**UNIT V**

**EVENT DRIVEN PROGRAMMING**

**Graphics programming - Frame – Components - working with 2D shapes - Using color, fonts, and images - Basics of event handling - event handlers - adapter classes - actions - mouse events - AWT event hierarchy - Introduction to Swing – layout management - Swing Components – Text Fields , Text Areas – Buttons- Check Boxes – Radio Buttons – Lists- choices- Scrollbars – Windows –Menus – Dialog Boxes**

**GRAPHICS PROGRAMMING**

The **Graphics class** is the abstract super **class** for all **graphics** contexts which allow an application to draw onto components that can be realized on various devices, or onto off-screen images as well. A **Graphics** object encapsulates all state information required for the basic rendering operations that **Java** supports.

**FRAME**

**Creating A Frame**

A top-level window (that is, a window that is not contained inside another window) is called a frame in Java. The AWT library has a class, called Frame, for this top level. The Swing version of this class is called JFrame and extends the Frame class. The JFrame is one of the few Swing components that is not painted on a canvas. Thus, the decorations (buttons, title bar, icons, and so on) are drawn by the user’s windowing system, not by Swing.

CAUTION: Most Swing component classes start with a “J”: JButton, JFrame, and so on. There are classes such as Button and Frame, but they are AWT components. If you accidentally omit a “J”, your program may still compile and run, but the mixture of Swing and AWT components can lead to visual and behavioral inconsistencies.

**The simplest visible frame**

Let’s work through this program, line by line. The Swing classes are placed in the javax.swing package. The package name javax indicates a Java extension package, not a core package. For historical reasons, Swing is considered an extension. However, it is present in every Java SE implementation since version 1.2. By default, a frame has a rather useless size of 0 × 0 pixels. We define a subclass Simple - Frame whose constructor sets the size to 300 × 200 pixels. This is the only difference between a SimpleFrame and a JFrame

**SimpleFrameTest.java**

import javax.swing.\*;

public class SimpleFrameTest{

public static void main(String[] args) {

SimpleFrame frame = new SimpleFrame();

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setVisible(true);} }

class SimpleFrame extends JFrame{

public SimpleFrame() {

setSize(DEFAULT\_WIDTH, DEFAULT\_HEIGHT); }

public static final int DEFAULT\_WIDTH = 300;

public static final int DEFAULT\_HEIGHT = 200;}

**Positioning a Frame**

The JFrame class itself has only a few methods for changing how frames look. Of course, through the magic of inheritance, most of the methods for working with the size and position of a frame come from the various superclasses of JFrame. Here are some of the most important methods:

• The setLocation and setBounds methods for setting the position of the frame

• The setIconImage method, which tells the windowing system which icon to display in the title bar, task switcher window, and so on

• The setTitle method for changing the text in the title bar

• The setResizable method, which takes a boolean to determine if a frame will be resizeable by the user As the API notes indicate, the Component class (which is the ancestor of all GUI objects) and the Window class (which is the superclass of the Frame class) are where you need to look to find the methods to resize and reshape frames. For example, the setLocation method in the Component class is one way to reposition a component. If you make the call setLocation(x, y) the top-left corner is located x pixels across and y pixels down, where (0, 0) is the top-left corner of the screen. Similarly, the setBounds method in Component lets you resize and relocate a component (in particular, a JFrame) in one step, as setBounds(x, y, width, height)

**Frame Properties**

Many methods of component classes come in getter/setter pairs, such as the following methods of the Frame class:

* public String getTitle()
* public void setTitle(String title)

Such a getter/setter pair is called a property. A property has a name and a type. The name is obtained by changing the first letter after the get or set to lowercase. For example, the Frame class has a property with name title and type String. Conceptually, title is a property of the frame. When we set the property, we expect that the title changes on the user’s screen. When we get the property, we expect that we get back the value that we set.

We do not know (or care) how the Frame class implements this property. Perhaps it simply uses its peer frame to store the title. Perhaps it has an instance field private String title; // not required for property If the class does have a matching instance field, we don’t know (or care) how the getter and setter methods are implemented. Perhaps they just read and write the instance field. Perhaps they do more, such as notifying the windowing system whenever the title changes. There is one exception to the get/set convention:

For properties of type boolean, the getter starts with is. For example, the following two methods define the locationByPlatform property:

public boolean isLocationByPlatform()

public void setLocationByPlatform(boolean b)

import java.awt.\*;

import javax.swing.\*;

public class SizedFrameTest

{

public static void main(String[] args)

{

EventQueue.invokeLater(new Runnable()

{

public void run()

{

SizedFrame frame = new SizedFrame();

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setVisible(true);

}

});

}

}

class SizedFrame extends JFrame

{

public SizedFrame()

{

// get screen dimensions

Toolkit kit = Toolkit.getDefaultToolkit();

Dimension screenSize = kit.getScreenSize();

int screenHeight = screenSize.height;

int screenWidth = screenSize.width;

set frame width, height and let platform pick screen location

setSize(screenWidth / 2, screenHeight / 2);

setLocationByPlatform(true);

// set frame icon and title

Image img = kit.getImage("icon.gif");

setIconImage(img);

setTitle("SizedFrame");

}

}

**boolean isVisible()**

* void setVisible(boolean b)

gets or sets the visible property. Components are initially visible, with the exception of top-level components such as JFrame.

* void setSize(int width, int height) **1.1**

resizes the component to the specified width and height.

* void setLocation(int x, int y) **1.1**

moves the component to a new location. The x- and y-coordinates use the coordinates of the container if the component is not a top-level component, or the coordinates of the screen if the component is top level (for example, a JFrame).

• void setBounds(int x, int y, int width, int height) **1.1** moves and resizes this component.

• Dimension getSize() **1.1**

• void setSize(Dimension d) **1.1**

gets or sets the size property of this component.

• void toFront()

shows this window on top of any other windows.

• void toBack()

moves this window to the back of the stack of windows on the desktop and rearranges all other visible windows accordingly.

• boolean isLocationByPlatform() **5.0**

• void setLocationByPlatform(boolean b) **5.0**

gets or sets the locationByPlatform property. When the property is set before this window is displayed, the platform picks a suitable location.

• boolean isResizable()

• void setResizable(boolean b)

gets or sets the resizable property. When the property is set, the user can resize the frame.

• String getTitle()

• void setTitle(String s)

gets or sets the title property that determines the text in the title bar for the frame.

• Image getIconImage()

• void setIconImage(Image image)

gets or sets the iconImage property that determines the icon for the frame. The windowing system may display the icon as part of the frame decoration or in other locations.

**COMPONENTS**

**Displaying Information in a Component**

In this section, we show you how to display information inside a frame. For example, rather than displaying “Not a Hello, World program” in text mode in a console window as we did in Chapter 1, You could draw the message string directly onto a frame, but that is not considered good programming practice. In Java, frames are really designed to be containers for components such as a menu bar and other user interface elements.

You normally draw on another component which you add to the frame. The structure of a JFrame is surprisingly complex. Look at Figure 7–7, which shows the makeup of a JFrame. As you can see, four panes are layered in a JFrame. The root pane, layered pane, and glass pane are of no interest to us; they are required to organize the menu bar and content pane and to implement the look and feel. The part that most con**java.awt.Toolkit**

**Displaying Information in a Component 295**

When designing a frame, you add components into the content pane, using code such as the following:

Container contentPane = frame.getContentPane();

Component c = . . .;

contentPane.add(c);

Up to Java SE 1.4, the add method of the JFrame class was defined to throw an exception with the message “Do not use JFrame.add(). Use JFrame.getContentPane().add() instead.” As of Java SE 5.0, the JFrame.add method has given up trying to reeducate programmers, and it simply calls add on the content pane. Thus, as of Java SE 5.0, you can simply use the call

frame.add(c);

Here’s how to make a component onto which you can draw:

class MyComponent extends JComponent{

public void paintComponent(Graphics g){

code for drawing}}

Each time a window needs to be redrawn, no matter what the reason, the event handler notifies the component. This causes the paintComponent methods of all components to be executed. Never call the paintComponent method yourself. It is called automatically whenever a part of your application needs to be redrawn, and you should not interfere with this automatic process. What sorts of actions trigger this automatic response?

For example, painting occurs because the user increased the size of the window or minimized and then restored the window. If the user popped up another window and it covered an existing window and then made the overlaid window disappear, the application window that was covered is now corrupted and will need to be repainted. (The graphics system does not save the pixels underneath.) And, of course, when the window is displayed for the first time, it needs to process the code that specifies how and where it should draw the initial elements.

As you saw in the code fragment above, the paintComponent method takes a single parameter of type Graphics. Measurement on a Graphics object for screen display is done in pixels. The (0, 0) coordinate denotes the top-left corner of the component on whose surface you are drawing. Displaying text is considered a special kind of drawing. The Graphics class has a drawStringmethod that has the following syntax:

g.drawString(text, x, y)

In our case, we want to draw the string "Not a Hello, World Program" in our original window, roughly one-quarter of the way across and halfway down. Although we don’t yet know how to measure the size of the string, we’ll start the string at coordinates (75, 100). This means the first character in the string will start at a position 75 pixels to the right and 100 pixels down. (Actually, it is the baseline for the text that is 100 pixels down—see page 313 for more on how text is measured.)

**Displaying Information in a Component 297**

class NotHelloWorldComponent extends JComponent{

public void paintComponent(Graphics g){

g.drawString("Not a Hello, World program", MESSAGE\_X, MESSAGE\_Y);}

public static final int MESSAGE\_X = 75;

public static final int MESSAGE\_Y = 100;}

The easiest way to achieve that is to paint the panel with the background color, by calling super.paintComponent in thepaintComponent method of each panel subclass:

class NotHelloWorldPanel extends JPanel{

public void paintComponent(Graphics g){

super.paintComponent(g);

. . . // code for drawing will go here}

}

**NotHelloWorld.java**

import javax.swing.\*;

import java.awt.\*;

@version 1.32 2007-06-12

@author Cay Horstmann

public class NotHelloWorld

{

public static void main(String[] args)

{

EventQueue.invokeLater(new Runnable()

{

public void run()

{

NotHelloWorldFrame frame = new NotHelloWorldFrame();

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setVisible(true);

}

• Container getContentPane()

returns the content pane object for this JFrame.

• Component add(Component c)

adds and returns the given component to the content pane of this frame. (Before Java SE 5.0, this method threw an exception.)

class NotHelloWorldFrame extends JFrame

{

public NotHelloWorldFrame()

{

setTitle("NotHelloWorld");

setSize(DEFAULT\_WIDTH, DEFAULT\_HEIGHT);

// add panel to frame

NotHelloWorldPanel panel = new NotHelloWorldPanel();

add(panel);

}

public static final int DEFAULT\_WIDTH = 300;

public static final int DEFAULT\_HEIGHT = 200;

}

class NotHelloWorldPanel extends JPanel

{

public void paintComponent(Graphics g)

{

g.drawString("Not a Hello, World program", MESSAGE\_X, MESSAGE\_Y);

}

public static final int MESSAGE\_X = 75;

public static final int MESSAGE\_Y = 100;

}

• void repaint()

causes a repaint of the component “as soon as possible.”

• public void repaint(int x, int y, int width, int height)

causes a repaint of a part of the component “as soon as possible.”

• void paintComponent(Graphics g)

overrides this method to describe how your component needs to be painted.

**WORKING WITH 2D SHAPES**

Starting with Java 1.0, the Graphics class had methods to draw lines, rectangles, ellipses, and so on. But those drawing operations are very limited. For example, you cannot vary the line thickness and you cannot rotate the shapes. Java SE 1.2 introduced the Java 2D library, which implements a powerful set of graphical operations. In this chapter, we only look at the basics of the Java 2D library—see the Advanced AWT chapter in Volume II for more information on the advanced features.

To draw shapes in the Java 2D library, you need to obtain an object of the Graphics2D class. This class is a subclass of the Graphics class. Ever since Java SE 2, methods such as paintComponent automatically receive an object of the Graphics2D class. Simply use a cast, as follows:

public void paintComponent(Graphics g){

Graphics2D g2 = (Graphics2D) g;. . .

}

The Java 2D library organizes geometric shapes in an object-oriented fashion. In particular, there are classes to represent lines, rectangles, and ellipses:

Line2D

Rectangle2D

Ellipse2D

These classes all implement the Shape interface.

To draw a shape, you first create an object of a class that implements the Shape interface and then call the draw method of the Graphics2D class. For example: Rectangle2D rect = . . .;

g2.draw(rect);

Using the Java 2D shape classes introduces some complexity. Unlike the 1.0 draw methods, which used integer pixel coordinates, the Java 2D shapes use floatingpoint coordinates. In many cases, that is a great convenience because it allows you to specify your shapes in coordinates that are meaningful to you (such as millimeters or inches) and then translate to pixels. The Java 2D library uses single-precision float quantities for many of its internal floating-point calculations. Single precision is sufficient—after all, the ultimate purpose of the geometric computations is to set pixels on the screen or printer.

As long as any roundoff errors stay within one pixel, the visual outcome is not affected. Furthermore, float computations are faster on some platforms, and float values require half the storage of double values. However, manipulating float values is sometimes inconvenient for the programmer because the Java programming language is adamant about requiring casts when converting double values into float values. For example, consider the following statement:

float f = 1.2; // Error

This statement does not compile because the constant 1.2 has type double, and the compiler is nervous about loss of precision. The remedy is to add an F suffix to the floatingpoint constant:

float f = 1.2**F**; // Ok

Now consider this statement:

Rectangle2D r = . . .

float f = r.getWidth(); // Error

This statement does not compile either, for the same reason. The getWidth method returns a double. This time, the remedy is to provide a cast:

float f = **(float)** r.getWidth(); // Ok

Because e the suffixes and casts are a bit of a pain, the designers of the 2D library decided to supply two versions of each shape class: one with float coordinates for frugal programmers, and one with double coordinates for the lazy ones. (In this book, we fall into the second camp and use double coordinates whenever we can.)

The library designers chose a curious, and initially confusing, method for packaging these choices. Consider the Rectangle2D class. This is an abstract class with two concretesubclasses, which are also static inner classes:

Rectangle2D.Float

Rectangle2D.Double

It is best to try to ignore the fact that the two concrete classes are static inner classes— that is just a gimmick to avoid names such as FloatRectangle2D and DoubleRectangle2D. (For more information on static inner classes, see Chapter 6.)

When you construct a Rectangle2D.Float object, you supply the coordinates as float numbers. For a Rectangle2D.Double object, you supply them as double numbers.

Rectangle2D.Float floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F); Rectangle2D.Double doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0); Actually, because both Rectangle2D.Float and Rectangle2D.Double extend the common Rectangle2D class and the methods in the subclasses simply override methods in the Rectangle2D superclass, there is no benefit in remembering the exact shape type. You can simply use Rectangle2D variables to hold the rectangle references.

**Rectangle2D** floatRect = new **Rectangle2D.Float**(10.0F, 25.0F, 22.5F, 20.0F);

**Rectangle2D** doubleRect = new **Rectangle2D.Double**(10.0, 25.0, 22.5, 20.0);

That is, you only need to use the pesky inner classes when you construct the shape objects. The construction parameters denote the top-left corner, width, and height of the rectangle. The Rectangle2D methods use double parameters and return values. For example, the getWidth method returns a double value, even if the width is stored as a float in a Rectangle2D.Float object.

**USING COLOR, FONTS, AND IMAGES**

**Using Color**

The setPaint method of the Graphics2D class lets you select a color that is used or all subsequent drawing operations on the graphics context. For example:

g2.setPaint(Color.RED);

g2.drawString("Warning!", 100, 100);

You can fill the interiors of closed shapes (such as rectangles or ellipses) with a color.

Simply call fill instead of draw:

Rectangle2D rect = . . .;

g2.setPaint(Color.RED);

g2.fill(rect); // fills rect with red color

To draw in multiple colors, you select a color, draw or fill, then select another color, and draw or fill again. You define colors with the Color class. The java.awt.Color class offers predefined constants for the following 13 standard colors: BLACK, BLUE, CYAN, DARK\_GRAY, GRAY, GREEN, LIGHT\_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW

NOTE: Before Java SE 1.4, color constant names were lowercase, such as Color.**red**. This is odd because the standard coding convention is to write constants in uppercase. You can now write the standard color names in uppercase or, for backward compatibility, in lowercase.

You can specify a custom color by creating a Color object by its red, green, and blue components. Using a scale of 0–255 (that is, one byte) for the redness, blueness, and greenness, call the Color constructor like this:

Color(int redness, int greenness, int blueness)

Here is an example of setting a custom color:

g2.setPaint(new Color(0, 128, 128)); // a dull blue-green

g2.drawString("Welcome!", 75, 125);

**java.awt.geom.Ellipse2D.Double 1.2**

**java.awt.geom.Point2D.Double 1.2**

**java.awt.geom.Line2D.Double 1.2**

NOTE: In addition to solid colors, you can select more complex “paint” settings, such as varying hues or images. See the Advanced AWT chapter in Volume II for more details. If you use a Graphics object instead of a Graphics2D object, you need to use the setColor method to set colors.

To set the background color, you use the setBackground method of the Component class, an ancestor of JComponent.

MyComponent p = new MyComponent();

p.setBackground(Color.PINK);

There is also a setForeground method. It specifies the default color that is used for drawing on the component.

TIP: The brighter() and darker() methods of the Color class produce, as their names suggest, either brighter or darker versions of the current color. Using the brighter method is also a good way to highlight an item. Actually, brighter() is just a little bit brighter. To make a color really stand out, apply it three times: c.brighter().brighter().brighter(). Java gives you predefined names for many more colors in its SystemColor clas s.

The constants in this class encapsulate the colors used for various elements of the user’s system. For example,

p.setBackground(SystemColor.window)

sets the background color of the component to the default used by all windows on the user’s desktop. (The background is filled in whenever the window is repainted.) Using the colors in the SystemColor class is particularly useful when you want to draw user interfaceelements so that the colors match those already found on the user’s desktop.

**System Colors**

**Name Purpose**

desktop Background color of desktop

activeCaption Background color for captions

activeCaptionText Text color for captions

activeCaptionBorder Border color for caption text

inactiveCaption Background color for inactive captions

inactiveCaptionText Text color for inactive captions

inactiveCaptionBorder Border color for inactive captions

window Background for windows

windowBorder Color of window border frame

**Using Color**

The setPaint method of the Graphics2D class lets you select a color that is used for all subsequent drawing operations on the graphics context. For example:

g2.setPaint(Color.RED);

g2.drawString("Warning!", 100, 100);

You can fill the interiors of closed shapes (such as rectangles or ellipses) with a color.

Simply call fill instead of draw:

Rectangle2D rect = . . .;

g2.setPaint(Color.RED);

g2.fill(rect); // fills rect with red color

To draw in multiple colors, you select a color, draw or fill, then select another color, and

draw or fill again. You define colors with the Color class. The java.awt.Color class offers predefined constants for the following 13 standard colors:

BLACK, BLUE, CYAN, DARK\_GRAY, GRAY, GREEN, LIGHT\_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW

NOTE: Before Java SE 1.4, color constant names were lowercase, such as Color.**red**. This is odd because the standard coding convention is to write constants in uppercase. You can now write the standard color names in uppercase or, for backward compatibility, in lowercase. You can specify a custom color by creating a Color object by its red, green, and blue components.

Using a scale of 0–255 (that is, one byte) for the redness, blueness, and greenness, call the Color constructor like this:

Color(int redness, int greenness, int blueness)

Here is an example of setting a custom color:

g2.setPaint(new Color(0, 128, 128)); // a dull blue-green

g2.drawString("Welcome!", 75, 125);

**Using Special Fonts for Text**

The “Not a Hello, World” program at the beginning of this chapter displayed a string in the default font. Often, you want to show text in a different font. You specify a font by its font face name. A font face name is composed of a font family name, such as “Helvetica, and an optio nal suffix such as “Bold.” For example, the font faces “Helvetica” and “Helvetica Bold” are both considered to be part of the family named “Helvetica.” To find out which fonts are available on a particular computer, call the getAvailable- FontFamilyNames method of the GraphicsEnvironment class.

The method returns an array of strings that contains the names of all available fonts. To obtain an instance of theGraphicsEnvironment class that describes the graphics environment of the user’s system, use the static getLocalGraphicsEnvironment method. Thus, the following program prints the names of all fonts on your system:

import java.awt.\*;

public class ListFonts

{

public static void main(String[] args)

{

String[] fontNames = GraphicsEnvironment

.getLocalGraphicsEnvironment()

.getAvailableFontFamilyNames();

for (String fontName : fontNames)

System.out.println(fontName);

}

}

**java.awt.Graphics2D 1.2**

**java.awt.Component 1.0**

Parameters: c The new background color

Parameters: c The new foreground color

Chapter 7. Graphics Programming

**Using Special Fonts for Text 311**

On one system, the list starts out like this:

Abadi MT Condensed Light

Arial

Arial Black

Arial Narrow

Arioso

Baskerville

Binner Gothic

. . .

and goes on for another 70 fonts or so. Font face names can be trademarked, and font designs can be copyrighted in some jurisdictions. Thus, the distribution of fonts often involves royalty payments to a font foundry. Of course, just as there are inexpensive imitations of famous perfumes, there are lookalikes for name-brand fonts. For example, the Helvetica imitation that is shipped with Windows is called Arial. To establish a common baseline, the AWT defines five logical font names:

* SansSerif
* Serif
* Monospaced
* Dialog
* DialogInput

These names are always mapped to fonts that actually exist on the client machine. For example, on a Windows system, SansSerif is mapped to Arial. In addition, the Sun JDK always includes three font families named “Lucida Sans,”

“Lucida Bright,” and “Lucida Sans Typewriter.”

To draw characters in a font, you must first create an object of the class Font. You specify the font face name, the font style, and the point size. Here is an example of how you construct a Font object:

Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);

The third argument is the point size. Points are commonly used in typography to indicate the size of a font. There are 72 points per inch.You can use a logical font name in the place of a font face name in the Font constructor. You specify the style (plain, **bold**, italic, or **bold italic**) by setting the second Font constructor argument to one of the following values:

* Font.PLAIN
* Font.BOLD
* Font.ITALIC
* Font.BOLD + Font.ITALIC

You can read font files in TrueType or PostScript Type 1 formats. You need an input stream for the font—typically from a file or URL. (See Chapter 1 of Volume II for more information on streams.) Then call the static Font.createFont method:

URL url = new URL("http://www.fonts.com/Wingbats.ttf");

InputStream in = url.openStream();

Font f1 = Font.createFont(Font.TRUETYPE\_FONT, in);

The font is plain with a font size of 1 point. Use the deriveFont method to get a font of the desired size:

Font f = f1.deriveFont(14.0F);

CAUTION: There are two overloaded versions of the deriveFont method. One of them (with a float parameter) sets the font size, the other (with an int parameter) sets the font style. Thus, f1.deriveFont(14) sets the style and not the size! (The result is an italic font because it happens that the binary representation of 14 sets the ITALIC bit but not the BOLD bit.)

The Java fonts contain the usual ASCII characters as well as symbols. For example, if you print the character '\u2297' in the Dialog font, then you get a ⊗ character. Only those symbols that are defined in the Unicode character set are available.

**Displaying Images**

You have already seen how to build up simple drawings by painting lines and shapes. Complex images, such as photographs, are usually generated externally, for example, with a scanner or special image-manipulation software .Once images are stored in local files or someplace on the Internet, you can read them into a Java application and display them on Graphics objects. As of Java SE 1.4, reading an image is very simple. If the image is stored in a local file, call

String filename = "...";

Image image = ImageIO.read(new File(filename));

Otherwise, you can supply a URL:

String urlname = "...";

Image image = ImageIO.read(new URL(urlname));

The read method throws an IOException if the image is not available. We discuss the general topic of exception handling in Chapter 11. For now, our sample program just catches that exception and prints a stack trace if it occurs. Now the variable image contains a reference to an object that encapsulates the image data

You can display the image with the drawImage method of the Graphics class.

public void paintComponent(Graphics g)

{

. . .

g.drawImage(image, x, y, null);

}

Listing 7–6 takes this a little bit further and tiles the window with the graphics image. The result looks like the screen shown in Figure 7–14. We do the tiling in the paintComponent

method. We first draw one copy of the image in the top-left corner and then use the copyArea call to copy it into the entire window:

for (int i = 0; i \* imageWidth <= getWidth(); i++)

for (int j = 0; j \* imageHeight <= getHeight(); j++)

if (i + j > 0)

g.copyArea(0, 0, imageWidth, imageHeight, i \* imageWidth, j \* imageHeight);

import java.awt.\*;

import java.io.\*;

import javax.imageio.\*;

import javax.swing.\*;

\* @version 1.33 2007-04-14

\* @author Cay Horstmann

\*/

public class ImageTest

{ public static void main(String[] args)

{

EventQueue.invokeLater(new Runnable()

{

public void run()

{

ImageFrame frame = new ImageFrame();

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

20. frame.setVisible(true);

}

});

}

class ImageFrame extends JFrame {

public ImageFrame()

{

setTitle("ImageTest");

setSize(DEFAULT\_WIDTH, DEFAULT\_HEIGHT);

ImageComponent component = new ImageComponent();

add(component);

}

public static final int DEFAULT\_WIDTH = 300;

public static final int DEFAULT\_HEIGHT = 200;

}

\* A component that displays a tiled image

\*/

class ImageComponent extends JComponent

{

public ImageComponent()

{

// acquire the image

try

{

image = ImageIO.read(new File("blue-ball.gif"));

}

catch (IOException e)

{

e.printStackTrace();

}

}

public void paintComponent(Graphics g)

{

if (image == null) return;

int imageWidth = image.getWidth(this);

// draw the image in the top-left corner

g.drawImage(image, 0, 0, null);

// tile the image across the component

for (int i = 0; i \* imageWidth <= getWidth(); i++)

for (int j = 0; j \* imageHeight <= getHeight(); j++)

if (i + j > 0) g.copyArea(0, 0, imageWidth, imageHeight, i \* imageWidth, j

\* imageHeight);

}

private Image image;

}

**EVENT HANDLING**

**E**vent handling is of fundamental importance to programs with a graphical user interface. To implement user interfaces, you must master the way in which Java handles events. This chapter explains how the Java AWT event model works. You will see how to capture events from user interface components and input devices. We also show you how to work with actions, a more structured approach for processing action events. Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks. The operating environment reports these events to the programs that are running.

Each program then decides what, if anything, to do in response to these events. In languages like Visual Basic, the correspondence between events and code is obvious. One writes code for each specific event of interest and places the code in what is usually called an event procedure. For example, a Visual Basic button named “HelpButton” would have a HelpButton\_Click event procedure associated with it. The code in this procedure executes whenever that button is clicked. Each Visual Basic GUI component responds to a fixed set of events, and it is impossible to change the events to which a Visual Basic component responds. On the other hand, if you use a language like raw C to do event-driven programming, you need to write the code that constantly checks the event queue for what the operating environment is reporting. (You usually do this by encasing your code in a loop with a massive switch statement!)

This technique is obviously rather ugly, and, in any case, it is much more difficult to code. The advantage is that the events you can respond to are not as limited as in languages, like Visual Basic, that go to great lengths to hide the event queue from the programmer.

The Java programming environment takes an approach somewhat between the Visual Basic approach and the raw C approach in terms of power and, therefore, in resulting complexity. Within the limits of the events that the AWT knows about, you completely control how events are transmitted from the event sources (such as buttons or scrollbars) to event listeners. You can designate any object to be an event listener—in practice, you pick an object that can conveniently carry out the desired response to the event. This event delegation model gives you much more flexibility than is possible with Visual Basic in which the listener is predetermined.Event sources have methods that allow you to register event listeners with them. When an event happens to the source, the source sends a notification of that event to all the listener objects that were registered for that event.

As one would expect in an object-oriented language like Java, the information about the event is encapsulated in an event object. In Java, all event objects ultimately derive from the class java.util.EventObject. Of course, there are subclasses for each event type, such as ActionEvent and WindowEvent. Different event sources can produce different kinds of events. For example, a button can send ActionEvent objects, whereas a window can send WindowEvent objects. To sum up, here’s an overview of how event handling in the AWT works:

• A listener object is an instance of a class that implements a special interface called (naturally enough) a listener interface.

• An event source is an object that can register listener objects and send them event

objects.

**BASICS OF EVENT HANDLING**

* The event source sends out event objects to all registered listeners when that event occurs.
* The listener objects will then use the information in the event object to determine their reaction to the event.

Here is an example for specifying a listener:

ActionListener listener = . . .;

JButton button = new JButton("Ok");

button.addActionListener(listener);

Now the listener object is notified whenever an “action event” occurs in the button. For

buttons, as you might expect, an action event is a button click.

To implement the ActionListener interface, the listener class must have a method called actionPerformed that receives an ActionEvent object as a parameter.

class MyListener **implements ActionListener**

{

. . .

**public void actionPerformed(ActionEvent event)**

{

// reaction to button click goes here

. . .

}

}

Whenever the user clicks the button, the JButton object creates an ActionEvent object and calls listener.actionPerformed(event), passing that event object. An event source such as a button can have multiple listeners. In that case, the button calls the actionPerformed methods of all listeners whenever the user clicks the button.

**Example: Handling a Button Click**

As a way of getting comfortable with the event delegation model, let’s work through all details needed for the simple example of responding to a button click. For this example, we will show a panel populated with three buttons. Three listener objects are added as action listeners to the buttons. You create a button by specifying a label string, an icon, or both in the button constructor.

Here are two examples:

JButton yellowButton = new JButton("Yellow");

JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));

Call the add method to add the buttons to a panel:

JButton yellowButton = new JButton("Yellow");

JButton blueButton = new JButton("Blue");

JButton redButton = new JButton("Red");

buttonPanel.add(yellowButton);

buttonPanel.add(blueButton);

buttonPanel.add(redButton);

**EVENT HANDLERS**

public class **EventHandler**

extends [Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)

implements [InvocationHandler](https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/InvocationHandler.html)

The EventHandler class provides support for dynamically generating event listeners whose methods execute a simple statement involving an incoming event object and a target object. The EventHandler class is intended to be used by interactive tools, such as application builders, that allow developers to make connections between beans.

Typically connections are made from a user interface bean (the event *source*) to an application logic bean (the *target*). The most effective connections of this kind isolate the application logic from the user interface.

For example, the EventHandler for a connection from a JCheckBox to a method that accepts a boolean value can deal with extracting the state of the check box and passing it directly to the method so that the method is isolated from the user interface layer.

Inner classes are another, more general way to handle events from user interfaces. The EventHandler class handles only a subset of what is possible using inner classes. However, EventHandler works better with the long-term persistence scheme than inner classes. Also, using EventHandler in large applications in which the same interface is implemented many times can reduce the disk and memory footprint of the application.

The reason that listeners created with EventHandler have such a small footprint is that the Proxy class, on which the EventHandler relies, shares implementations of identical interfaces. For example, if you use the EventHandler create methods to make all the ActionListeners in an application, all the action listeners will be instances of a single class (one created by the Proxy class). In general, listeners based on the Proxy class require one listener class to be created per *listener type* (interface), whereas the inner class approach requires one class to be created per *listener* (object that implements the interface).

**ADAPTER CLASSES**

Not all events are as simple to handle as button clicks. In a non-toy program, you will want to monitor when the user tries to close the main frame because you don’t want your sers to lose unsaved work. When the user closes the frame, you want to put up a dialog and exit the program only when the user agrees.When the program user tries to close a frame window, the JFrame object is the source of a WindowEvent. If you want to catch that event, you must have an appropriate listener object and add it to the frame’s list of window listeners.

WindowListener listener = . . .;

frame.addWindowListener(listener);

The window listener must be an object of a class that implements the WindowListener interface. There are actually seven methods in the WindowListener interface. The frame calls them as the responses to seven distinct events that could happen to a window. The names are self-explanatory, except that “iconified” is usually called “minimized” under Windows. Here is the complete WindowListener interface:

public interface WindowListener

{

void windowOpened(WindowEvent e);

void windowClosing(WindowEvent e);

void windowClosed(WindowEvent e);

void windowIconified(WindowEvent e);

void windowDeiconified(WindowEvent e);

void windowActivated(WindowEvent e);

void windowDeactivated(WindowEvent e);

}

**ACTIONS**

It is common to have multiple ways to activate the same command. The user can choose a certain function through a menu, a keystroke, or a button on a toolbar. This is easy to achieve in the AWT event model: link all events to the same listener. For example, suppose blueAction is an action listener whose actionPerformed method changes the background color to blue. You can attach the same object as a listener to several event sources:

• A toolbar button labeled “Blue”

• A menu item labeled “Blue”

• A keystroke CTRL+B

Then the color change command is handled in a uniform way, no matter whether it was caused by a button click, a menu selection, or a key press. The Swing package provides a very useful mechanism to encapsulate commands and to attach them to multiple event sources: the Action interface. An action is an object that encapsulates

• A description of the command (as a text string and an optional icon); and

• Parameters that are necessary to carry out the command (such as the requested

color in our example).

The Action interface has the following methods:

void actionPerformed(ActionEvent event)

void setEnabled(boolean b)

boolean isEnabled()

void putValue(String key, Object value)

Object getValue(String key)

void addPropertyChangeListener(PropertyChangeListener listener)

void removePropertyChangeListener(PropertyChangeListener listener)

The first method is the familiar method in the ActionListener interface: in fact, the Action interface extends the ActionListener interface. Therefore, you can use an Action object whenever an ActionListener object is expected.The next two methods let you enable or disable the action and check whether the action is currently enabled. When an action is attached to a menu or toolbar and the action is disabled, then the option is grayed out.

The putValue and getValue methods let you store and retrieve arbitrary name/value pairs in the action object. A couple of important predefined strings, namely, Action.NAME and Action.SMALL\_ICON, store action names and icons into an action object:

action.putValue(Action.NAME, "Blue");

action.putValue(Action.SMALL\_ICON, new ImageIcon("blue-ball.gif"));

If the action object is added to a menu or toolbar, then the name and icon are automatically retrieved and displayed in the menu item or toolbar button. The SHORT\_DESCRIPTION value turns into a tooltip. The final two methods of the Action interface allow other objects, in particular menus or toolbars that trigger the action, to be notified when the properties of the action object change. For example, if a menu is added as a property change listener of an action object and the action object is subsequently disabled, then the menu is called and can gray out the action name. Property change listeners are a general construct that is a part of the “JavaBeans” component model.

public class ColorAction extends AbstractAction

{

public ColorAction(String name, Icon icon, Color c)

{

putValue(Action.NAME, name);

putValue(Action.SMALL\_ICON, icon);

putValue("color", c);

putValue(Action.SHORT\_DESCRIPTION, "Set panel color to " + name.toLowerCase());

}

public void actionPerformed(ActionEvent event)

{

Color c = (Color) getValue("color");

buttonPanel.setBackground(c);

}

}

NAME The name of the action; displayed on buttons and menu items.

SMALL\_ICON A place to store a small icon; for display in a button, menu item, or toolbar.

SHORT\_DESCRIPTION A short description of the icon; for display in a tooltip.

LONG\_DESCRIPTION A long description of the icon; for potential use in on-line help. No Swing component uses this value.

MNEMONIC\_KEY A mnemonic abbreviation; for display in menu items ACCELERATOR\_KEY A place to store an accelerator keystroke. No Swing component uses this value.

ACTION\_COMMAND\_KEY Historically, used in the now obsolete registerKeyboardAction method.

DEFAULT Potentially useful catch-all property. No Swing component uses this value. Our test program creates three objects of this class, such as Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);

Next, let’s associate this action with a button. That is easy because we can use a JButton

constructor that takes an Action object.

JButton blueButton = new JButton(blueAction);

**MOUSE EVENTS**

You do not need to handle mouse events explicitly if you just want the user to be able to click on a button or menu. These mouse operations are handled internally by the various components in the user interface. However, if you want to enable the user to draw with the mouse, you will need to trap mouse move, click, and drag events.

Some user interface designers inflict mouse click and keyboard modifier combinations, such as CONTROL + SHIFT + CLICK, on their users. We find this practice reprehensible, but if you disagree, you will find that checking for mouse buttons and keyboard modifiers is a mess. You use bit masks to test which modifiers have been set. In the original API, two of the button masks equal two keyboard modifier masks, namely

BUTTON2\_MASK == ALT\_MASK

BUTTON3\_MASK == META\_MASK

This was done so that users with a one-button mouse could simulate the other mouse buttons by holding down modifier keys instead. However, as of Java SE 1.4, a different approach is recommended. There are now masks

BUTTON1\_DOWN\_MASK

BUTTON2\_DOWN\_MASK

BUTTON3\_DOWN\_MASK

SHIFT\_DOWN\_MASK

CTRL\_DOWN\_MASK

ALT\_DOWN\_MASK

ALT\_GRAPH\_DOWN\_MASK

META\_DOWN\_MASK

The getModifiersEx method accurately reports the mouse buttons and keyboard modifiers

of a mouse event. Note that BUTTON3\_DOWN\_MASK tests for the right (nonprimary) mouse button under Windows. For example, you can use code like this to detect whether the right mouse button is down:

if ((event.getModifiersEx() & InputEvent.BUTTON3\_DOWN\_MASK) != 0)

. . . // code for right click

In our sample program, we supply both a mousePressed and a mouseClicked method. When you click onto a pixel that is not inside any of the squares that have been drawn, a new square is added. We implemented this in the mousePressed method so that the user receives immediate feedback and does not have to wait until the mouse button is released. When a user double-clicks inside an existing square, it is erased. We implemented this in the mouseClicked method because we need the click count.

public void mousePressed(MouseEvent event)

{

current = find(event.getPoint());

if (current == null) // not inside a square

add(event.getPoint());

}

public void mouseClicked(MouseEvent event)

{

current = find(event.getPoint());

if (current != null && event.getClickCount() >= 2)

remove(current);

}

**AWT EVENT HIERARCHY**

Having given you a taste of how event handling works, we finish this chapter with an overview of the AWT event handling architecture. As we briefly mentioned earlier, event handling in Java is object oriented, with all events descending from the EventObject class in the java.util package. (The common superclass is not called Event because that is the name of the event class in the old event model. Although the old model is now deprecated, its classes are still a part of the Java library.)

The EventObject class has a subclass AWTEvent, which is the parent of all AWT event classes. Figure 8–8 shows the inheritance diagram of the AWT events. Some of the Swing components generate event objects of yet more event types; these directly extend EventObject, not AWTEvent. The event objects encapsulate information about the event that the event source communicates to its listeners. When necessary, you can then analyze the event objects that were passed to the listener object, as we did in the button example with the getSource and getActionCommand methods.

Some of the AWT event classes are of no practical use for the Java programmer. For example, the AWT inserts PaintEvent objects into the event queue, but these objects are not delivered to listeners. Java programmers don’t listen to paint events; they override the paintComponent method to control repainting. The AWT also generates a number of events that are needed only by system programmers, to provide input systems for ideographic languages, automated testing robots, and so on.

**INTRODUCTION TO SWING**

* The original Java GUI subsystem was the Abstract Window Toolkit (AWT).
* AWT translates it visual components into platform-specific equivalents (peers).
* Under AWT, the look and feel of a component was defined by the platform.
* AWT components are referred to as **heavyweight**.
* Swing was introduced in 1997 to fix the problems with AWT.
* Swing offers two key features:
  1. Swing components are **lightweight** and don't rely on peers.
  2. Swing supports a pluggable look and feel. The three PLAFs available to all users are Metal (default), Windows, and Motif.
* Swing is built on AWT.

**LAYOUT MANAGEMENT**

**Introduction to Layout Management**

Before we go on to discussing individual Swing components, such as text fields and radio buttons, we briefly cover how to arrange these components inside a frame. Unlike Visual Basic, the JDK has no form designer. You need to write code to position (lay out) the user interface components where you want them to be. Of course, if you have a Java-enabled development environment, it will probably have a layout tool that automates some or all of these tasks. Nevertheless, it is important to know exactly what goes on “under the hood” because even the best of these tools will usually require hand-tweaking.

**Border Layout**

The border layout manager is the default layout manager of the content pane of every JFrame. Unlike the flow layout manager, which completely controls the position of each component, the border layout manager lets you choose where you want to place each component. You can choose to place the component in the center, north, south, east, or west of the content pane For example:

frame.add(component, BorderLayout.SOUTH);

The edge components are laid out first, and the remaining available space is occupied by the center. When the container is resized, the dimensions of the edge components are unchanged, but the center component changes its size. You add components by specifying a constant CENTER, NORTH, SOUTH, EAST, or WEST of the BorderLayout class. Not all of the positions need to be occupied. If you don’t supply any value, CENTER is assumed.

**Grid Layout**

The grid layout arranges all components in rows and columns like a spreadsheet. All components are given the same size. The calculator program in Figure 9–12 uses a grid layout to arrange the calculator buttons. When you resize the window, the buttons grow and shrink, but all buttons have identical sizes.

**SWING COMPONENTS**

A component is an independent visual control. Swing Framework contains a large set of components which provide rich functionalities and allow high level of customization. They all are derived from JComponent class. All these components are lightweight components. This class provides some common functionality like pluggable look and feel, support for accessibility, drag and drop, layout, etc.

A container holds a group of components. It provides a space where a component can be managed and displayed. Containers are of two types,

* Top level Containers

It inherits Component and Container of AWT.

It cannot be contained within other containers.

Heavyweight.

Example: JFrame, JDialog, JApplet

* Lightweight Containers

It inherits JComponent class.

It is a general purpose container.

It can be used to organize related components together.

Example: JPanel

JButton class provides functionality of a button. JButton class has three constuctors,

* JButton(Icon ic)
* JButton(String str)
* JButton(String str, Icon ic)

**TEXT FIELDS**

JTextField is used for taking input of single line of text. It is most widely used text component. It has three constructors,

* JTextField(int cols)
* JTextField(String str, int cols)
* JTextField(String str)

cols represent the number of columns in text field.

**Example using JTextField**

import javax.swing.\*;

import java.awt.event.\*;

import java.awt.\*;

public class MyTextField extends JFrame

{

public MyTextField()

{

JTextField jtf = new JTextField(20); //creating JTextField.

add(jtf); //adding JTextField to frame.

setLayout(new FlowLayout());

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(400, 400);

setVisible(true);

}

public static void main(String[] args)

{

new MyTextField();

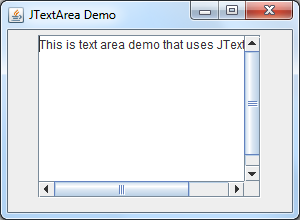
}

}

**TEXT AREAS**

we create a new text area using the *JTextArea* class.

Here is the screenshot of the demo application:



**BUTTONS**

package jtextareademo;

import javax.swing.\*;

import java.awt.\*;

public class Main {

public static void main(String[] args) {

final JFrame frame = new JFrame("JTextArea Demo");

JTextArea ta = new JTextArea(10, 20);

JScrollPane sp = new JScrollPane(ta);

frame.setLayout(new FlowLayout());

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setSize(300, 220);

frame.getContentPane().add(sp);

frame.setVisible(true);

}}

**Example using JButton**

import javax.swing.\*;

import java.awt.event.\*;

import java.awt.\*;

public class testswing extends JFrame

{

testswing() {

JButton bt1 = new JButton("Yes"); //Creating a Yes Button.

JButton bt2 = new JButton("No"); //Creating a No Button.

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE) //setting close operation.

setLayout(new FlowLayout()); //setting layout using FlowLayout object

setSize(400, 400); //setting size of Jframe

add(bt1); //adding Yes button to frame.

add(bt2); //adding No button to frame.

setVisible(true); }

public static void main(String[] args) {

new testswing(); }}

**JCHECKBOX**

JCheckBox class is used to create checkboxes in frame. Following is constructor for JCheckBox,

JCheckBox(String str)

**Example using JCheckBox**

import javax.swing.\*;

import java.awt.event.\*;

import java.awt.\*;

public class Test extends JFrame{

public Test() {

JCheckBox jcb = new JCheckBox("yes"); //creating JCheckBox.

add(jcb); //adding JCheckBox to frame.

jcb = new JCheckBox("no"); //creating JCheckBox.

add(jcb); //adding JCheckBox to frame.

jcb = new JCheckBox("maybe"); //creating JCheckBox.

add(jcb); //adding JCheckBox to frame.

setLayout(new FlowLayout());

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(400, 400);

setVisible(true); }

public static void main(String[] args) {

new Test(); }}

Objects for the above components can be created by invoking its constructor . Let we see an example program

**RADIO BUTTONS**

Radio button is a group of related button in which only one can be selected. JRadioButton class is used to create a radio button in Frames. Following is the constructor for JRadioButton,

JRadioButton(String str)

**Example using JRadioButton**

import javax.swing.\*;

import java.awt.event.\*;

import java.awt.\*;

public class Test extends JFrame

{

public Test()

{

JRadioButton jcb = new JRadioButton("A"); //creating JRadioButton.

add(jcb); //adding JRadioButton to frame.

jcb = new JRadioButton("B"); //creating JRadioButton.

add(jcb); //adding JRadioButton to frame.

jcb = new JRadioButton("C"); //creating JRadioButton.

add(jcb); //adding JRadioButton to frame.

jcb = new JRadioButton("none");

add(jcb);

setLayout(new FlowLayout());

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(400, 400);

setVisible(true);

}

public static void main(String[] args)

{

new Test();

**}}**

**LISTS**

Java.util.List is a child interface of [Collection](https://www.geeksforgeeks.org/collections-in-java-2/). List is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order it allows positional access and insertion of elements. List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

**Creating List Objects:**  
List is an interface, we can create instance of List in following ways:

**List a = new ArrayList();**

**List b = new LinkedList();**

**List c = new Vector();**

**List d = new Stack(); Generic List Object:**  
 After the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the List.

**Operations on List:**  
 List Interface extends Collection, hence it supports all the operations of Collection Interface and along with following operations:

1. **Positional Access:**  
   List allows add, remove, get and set operations based on numerical positions of elements in List. List provides following methods for these operations:
   * **void add(int index,Object O):** This method adds given element at specified index.
   * **boolean addAll(int index, Collection c):** This method adds all elements from specified collection to list. First element gets inserted at given index. If there is already an element at that position, that element and other subsequent elements(if any) are shifted to the right by increasing their index.
   * **Object remove(int index):** This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.
   * **Object get(int index):** This method returns element at the specified index.
   * **Object set(int index, Object new):** This method replaces element at given index with new element. This function returns the element which was just replaced by new element.

|  |
| --- |
| // Java program to demonstrate positional access  // operations on List interface  import java.util.\*;   public class ListDemo  {      public static void main (String[] args)      {          // Let us create a list          List l1 = new ArrayList();          l1.add(0, 1);  // adds 1 at 0 index          l1.add(1, 2);  // adds 2 at 1 index          System.out.println(l1);  // [1, 2]           // Let us create another list          List l2 = new ArrayList();          l2.add(1);          l2.add(2);          l2.add(3);           // will add list l2 from 1 index          l1.addAll(1, l2);          System.out.println(l1);           l1.remove(1);     // remove element from index 1          System.out.println(l1); // [1, 2, 3, 2]           // prints element at index 3          System.out.println(l1.get(3));           l1.set(0, 5);   // replace 0th element with 5          System.out.println(l1);  // [5, 2, 3, 2]      }  } |

1. **Search:**  
   List provides methods to search element and returns its numeric position. Following two methods are supported by List for this operation:

* **int indexOf(Object o):** This method returns first occurrence of given element or -1 if element is not present in list.
* **int lastIndexOf(Object o):**This method returns the last occurrence of given element or -1 if element is not present in list.

1. **Iteration:**  
   ListIterator(extends Iterator) is used to iterate over List element. List iterator is bidirectional iterator. For more details about ListIterator refer [Iterators in Java](https://www.geeksforgeeks.org/iterators-in-java/).
2. **Range-view:**  
   List Interface provides method to get List view of the portion of given List between two indices. Following is the method supported by List for range view operation.
   * **List subList(int fromIndex,int toIndex):**This method returns List view of specified List between fromIndex(inclusive) and toIndex(exclusive).

**CHOICES**

Creates an empty choice box; use addItem() to populate

import [java](http://ecomputernotes.com/java/what-is-java/what-is-java-explain-basic-features-of-java-language).awt.\*;   
class ChoiceExample extends Frame   
{   
     ChoiceExample()   
     {   
          setLayout(new FlowLayout());   
          Label lblCourse = new Label("Course");   
          Label lblweekDay = new Label("Day");   
          Choice course = new Choice();   
          course.add("BCA");   
          course.add("MCA");   
          course.add("MBA");   
          String[] day={"Mon","Tue","wed","Thu","fri","Sat","Sun"};   
          Choice weekDay =new Choice();   
          for(int i=0;i<day.length; i++)   
              {   
                    weekDay.add(day[i]);   
              }                                     
                   add(lblCourse);    add(course);         
                   add(lblweekDay);  add(weekDay);   
      }   
}   
  class ChoiceJavaExample   
  {   
       public static void main(String args[])   
      {   
               ChoiceExample frame = new ChoiceExample();   
               frame.setTitle("Choice in Java Example");   
               frame.setSize(250,100);   
               frame.setResizable(false);   
               frame.setVisible(true);   
      }   
 }

**SCROLLBARS**

The Scrollbar class embodies a scroll bar, a familiar user-interface object. A scroll bar provides a convenient means for allowing a user to select from a range of values. The following three vertical scroll bars could be used as slider controls to pick the red, green, and blue components of a color:

Each scroll bar in this example could be created with code similar to the following:

redSlider=new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, 255);

add(redSlider);

Alternatively, a scroll bar can represent a range of values. For example, if a scroll bar is used for scrolling through text, the width of the "bubble" (also called the "thumb" or "scroll box") can be used to represent the amount of text that is visible. Here is an example of a scroll bar that represents a range:

Image shows horizontal slider with starting range of 0 and ending range of 300. The slider thumb is labeled 60.

**WINDOWS**

A Window object is a top-level window with no borders and no menubar. The default layout for a window is BorderLayout. A window must have either a frame, dialog, or another window defined as its owner when it's constructed.

In a multi-screen environment, you can create a Window on a different screen device by constructing the Window with [Window(Window, GraphicsConfiguration)](https://docs.oracle.com/javase/7/docs/api/java/awt/Window.html#Window(java.awt.Window,%20java.awt.GraphicsConfiguration)).

In a virtual device multi-screen environment in which the desktop area could span multiple physical screen devices, the bounds of all configurations are relative to the virtual device coordinate system. The origin of the virtual-coordinate system is at the upper left-hand corner of the primary physical screen. Depending on the location of the primary screen in the virtual device, negative coordinates are possible

**MENUS**

Menu menuView = new Menu ("View");

Once the Menu objects are created, we need to add them to the menu bar. For this, you have to use the add () method of the MenuBar class whose syntax is as follows,

Menu add (Menu menu)

where menu is Menu instance that is added to the menu bar. This method returns a reference to the menu. By default, consecutively added menus are positioned in the menu bar from left to right.

This makes the first menu added the leftmost menu and the last menu added the rightmost menu. If you want to add a menu at a specific location, then use the following version of add ()method inherited from the Container class.

Component add (Component menu, int index)

where menu is added to the menu bar at the specified index. Indexing begins at 0, with 0 being the leftmost menu. For example: In order to add menu instance menuFile to the menuBar use the following statement,

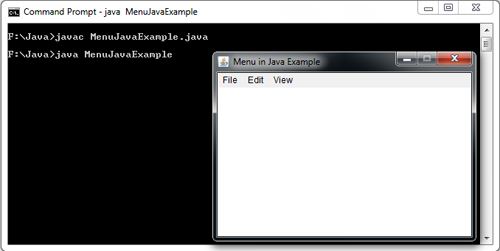
menuBar.add(menuFile);

Similarly, add other Menu instances menuEdit,menuView using the following statements,

menuBar.add(menuEdit);

menuBar.add(menuView);

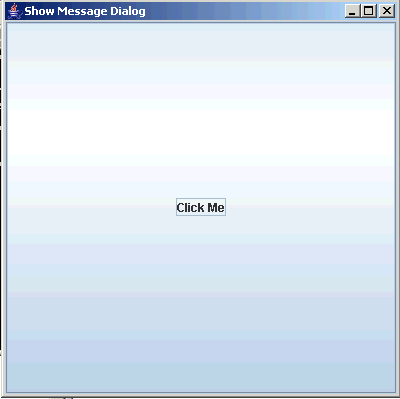
import [java](http://ecomputernotes.com/java/what-is-java/what-is-java-explain-basic-features-of-java-language).awt.\*;   
class MenuExample extends Frame   
{   
       MenuExample()   
      {   
           MenuBar menuBar = new MenuBar();   
           setMenuBar(menuBar);   
           Menu menuFile = new Menu("File");   
           Menu menuEdit = new Menu("Edit");   
           Menu menuView = new Menu("View");   
           menuBar.add(menuFile);   
           menuBar.add(menuEdit);   
           menuBar.add(menuView);   
           MenuItem itemOpen = new MenuItem("Open");   
           MenuItem itemSave = new MenuItem("Save");   
           MenuItem itemExit = new MenuItem("Exit");   
           menuFile.add(itemOpen);   
           menuFile.add(itemSave);   
           menuFile.add(itemExit);   
           MenuItem itemcopy = new MenuItem("Copy");   
           menuEdit.add(itemcopy);   
     }   
}   
  class MenuJavaExample   
  {   
          public static void main(String args[])   
         {   
              MenuExample frame = new MenuExample();   
              frame.setTitle("Menu in Java Example");   
              frame.setSize(350,250);   
              frame.setResizable(false);   
              frame.setVisible(true);   
         }   
  }

[](http://ecomputernotes.com/images/MenuJavaExample.jpg)

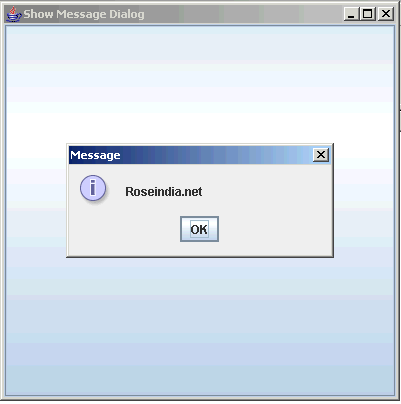
**DIALOG BOXES**

Message dialog box is used to display informative messages to the user. In this section we will use JOptionPane class to display the message Dialog box. Our program display "Click Me" button on the window and when user clicks on it program displays Message box with "OK" button and message "Roseindia.net".

When you run the program following window will be displayed:



When you click on "Click Me" button, following Message is displayed:



**Program description**:

**JOptionPane Class:**

In non-swing application we were using System.in class for input or output some text or numeric values but now in the swing application we can use JOptionPane to show the output or show the message. This way of inputting or outputting works very efficiently in the Swing Applications. The window for showing message for input or output makes your application very innovative.

JOptionPane class is available in the javax.swing.\*; package. This class provide various types of message dialog box as follows:

A simple message dialog box which has only one button i.e. "Ok". This type of message dialog box is used only for showing the appropriate message and user can finish the message dialog box by clicking the "Ok" button.

A message dialog box which has two or three buttons. You can set several values for viewing several message dialog box as follows:

1.) "Yes" and "No"

2.) "Yes", "No" and "Cancel"

3.) "Ok", and "Cancel"

A input dialog box which contains two buttons "Ok" and "Cancel".

The JOptionPane class has three methods as follows:

**showMessageDialog():** First is the showMessageDialog() method which is used to display a simple message.

**showInputDialog():** Second is the showInputDialog() method which is used to display a prompt for inputting. This method returns a String value which is entered by you.

**showConfirmDialog():** And the last or third method is the showConfirmDialog() which asks the user for confirmation (Yes/No) by displaying message. This method return a numeric value either 0 or 1. If you click on the "Yes" button then the method returns 1 otherwise 0.

**How program Works:**

This program illustrates you how to show a message dialog box when you click on the button.

**showMessageDialog():**

This method is used to show a message dialog box which contains some text messages. This is being used with two arguments in the program where the first argument is the parent object in which the dialog box opens and another is the message which has to be shown.

Here is the code of the program:

import javax.swing.\*;

import java.awt.event.\*;

public class ShowDialogBox{

JFrame frame;

public static void main(String[] args){

ShowDialogBox db = new ShowDialogBox();

}

public ShowDialogBox(){

frame = new JFrame("Show Message Dialog");

JButton button = new JButton("Click Me");

button.addActionListener(new MyAction());

frame.add(button);

frame.setSize(400, 400);

frame.setVisible(true);

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public class MyAction implements ActionListener{

public void actionPerformed(ActionEvent e){

JOptionPane.showMessageDialog(frame,"Roseindia.net");

} }}

import javax.swing.JOptionPane;

public class Main {

public static void main(String[] argv) throws Exception {

JOptionPane.showMessageDialog(null, "I am happy.");

}}