawn by the border marks the edge of the container.



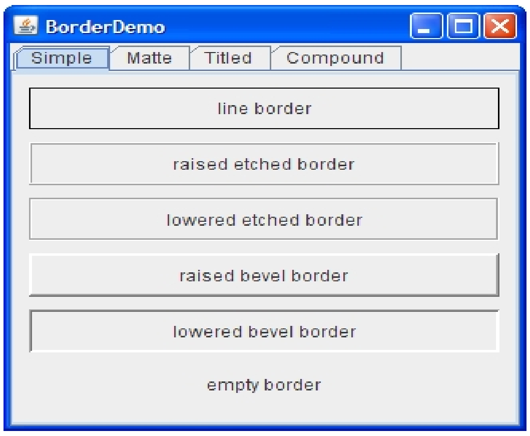
The rest of this page discusses the following topics:

- The BorderDemo Example
- Using the Borders Provided by Swing
- Creating Custom Borders
- The Border API
- Examples of Using Borders

The BorderDemo Example

The following pictures show an application called BorderDemo that displays the borders Swing provides. We show the code for creating these borders a little later, in Using the Borders Provided by Swing.

Click the Launch button to run the BorderDemo example using Java™ Web Start (download JDK 7 or later). Alternatively, to compile and run the example yourself, consult the example index.



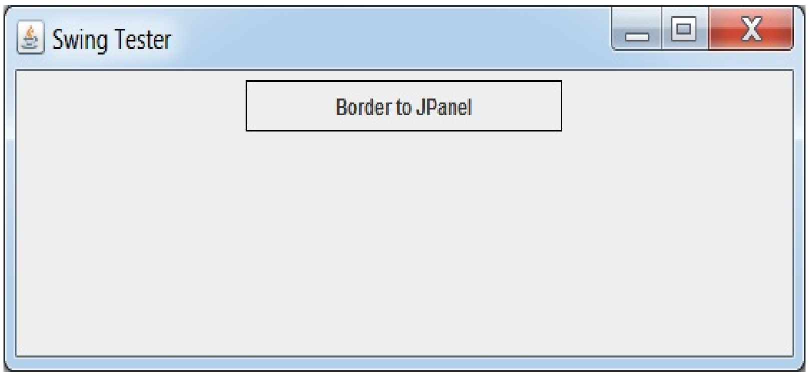
Swing

Following example showcase how to add border to a JPanel in a Java Swing application.

We are using the following APIs.

- **BorderFactory.createLineBorder()** – To create a line border.
- **JPanel.setBorder(border)** – To set the desired border to the JPanel.

Output


Swing Tester
Border to JPanel