MT311
Unit 3
Graphics and graphical user interface
Course team
Developer: Dr Man Leung Wong, Lingnan University
Designer: Mr Chris Baker, OUHK
Coordinator: Dr Tak Sing Li, OUHK
Members: Dr Reggie Kwan, OUHK
          Mr Tai Loon Fong, OUHK

External Course Assessor
Prof. To Yat Cheung, City University of Hong Kong

Production
ETPU Publishing Team

Acknowledgement
The OUHK gratefully acknowledges a grant of $100,000,000 from the Hong Kong Government designated for course development.

MT311 Programming Languages and Java Programming was developed in part using resources provided by this generous grant.


All rights reserved.
No part of this material may be reproduced in any form by any means without permission in writing from the President, The Open University of Hong Kong.

The Open University of Hong Kong
30 Good Shepherd Street
Ho Man Tin, Kowloon
Hong Kong
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overview</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>Objectives</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Graphics and Java2D</strong></td>
<td>3</td>
</tr>
<tr>
<td>Graphics contexts and graphics objects</td>
<td>3</td>
</tr>
<tr>
<td>JFrame</td>
<td>5</td>
</tr>
<tr>
<td>Drawing shapes</td>
<td>7</td>
</tr>
<tr>
<td>Drawing lines</td>
<td>8</td>
</tr>
<tr>
<td>Color control</td>
<td>9</td>
</tr>
<tr>
<td>Font control</td>
<td>11</td>
</tr>
<tr>
<td>Drawing rectangles</td>
<td>14</td>
</tr>
<tr>
<td><strong>Basic graphical user interface components</strong></td>
<td>16</td>
</tr>
<tr>
<td>Event-driven programs</td>
<td>16</td>
</tr>
<tr>
<td>Controlling layout</td>
<td>17</td>
</tr>
<tr>
<td>FlowLayout layout manager</td>
<td>18</td>
</tr>
<tr>
<td>BorderLayout layout manager</td>
<td>20</td>
</tr>
<tr>
<td>GridLayout layout manager</td>
<td>22</td>
</tr>
<tr>
<td>JPanel</td>
<td>24</td>
</tr>
<tr>
<td>Capturing events</td>
<td>27</td>
</tr>
<tr>
<td>Event and listener types</td>
<td>31</td>
</tr>
<tr>
<td>Adapter class</td>
<td>32</td>
</tr>
<tr>
<td>Graphical user interface components</td>
<td>33</td>
</tr>
<tr>
<td>JLabel</td>
<td>34</td>
</tr>
<tr>
<td>JTextField</td>
<td>36</td>
</tr>
<tr>
<td>JButton</td>
<td>38</td>
</tr>
<tr>
<td>JCheckBox</td>
<td>41</td>
</tr>
<tr>
<td>JList and JScrollPane</td>
<td>43</td>
</tr>
<tr>
<td>Mouse events handling</td>
<td>45</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>51</td>
</tr>
<tr>
<td><strong>Feedback on activities</strong></td>
<td>52</td>
</tr>
<tr>
<td><strong>Solutions to self-tests</strong></td>
<td>58</td>
</tr>
</tbody>
</table>
Overview

This unit gives an overview of what the Java environment provides to help you create a user interface (UI). There are two kinds of GUI program in Java: applications and applets. This unit focuses on applications.

You first learn how to draw lines and shapes in different colors and select different fonts and font styles for writing text. After that, you learn how to produce an ‘interactive’ interface. As user interface programs are event-driven, it is necessary to define how to respond when a particular event happens. Capturing events and creating event handlers is discussed.

Also discussed is how to control the layout of different components onto a Java window. This is done by using the FlowLayout, GridLayout and BorderLayout layout managers. Some basic Java GUI components are discussed, including JButton, JLabel, JTextField, JCheckBox, JList and JScrollPane. You learn how to use these components and how to capture events on these components. Finally, you learn how to react with the events that come from the hardware like the mouse and the keyboard.

This unit is lengthy, and should take you about three weeks to complete (or about 19 hours of study time). Please plan your time carefully. If you have any difficulties, please contact your tutor.
Objectives

By the end of Unit 3, you should be able to:

1. *Describe* graphics contexts, graphics objects, and paint modes.
2. *Apply* methods for drawing strings, characters, and bytes.
3. *Use* methods for manipulating fonts and colors.
4. *Apply* methods for drawing lines, rectangles, ovals, arcs, and polygons.
5. *Develop* graphical user interfaces.
6. *Create* buttons, labels, lists, and text fields.
7. *Manage* mouse events and keyboard events.
8. *Apply* different layout managers.
Graphics and Java2D

Many classes of Java are introduced in this unit. When a new class is introduced, an URL of a Web page that contains detailed information of the class is given. Reading the Web pages is optional. The source code of all example programs in this unit is available in WebCT.

This subsection gives an overview of several Java capabilities for drawing two-dimensional shapes, controlling colors and fonts. The following figure shows a portion of the Java class hierarchy that includes several of the basic graphics classes and Java2D API classes.

- **Class Color** contains methods and constants for manipulating colors.
- **Class Font** contains methods and constants for manipulating fonts.
- **Class FontMetrics** contains methods for getting information about a font.
- **Class Polygon** contains methods for creating polygons.
- **Class Graphics** contains methods for drawing strings, lines, rectangles and other shapes.
- **Class Toolkit** contains methods for getting graphical information.

You’ll learn the methods of the above classes soon. Before manipulating figures, you first need to understand the coordinate system of Java. You draw graphical images on the screen by using coordinates that identify pixel locations. A coordinate system describes every point on the window. The upper-left corner of the window has the coordinates (0, 0). The x coordinate increases as you go to the right of the window; the y coordinate increases as you go to the bottom of the window. The coordinates are measured in the number of pixels. Thus the actual size of an entity drawn on the window depends on the size of the window and the resolution.

Graphics contexts and graphics objects

A graphics context is the medium through which drawing can be done. Thus, if you want to draw something on a particular window, you need to make a call to a method of the graphics context of that window. In Java, graphics contexts are modelled by the object Graphics <http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Graphics.html>.

The following reading gives an introduction to the object Graphics.
The points to note:

- All Java entities that allow drawings are descendents of the `Component` class [http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Component.html].

- `Graphics` is an abstract class. Thus you cannot create any `Graphics` object. In fact, when you create a Java entity that is drawing enabled, an object that is a subclass of `Graphics` is created by the constructor of that entity. Any drawing requests to that entity should then be forwarded to that `Graphics` object.

- the following are three of the methods of `Component`:
  ```java
  public void paint(Graphics g)
  public void repaint()
  public void update(Graphics g)
  ```

- The `paint` method is rarely called explicitly in a program. It is called automatically by the system when there is a need to re-draw the corresponding entity; for example when the entity reappears on the screen because another entity has been moved. The default action is to do nothing. Then, the programmer must override the `paint` method in order to draw the entity accordingly.

- The `repaint` method is the one called when you want to force a re-draw of a particular entity. The default action is to call the `update` method.

- The `update` method is rarely called explicitly in a program. The default action of `update` is to clear the background and then call `paint`. 
JFrame

In a Java GUI, the main component, which all other components depend on, is a container. A container is the outmost window of your application. Frame and Dialog are examples of a container. All GUI components must be place on a container through an add method. The following table lists some frequently used containers.

<table>
<thead>
<tr>
<th>Container</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container</td>
<td>the root of the container hierarchy, suitable for subclassing <a href="http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Container.html">http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Container.html</a></td>
</tr>
<tr>
<td>JDialog</td>
<td>a window suitable for dialog boxes <a href="http://java.sun.com/products/j2se/1.4/docs/api/javax/swing/JDialog.html">http://java.sun.com/products/j2se/1.4/docs/api/javax/swing/JDialog.html</a></td>
</tr>
<tr>
<td>JFrame</td>
<td>a top-level window with a decorated window <a href="http://java.sun.com/products/j2se/1.4/docs/api/javax/swing/JFrame.html">http://java.sun.com/products/j2se/1.4/docs/api/javax/swing/JFrame.html</a></td>
</tr>
<tr>
<td>JScrollPane</td>
<td>a container that scrolls its contents <a href="http://java.sun.com/products/j2se/1.4/docs/api/javax/swing/JScrollPane.html">http://java.sun.com/products/j2se/1.4/docs/api/javax/swing/JScrollPane.html</a></td>
</tr>
<tr>
<td>Window</td>
<td>a top-level window with no border and no menubar, suitable for popup menus. <a href="http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Window.html">http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Window.html</a></td>
</tr>
</tbody>
</table>

The following examples use JFrame as the container. A frame is a top-level container that has a title and a border and serves as the main window for a standalone Java GUI application. The rectangle that contains everything in your word processing software package is a Frame. Frame is a specific class in the java.awt package. All other GUI components — menus, scrollbars, buttons etc. are all inside a Frame. In Java 2 is JFrame, which is a class in javax.swing package. In the following examples for illustrating colors and fonts etc., JFrame is used as a container to ‘hold’ all figures and texts.

Figure 3.1 shows an example using JFrame.
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class JFrameExample {
    static public void main(String st[]) {
        JFrame app=new JFrame();
        app.setSize(400,400);
        app.show();
        app.addWindowListener(
            new WindowAdapter() {
                public void windowClosing( WindowEvent e) { System.exit(0); }
            }
        );
    }
}

Figure 3.1

Both setSize and show are member methods of JFrame, which are used to set the size of the frame and display it on the screen, as suggested by the names. The last statement that involves the calling of the addWindowListener is to tell the system that the program would terminate when the user closes the frame. You return to this method later when event-driven programs are introduced. The program will only display a window on the screen. Note that the frame displayed is empty, because the default paint method of JFrame is empty and we have not over-ridden it here.

You should note that the argument to the method addWindowListener is

    new WindowAdapter () {
    public void windowClosing (WindowEvent e) {
        System.exit(0); }
    }
}

In these few lines, a new instance of WindowAdapter is created. However, it also defines a method called windowClosing at the same time. Therefore, the object just created is actually, not a WindowAdapter, but a derived class of WindowAdapter that has the new definition of windowClosing.

You can of course define the new class separately. However, defining the class as shown in Figure 9.1 has the advantage that the method windowClosing can have access to even private members of JFrameExample.
Drawing shapes

Java’s Graphics class includes methods for drawing many different types of shapes — everything from straight lines to polygons. The Graphics class is introduced in the beginning of this subsection. As you may remember, when an object is displayed, the method paint() method will be called automatically. And a reference of a Graphics object is passed as its argument. The following lines are usually included at the top of any program that uses the drawing facilities of the AWT package:

```java
import java.awt.Graphics
import java.awt.*
```

The first line in the preceding imports only the Graphics class, whereas the second line imports all the classes included in the AWT package.

Some of the drawing methods provided by the Graphics class are summarized in the following table:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setColor()</td>
<td>sets the drawing color</td>
</tr>
<tr>
<td>getColor()</td>
<td>retrieves the current drawing color</td>
</tr>
<tr>
<td>getFont()</td>
<td>retrieves the currently used font</td>
</tr>
<tr>
<td>getFontMetrics()</td>
<td>retrieves information about the current font</td>
</tr>
<tr>
<td>setFont()</td>
<td>sets the font</td>
</tr>
<tr>
<td>drawLine()</td>
<td>draws a straight line</td>
</tr>
<tr>
<td>drawRect()</td>
<td>draws a hollow rectangle</td>
</tr>
<tr>
<td>fillRect()</td>
<td>draws a filled rectangle</td>
</tr>
<tr>
<td>drawRoundRect()</td>
<td>draws a hollow rectangle with rounded corners</td>
</tr>
<tr>
<td>fillRoundRect()</td>
<td>draws a filled rectangle with rounded corners</td>
</tr>
<tr>
<td>draw3DRect()</td>
<td>draws a hollow 3-D rectangle</td>
</tr>
<tr>
<td>fill3DRect()</td>
<td>draw a filled 3-D rectangle</td>
</tr>
<tr>
<td>drawOval()</td>
<td>draws a hollow oval</td>
</tr>
<tr>
<td>fillOval()</td>
<td>draws a filled oval</td>
</tr>
<tr>
<td>drawArc()</td>
<td>draws a hollow arc</td>
</tr>
<tr>
<td>fillArc()</td>
<td>draws a filled arc</td>
</tr>
<tr>
<td>drawPolygon()</td>
<td>draws a hollow polygon</td>
</tr>
</tbody>
</table>

You may want to check the detailed information about these methods at: [http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Graphics.html](http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Graphics.html).
Drawing lines

To draw a shape in an application’s display area, you need to overload the `paint` method of `JFrame`. This can be done in a number of ways. The example shown in Figure 9.2 defines a new class `MyFrame`, which is a subclass of `JFrame`. Then the `paint` method is overloaded as shown, which would draw two lines on the frame.

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

class MyFrame extends JFrame {
    public void paint(Graphics g) {
        g.drawLine(100,100,250,200);
        g.drawLine(150,150,50,50);
    }
}

public class Frame2 {
    static public void main(String st[]) {
        MyFrame app=new MyFrame();
        app.setSize(400,400);
        app.show();
        app.addWindowListener(
        new WindowAdapter() {
            public void windowClosing( WindowEvent e) {
                System.exit(0);
            }
        });
    }
}
```

Figure 3.2

Points to note:

- The first two parameters of the `drawLine` method of `Graphics` represent the x, y coordinates of the start point of the line. The remaining two parameters represent the x, y coordinates of the end point.

- We did not specify the colors, or width of the line, so the default values will be used. There are methods in `Graphics` which enable us to change these values.

- Although we have defined the `paint` method in `MyFrame` in the above example, it was never called explicitly in the program. The method is called by the system whenever there is a need to re-draw any instance of `MyFrame`, for example, when it is moved or its size is changed.
• All drawing requests are directed to the Graphics object passed to the paint method by the system.

Another way to write the program is to extend the public class directly from JFrame, as shown in Figure 9.3.

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

public class Frame2 extends JFrame {
    public void paint(Graphics g) {
        g.drawLine(100, 100, 250, 200);
        g.drawLine(150, 150, 50, 50);
    }
    static public void main(String st[])
    {
        Frame2 app = new Frame2();
        app.setSize(400, 400);
        app.show();
        app.addWindowListener();
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        }
    }
}
```

Figure 3.3

The two programs work in exactly the same way.

**Color control**

The Color class [http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Color.html] contains color constants and color methods. Every color is created from an RGB value (Red/Green/Blue). An RGB value is created from three integer or three floating-point parts. Each RGB part is either an integer value between 0 and 255 or a floating point value between 0.0 and 1.0.

Two Color constructors are given below:

```java
    public Color ( int r, int g, int b )
    public Color ( float r, float g, float b )
```
The `Color` methods `getRed`, `getGreen`, and `getBlue` return integer values between 0 and 255 representing the amount of red, green, and blue.

```java
public int getRed()
public int getBlue()
public int getGreen()
```

The `Graphics` method `getColor` returns a `Color` object representing the current color.

```java
public abstract Color getColor()
```

The `Graphics` method `setColor` sets the current color.

```java
public abstract void setColor(Color c)
```

The example in Figure 3.3 is changed to illustrate how to use the `setColor` method. The modified program is shown in Figure 3.4.

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

public class Frame2 extends JFrame {
    public void paint(Graphics g) {
        Color c=new Color(255,0,0);  //red
        g.setColor(c);
        g.drawLine(100,100,200,200);
        c=new Color(0,255,0);  //green
        g.setColor(c);
        g.drawLine(150,150,50,50);
    }
    static public void main(String st[]) {
        Frame2 app=new Frame2();
        app.setSize(400,400);
        app.show();
        app.addWindowListener();
        new WindowAdapter() {
            public void windowClosing( WindowEvent e)
            {
                System.exit(0);
            }
        }
    }
}
```

Figure 3.4
Points to note in Figure 3.4:

- The `setColor` method only affects the drawing requested that is executed after the calling of the method. Thus, we can use the `setColor` to change the color of the entities to be drawn whenever we want.

- The default color is black.

- The two statements

  ```java
c=new Color(0,255,0);
g.setColor(c);
```

  can be combined into one statement:

  ```java
g.setColor(new Color(0,255,0));
```

Reading


Font control

The `getDefaultToolkit` method is a static method of the `Toolkit` class <http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Toolkit.html>.

The `Toolkit` method `getFontList` returns the names of the system’s available fonts.

```java
public abstract String[] getFontList().
```

The program in Figure 3.5 can be used to show the available fonts in your system.

```java
import java.awt.*;

public class CheckFonts {
    public static void main(String arg[])
    {
        String fontList[]=Toolkit.getDefaultToolkit().
            getFontList();
        int i;
        for (i=0;i<fontList.length;i++)
        {
            System.out.println(fontList[i]);
        }
    }
}
```

Figure 3.5
In Java, the Font class <http://java.sun.com/products/j2se/1.4/docs/api/java/awt/Font.html> contains constants and methods for manipulating fonts.

The static constants `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC` are used to specify the font style and are defined in the `Font` class. The `Font` constructor takes three arguments: the font name, the font style, and the font size in points. A point is $1/72$ of an inch.

```java
public Font(String name, int style, int size)
```

The `Graphics` method `setFont` sets the current font. It takes a `Font` object as an argument.

```java
public abstract void setFont(Font f)
```

The `Graphics` method `getFont` returns a `Font` object representing the current font.

```java
public abstract Font getFont()
```

The `Font` method `getStyle` returns an integer value representing the current style.

```java
public int getStyle()
```

The `Font` method `getSize` returns the font size. The integer value returned represents the size of the font in points.

```java
public int getSize()
```

The `Font` method `getName` returns the current font name as a `String`.

```java
public String getName()
```

The `Font` method `isPlain` returns `true` if the current font style is plain. The `isBold` returns `true` if the current font style is bold. The `isItalic` returns `true` if the current font style is italic.

```java
public Boolean isPlain()
pUBLIC Boolean isBold()
pUBLIC Boolean isItalic()
```

The following example illustrates the usage of some methods listed above.

```java
```

```java```
// Demonstrating the Font constants, the Font constructor and the setFont method
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Fonts extends JFrame {
    public Fonts()
    {
        super ( "Using fonts" );
        setSize ( 420, 125 );
        show ();
    }

    public void paint ( Graphics g )
    {
        g.setFont ( new Font ( "Serif", Font.BOLD, 12 ) );
        g.drawString ( g.getFont().getName () + " " + g.getFont().getSize () + " point bold italic.", 20, 110 );
    }

    public static void main ( String args[] )
    {
        Fonts app = new Fonts ();
        app.addWindowListener ( new WindowAdapter() {
            public void windowClosing ( WindowEvent e )
            {
                System.exit (0);
            }
        } );
    }
}

Figure 3.6
You may refer to the text for a detailed description.

**Reading**
Deitel and Deitel, section 12.4, pages 580–85.
Self-test 3.1

1. Which method in the Graphics class returns the active font?
2. Which method of the Font class returns the font’s name?
3. Which method of the Font class returns the font’s height?
4. Why would we be interested in the height of the current font?
5. How do you get a reference to a FontMetrics object?
6. When would you use a FontMetrics object to obtain information about a font instead of using the Font object?

Self-test 3.2

1. How is Java’s graphical coordinate system organized?
2. What method of the Graphics class do you call to draw a line?
3. Rewrite the paint method of the above program to draw a red line from the upper-left corner to the point at (200, 100) and draw another horizontal line from (20, 120) to (250, 12) in blue.

Drawing rectangles

The Graphics method drawRect draws a rectangle with the specified upper-left corner, width and height.

```java
public void drawRect(int x, int y, int width, int height)
```

The Graphics method fillRect draws a filled rectangle with the specified upper-left corner, width and height. The current color is used to fill the rectangle.

```java
public abstract void fillRect(int x, int y, int width, int height)
```

The Graphics method clearRect draws a rectangle in the current background color over the specified rectangle.

```java
public abstract void clearRect(int x, int y, int width, int height)
```
<table>
<thead>
<tr>
<th>Self-test 3.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 What is the difference between the shapes drawn by the drawRect() and fillRect() methods?</td>
</tr>
<tr>
<td>2 What are the four arguments for the drawRect() method?</td>
</tr>
<tr>
<td>3 Rewrite the paint method of the above program to draw a red rectangle at (150,100) that is 200 pixels wide and 120 pixels high.</td>
</tr>
</tbody>
</table>

As all other drawing methods of Graphics are quite self-explanatory, they are not illustrated here. Please refer to the following textbook readings for detailed information of these methods.

<table>
<thead>
<tr>
<th>Reading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deitel and Deitel, sections 12.5–12.7, pages 585–94.</td>
</tr>
</tbody>
</table>
Basic graphical user interface components

In the last subsection, you created static UI, as the output of the program is always the same. It just ‘paints’ the output once when the application is executed. In this subsection, you create a user interface that enables users to interact with the programs. In the last subsection, figures are drawn when the JFrame is created. Now, you would like to have some buttons on your frame or container. These buttons are what is classified as a graphical user interface component. Next, you need to put a button on our container by using different layout managers. After putting a button or text field for input, what should you do when the button is pressed or there is text input? In Java, the graphics application is event-driven. Different actions, like moving the mouse, cause different events to occur. Hence, you learn how to capture events in Java.

Event-driven programs

When you studied your first programming course, the programs you wrote had the following characteristics:

- The flow of control of the programs depends on the input data only.
- The programs are designed to deal with one input channel at a time.

For example, consider that a program is designed to read a character from the keyboard, then a character from the serial port. Unfortunately, no character is available at the keyboard at the time, but one is available at the serial port. The program would still wait for the character from the keyboard and ignore the character from the serial port.

This mechanism is clearly not adequate for systems that have to respond to random external events, because we never know the exact sequence of these events. For example, consider your favourite word processor. At any time, you can either type a character to a file or drag the pointer to select a portion of the file. These two events can happen in any order, at any time. Therefore, you cannot hardcode the program to process these events in a particular order. Instead, you would like the program to be able to deal with any event when it arrives. This type of program is therefore known as event-driven.

In writing non-event-driven programs, a programmer would design the flow of control depending on the value of input data. This is not so in writing an event-driven program. Instead, a programmer would write the subroutine that would be called when an event arrives, then tell the system that when such an event arrives, it should invoke the subroutine accordingly. Such subroutines are called event-handlers. In fact, most of the programs presented previously in this unit are event-driven. For example, when the user changes the size of a JFrame window, a resize
event is generated. After the system has caught this event, the corresponding event handler for the resizing of JFrame would be invoked, which would then call the method paint to redraw the window.

**Controlling layout**

Layout managers are special objects that determine how elements of your application are organized in the application’s display. Depending on the order you ‘add’, the components, the size, shape, and placement of components will be different for different layout managers. Moreover, the layout manager will change the placement of components when the users modify the size of the container.

When you create an application or applet, Java automatically creates and assigns a default layout manager. In many of the applications and applets you’ve created so far, it’s the default layout manager that determined where the components appear. You can, however, create different types of layout manager in order to have better control of how your applications/applets look. Some of the more popular layout managers are listed below:

- **FlowLayout** [http://java.sun.com/products/j2se/1.4/docs/api/java/awt/FlowLayout.html]. This would arrange displayed components from left to right.

- **GridLayout** [http://java.sun.com/products/j2se/1.4/docs/api/java/awt/GridLayout.html]. This would arrange displayed components in a rectangular grid.

- **BorderLayout** [http://java.sun.com/products/j2se/1.4/docs/api/java/awt(BorderLayout.html)]. This would allow the programmer to add components in five regions of the container: North, East, South, West and Center.

These layout managers are represented by classes with the same names. In order to use a specific layout manager, you first need to create an instance of the desired layout class and then call the `setLayout()` method. After that, the layout manager is created for your application/applet. In the following sections, you see how to create these layout managers in detail.

In Java, JApplet [http://java.sun.com/products/j2se/1.4/docs/api/javax/swing/JApplet.html], JFrame, JWindow [http://java.sun.com/products/j2se/1.4/docs/api/java/swing/JWindow.html], and JDialog [http://java.sun.com/products/j2se/1.4/docs/api/java/swing/JDialog.html] etc. all produce a Container with `getContentPane()`. A container is where the components are placed, and thus a container needs a layout manager to arrange the components for it. Some default layout managers are listed in the following table:
In the following examples, buttons are used as the components to be added to the containers. The features of buttons will be described soon. At this time, you just need to know how to create buttons. Buttons can be created by simply calling the JButtons’s constructor, like this:

```java
JButton button = new JButton(string);
```

Here, `string` is the text string that you want to appear on the button.


**FlowLayout layout manager**

Layout managers are used to arrange components on a container. GUI components are placed on a container from left to right and from top to bottom. Components are placed in the order in which they’re added, one after the other in horizontal rows. When the layout manager reaches the right border of the container, it begins placing controls on the next row. In its default state, the FlowLayout manager centers controls on each row. However, you can set the alignment when you create the layout manager for your container. A FlowLayout may be right-, left-, or center-aligned. Center is the default alignment. Three FlowLayout constructors are given below:

<table>
<thead>
<tr>
<th>FlowLayout()</th>
<th>constructs a FlowLayout in which components are center aligned by default</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowLayout(int alignment)</td>
<td>constructs a FlowLayout with a specified alignment; the value for alignment is FlowLayout.RIGHT, FlowLayout.LEFT, or FlowLayout.CENTER.</td>
</tr>
<tr>
<td>public FlowLayout(int alignment, int horizontal_gap, int vertical_gap)</td>
<td>constructs a FlowLayout with a specified alignment; the value for alignment is FlowLayout.RIGHT, FlowLayout.LEFT, or FlowLayout.CENTER. The distance in pixels between components is specified by the last two arguments.</td>
</tr>
</tbody>
</table>
Container method `setLayout` specifies a new layout for a container. You’ll notice that with `FlowLayout` the components take on their ‘natural’ size. The following program shows how to use the `FlowLayout` manager:

```java
// Demonstrating FlowLayout alignments.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FlowLayoutDemo extends JFrame {
    private JButton left, center, right;
    private Container c;
    private FlowLayout layout;

    public FlowLayoutDemo()
    {
        super ( "FlowLayout Demo");

        //Create an instance of the layout class
        layout = new FlowLayout ();
        c = getContentPane();

        //create layout of the container
        c.setLayout ( layout );

        //Create three buttons , Left, Center and Right
        left = new JButton ( "Left");
        c.add ( left );

        center = new JButton("Center");
        c.add ( center );

        right = new JButton ( "Right");
        c.add ( right );

        setSize ( 300, 75 );
        show ();
    }

    public static void main ( String args[] )
    {
        FlowLayoutDemo app = new FlowLayoutDemo ();
        app.addWindowListener ( new WindowAdapter() {
            public void windowClosing ( WindowEvent e )
            {
                System.exit(0);
            }
        });
    }
}
```

Figure 3.7
If you run the above program, the buttons **Left, Center** and **Right** will be added from left to right. A container’s layout is set with method `setLayout` of class `Container`.

```java
c.setLayout ( layout );
```

**BorderLayout layout manager**

The `BorderLayout` layout manager arranges components into five areas: North, South, East, West, and Center. In `BorderLayout`, up to five GUI components can be used — one for each position. Two `BorderLayout` constructors are given below:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BorderLayout()</code></td>
<td>constructs a <code>BorderLayout</code> with no gaps between components</td>
</tr>
<tr>
<td><code>BorderLayout(int horizontalGap, int verticalGap)</code></td>
<td>constructs a <code>BorderLayout</code> with horizontal components separated by <code>horizontalGap</code> pixels and vertical components separated by <code>verticalGap</code> pixels</td>
</tr>
</tbody>
</table>

Without any other instruction, this manager takes whatever you add( ) to it and places it in the center, stretching the object all the way out to the edges. However, there is more to the `BorderLayout`. This layout manager has the concept of four border regions and a center area. When you add something to a container that is using a `BorderLayout` you can use the overloaded `add()` method that takes a constant value as its second argument. This value can be any of the following:

- `BorderLayout.NORTH` (top)
- `BorderLayout.SOUTH` (bottom)
- `BorderLayout.EAST` (right)
- `BorderLayout.WEST` (left)
- `BorderLayout.CENTER` (fill the middle, up to the other components or to the edges)

If you don’t specify an area to place the object, it defaults to `CENTER`. The following program demonstrates how to use the `BorderLayout layout manager`. 
// Demonstrating Borderlayout alignments.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class BorderLayoutDemo extends JFrame {
private JButton north, east, south, west, centre;
private BorderLayout layout;
private Container c;

public BorderLayoutDemo() {
    super ( "BorderLayout Demo");
    layout = new BorderLayout();
    c = getContentPane();
    c.setLayout ( layout );

    north = new JButton ( "North");
    c.add (north, BorderLayout.NORTH );

    east = new JButton("East");
    c.add (east, BorderLayout.EAST );

    south = new JButton ( "South");
    c.add (south, BorderLayout.SOUTH);

    west = new JButton("West");
    c.add (west, BorderLayout.WEST );

    centre = new JButton ( "Centre");
    c.add (centre, BorderLayout.CENTER);

    setSize ( 300, 300 );
    show ();
}

public static void main ( String args[] ) {
    BorderLayoutDemo app =new BorderLayoutDemo();
    app.addWindowListener (
    new WindowAdapter() {
        public void windowClosing ( WindowEvent e )
        {
            System.exit(0);
        }
    });
}
}

Figure 3.8

After the program has been run, a window with five buttons will be shown.
GridLayout layout manager

Java’s GridLayout manager organizes the display of your application/applet into a rectangular grid, similar to the grid used in a spreadsheet. Java then places the components you create for the application/applet into each cell of the grid, working from left to right and top to bottom. Each component is given the same size. Two GridLayout constructors are given below:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridLayout(int i, int j)</td>
<td>constructs a GridLayout of i rows and j columns</td>
</tr>
<tr>
<td>GridLayout(int i, int j, int h, int v)</td>
<td>constructs a GridLayout of i number of rows and j number of columns; each component is separated horizontally by h pixels and separated vertically by v pixels.</td>
</tr>
</tbody>
</table>
The following program shows how to use the `GridLayout` manager:

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

// Demonstrating GridLayout alignments.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class GridLayoutDemo extends JFrame {
    private JButton b[] = new JButton[6];
    private Container c;
    private String names[] =
    {"one", "two", "three","four","five","six"};
    private GridLayout grid;

    public GridLayoutDemo()
    {
        super("GridLayout Demo");
        grid = new GridLayout(2,3);
        c = getContentPane();
        c.setLayout( grid );

        for ( int i = 0; i <names.length; i++ ) {
            b[i] = new JButton( names[i]);
            c.add( b[i] );
        }
        setSize( 300, 150 );
        show();
    }

    public static void main ( String args[] )
    {
        GridLayoutDemo app =new GridLayoutDemo();
        app.addWindowListener ( new WindowAdapter()
        {
            public void windowClosing ( WindowEvent e )
            {
                System.exit(0);
            }
        });
    }
}
```

Figure 3.9
The GridLayout constructor

    grid = new GridLayout (2,3);

specifies a GridLayout with two rows and three columns and no gap space. The first JButton object in this example is added to the first column of the first row, and the next component is added to the second column of the first row etc. Thus, the three buttons one, two, and three will be in the first row, and the three buttons four, five, and six will be in the second row.

A detailed description of layout managers is in section 12.14 of your textbook. You may not understand all of the programs presented in this section, because we have not covered the event capturing mechanism in Java. You may defer this reading until you finish the next subsection about event capturing.

---

**Reading**

Deitel and Deitel, section 13.15, pages 654–62.

---

**Self-test 3.4**

1. Give the names of the layout managers just described.
2. Which layout manager is the default manager for a JFrame?
3. How does the BorderLayout manager organize components?
4. What is the meaning of the arguments of the GridLayout manager’s constructor?
5. What component positions can be used with a BorderLayout manager?
6. How do you add components to a frame that uses the BorderLayout manager?

---

**JPanel**

In the past few sub-sections, you were told of using layout managers to contain the layout of a frame. For example, you can use the BorderLayout to place five items in five different locations, namely NORTH, EAST, SOUTH, WEST and CENTER. However, it only allows you to place exactly one item in each position. What if we want to place two buttons in the NORTH position? In this case, we need to first put a panel in the position and then add the two buttons into this panel. The entity we are using is called JPanel. A JPanel works like the
contentpane of JFrame. So you can also use a layout manager to control its layout. You can also add some JPanels to it if you want the former to be further divided into several regions. Therefore, you can see that JPanels are most useful in separating different GUI components into groups. Consider the following figure:

There are three groups of buttons: the top 2, the middle 4 and the bottom 2. BorderLayout is used in the outmost frame. The top two buttons are in the NORTH position. The middle four buttons are in the CENTER position. The bottom two buttons are in the SOUTH position. As there is more than one component in each position, we need to add a JPanel in each of these positions. The buttons are then added to these JPanels correspondently. FlowLayout managers are used in the NORTH and SOUTH positions while a GridLayout manager is used in the CENTER position.

The following figure shows the hierarchy of the GUI components of the above example:
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Frame1 extends JFrame {
    JPanel contentPane;
    BorderLayout borderLayout1 = new BorderLayout();
    JPanel jPanel1 = new JPanel();
    JPanel jPanel2 = new JPanel();
    JPanel jPanel3 = new JPanel();
    JButton jButton1 = new JButton("jButton1");
    JButton jButton2 = new JButton("jButton2");
    JButton jButton3 = new JButton("jButton3");
    JButton jButton4 = new JButton("jButton4");
    GridLayout gridLayout1 = new GridLayout();
    JButton jButton5 = new JButton("jButton5");
    JButton jButton6 = new JButton("jButton6");
    JButton jButton7 = new JButton("jButton7");
    JButton jButton8 = new JButton("jButton8");

    //Construct the frame
    public Frame1() {
        contentPane = (JPanel) this.getContentPane();
        contentPane.setLayout(borderLayout1);
        this.setSize(new Dimension(400, 300));
        jPanel1.setBackground(Color.green);
        jPanel2.setBackground(Color.orange);
        jPanel3.setLayout(gridLayout1);
        gridLayout1.setColumns(2);
        gridLayout1.setRows(2);
        contentPane.add(jPanel1, BorderLayout.NORTH);
        jPanel1.add(jButton1, null);
        jPanel1.add(jButton2, null);
        contentPane.add(jPanel2, BorderLayout.SOUTH);
        jPanel2.add(jButton3, null);
        jPanel2.add(jButton4, null);
        contentPane.add(jPanel3, BorderLayout.CENTER);
        jPanel3.add(jButton5, null);
        jPanel3.add(jButton7, null);
        jPanel3.add(jButton6, null);
        jPanel3.add(jButton8, null);
        show();
    }
    public static void main(String args[]) {
        Frame1 frame=new Frame1();
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}

Figure 3.10
Activity 3.1

Modify the program in Figure 3.10 so that the window of the application becomes:

![Diagram of modified window with six buttons]

Capturing events

If you compile and run the above coding of layout manager subsection, you will find that nothing happens when the buttons are pressed. The reason is that you haven’t registered to Java that you are interested in the event ‘Button Pressed’, and you haven’t told Java what you want to do in response to that event. Java GUI programming is event-driven, so you have to specify the event of your interest and the action you require. You may have noticed that the ‘appearance of the components’ and the ‘implementation of the components (or action of the components)’ are completely separate in Java.

A Java Program should register a listener to a given GUI component if you want to do something on the events that occur in this component. However, if you’re not interested in, for example, whether the mouse is being moved over your button, you don’t register your interest in that event. In every program that has an event handler, you’ll see the following:

The event handler class should implement the ActionListener interface (<http://java.sun.com/products/j2se/1.4/docs/api/java/awt/event/ActionEvent.html>) as shown below:

```java
public class MyHandlerClass implements ActionListener
```
or be derived from a class that has implemented this interface.
The following code illustrates how to register an object of the event handling class as a listener upon one or more components.

```java
aComponent.addActionListener(instance_Of_MyHandlerClass);
```

The implementation of the methods in the listener interface. For example:

```java
public void actionPerformed(ActionEvent e) {
    ...//code that reacts to the action...
}
```

Buttons are used as an example to illustrate how Java captures events. First, just focus on the event of pressing a JButton. The first thing is to call the JButton's addActionListener() method to register the listener for this event.

```java
// Responding to button presses.
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Button1 extends JFrame {
    private JButton b1;

    public Button1 ()
    {
        super ( "Testing JButton ");

        // Create the Layout Manager
        Container c = getContentPane();
        c.setLayout(new BorderLayout());

        b1 = new JButton ( "Click Me");
        c.add ( b1 );

        //Create an instance of innner class ButtonHandler
        // use for button event handling
        // ButtonHandler implement the ActionListener
        // interface
        ButtonHandler handler = new ButtonHandler();
        b1.addActionListener ( handler );
        setSize ( 275, 100 );
        show ();
    }
}
```
public static void main ( String args[] )
{
    Button1 app = new Button1();

    app.addWindowListener ( new WindowAdapter () {
        public void windowClosing ( WindowEvent e )
        {
            System.exit (0);
        }
    });
}

// inner class for button event handling

private class ButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }
}

Figure 3.11

The private class ButtonHandler is the actual class for the event handler when the button is pressed. Its only action is to emit an audio beep. Then, an instance of ButtonHandler is registered with the system using the addActionListener method of the JButton.

Now, we’d like to see something change on the screen, so a new Swing component is introduced: the JTextField <http://java.sun.com/products/j2se/1.4/docs/api/javax/swing/JTextField.html>. This is a place to display a single line of text. Here is an example of how to use a JButton and a JTextField:
// Responding to button presses.
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Button2 extends JFrame {
    private JButton b2;
    private JTextField txt;

    public Button2 () {
        super ( "Testing JTextField and JButton ");

        // Create the Layout Manager
        Container c = getContentPane();
        c.setLayout ( new FlowLayout () );

        // constructor textfield with default sizing
        txt = new JTextField ( 10 );
        c.add ( txt );

        b2 = new JButton ( "Button");
        c.add ( b2 );

        //Create an instance of inner class ButtonHandler
        // use for button event handling
        // ButtonHandler implement the ActionListener
        // interface
        ButtonHandler handler = new ButtonHandler();
        b2.addActionListener ( handler );
        setSize ( 275, 100 );
        show ();
    }

    public static void main ( String args[] ) {
        Button2 app = new Button2();
        app.addWindowListener ( new WindowAdapter () {
            public void windowClosing ( WindowEvent e ) {
                System.exit (0);
            }
        });
    }

    // inner class for button event handling
    private class ButtonHandler implements ActionListener {
        public void actionPerformed ( ActionEvent e ) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name);
        }
    }
}

Figure 3.12
In this example, a text field is created along with a button. The action event handle of the button is implemented so that when it is pressed, it will get the label of the button using the `getText()` method and then put the label to the text field.

## Event and listener types

All Swing components include `addXXXListener()` and `removeXXXListener()` methods to add or remove appropriate types of listener. The following table lists the event types defined, the corresponding listener and the method defined for each listener interface.

<table>
<thead>
<tr>
<th>Event, listener interface and add- and remove-methods</th>
<th>Components supporting this event</th>
<th>Listener methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent ActionListener addActionListener() removeActionListener()</td>
<td>JButton, JList, JTextField, JMenuItem and its derivatives including JCheckBoxMenuItem, JMenu, and JpopupMenu.</td>
<td>actionPerformed()</td>
</tr>
<tr>
<td>AdjustmentEvent AdjustmentListener addAdjustmentListener() removeAdjustmentListener()</td>
<td>JScrollBar and anything you create that implements the Adjustable interface.</td>
<td>adjustmentValueChanged()</td>
</tr>
<tr>
<td>ComponentEvent ComponentListener addComponentListener() removeComponentListener()</td>
<td>*Component and its derivatives, including JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileChooser, JFrame, JLabel, JPanel, JTextArea, and JTextField.</td>
<td>componentHidden() componentMoved() componentResized() componentShown()</td>
</tr>
<tr>
<td>ContainerEvent ContainerListener addContainerListener() removeContainerListener()</td>
<td>Container and its derivatives, including JPanel, JApplet, JScrollPane, Window, JDialog, JFileChooser, JFrame.</td>
<td>componentAdded() componentRemoved()</td>
</tr>
<tr>
<td>FocusEvent FocusListener addFocusListener() removeFocusListener()</td>
<td>Component and derivatives*.</td>
<td>focusGained() focusLost()</td>
</tr>
<tr>
<td>KeyEvent KeyListener addKeyListener() removeKeyListener()</td>
<td>Component and derivatives*.</td>
<td>keyPressed() keyReleased() keyTyped()</td>
</tr>
<tr>
<td>MouseEvent (for both clicks and motion) MouseListener addMouseListener() removeMouseListener()</td>
<td>Component and derivatives*.</td>
<td>mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased()</td>
</tr>
</tbody>
</table>
### MouseEvent\(^1\) (for both clicks and motion)

**MouseListener**
- `addMouseListener()`
- `removeMouseListener()`

**Component** and derivatives*.
- `mouseDragged()`
- `mouseMoved()`

### WindowEvent

**WindowListener**
- `addWindowListener()`
- `removeWindowListener()`

**Window** and its derivatives, including JDialog, JFileChooser, and JFrame.
- `windowActivated()`
- `windowClosed()`
- `windowClosing()`
- `windowDeactivated()`
- `windowDeiconified()`
- `windowIconified()`
- `windowOpened()`

### TextEvent

**TextListener**
- `addTextListener()`
- `removeTextListener()`

Anything derived from JTextComponent, including JTextArea and JTextField.
- `textValueChanged()`

---

You may refer to Figures 13.5 and 13.6 of your textbook for the classes hierarchy. After reading the above subsection and the textbook, you should make sure you understand the difference between **Event Source**, **EventListener** and **Listener Methods**.

---

**Reading**


---

**Activity 3.2**

Write an application that displays a blue circle centred within the display area. Put four buttons at the top, bottom, left and right of the display area. You may like to use the BorderLayout layout manager to achieve this layout. When the user clicks on any of the four buttons, move the circle a few pixels in the indicated direction.

---

**Adapter class**

An event listener interface may define more than one method. For example, something you must always do when creating an application is to provide a WindowListener to the JFrame so that when you get the 'window closing' event you can call `System.exit()` to exit the application. However, WindowListener is an interface. It responds to several different types of window event, most of which are not required.

---

\(^1\) There is no MouseMotionEvent even though it seems like there ought to be. Clicking and motion is combined into MouseEvent, so this second appearance of MouseEvent in the table is not an error.
in all the programs. In order to implement this interface, you might have to create an event-listener class that has several empty methods, like the one shown here:

```java
public void WindowClosed (WindowEvent e) {}
```

In order to save a programmer’s time in having to write empty methods for conditions that programs do not need to deal with, Java provides Adapter classes. Each adapter provides default empty methods for each of the interface methods. These are convenience classes that implement the various methods with empty curly braces, {}. By subclassing an Adapter class, you can override just the methods of interest to you.

**Reading**

Deitel and Deitel, section 13.13, pages 646–51.

**Graphical user interface components**

Up to now, you have used JButton and JTextField. Java has other components, as listed in the following table. This subsection introduces these components in detail. A GUI component (or widget) is a visual object with which the user may interact via the mouse or the keyboard.

Several GUI components are listed below:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JLabel</td>
<td>an area where read-only text can be displayed</td>
</tr>
<tr>
<td>JButton</td>
<td>an area that triggers an event when clicked</td>
</tr>
<tr>
<td>JTextField</td>
<td>an area in which the user inputs data from the keyboard; the area can also display information</td>
</tr>
<tr>
<td>JComboBox</td>
<td>a drop-down list of items from which the user can make a selection by clicking an item in the list or by typing into the box, if permitted</td>
</tr>
<tr>
<td>JCheckBox</td>
<td>a boolean GUI component that is either selected or not selected</td>
</tr>
<tr>
<td>JList</td>
<td>an area where a list of items is displayed from which the user can make a selection by clicking once on any element in the list; double-clicking an element in the list generates an action event</td>
</tr>
<tr>
<td>JPanel</td>
<td>a container in which components can be placed</td>
</tr>
</tbody>
</table>

Detailed information about these components can be found at:

The classes that are used to create GUI components are part of the Swing GUI components. These components are from the package java.swing. Swing is JavaSoft’s direction for GUI development. The original GUI components from the java.awt package are tied to the local platform’s graphical user interface capabilities. The Swing components are platform-independent. In other words, their appearance will be the same in different platforms.

**JLabel**

Labels are used to provide text instructions or information on a GUI. A JLabel displays a single line of read-only text. Three JLabel constructors are given below:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JLabel()</td>
<td>constructs an empty JLabel</td>
</tr>
<tr>
<td>JLabel(String s)</td>
<td>constructs a JLabel that displays the text with the default left-justified alignment</td>
</tr>
<tr>
<td>JLabel(String s, int alignment)</td>
<td>constructs a JLabel that displays the text s with the specified alignment; possible alignments are SwingConstants.LEFT, SwingConstants.CENTER, and SwingConstants.RIGHT</td>
</tr>
</tbody>
</table>

Three JLabel methods are given below:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getText()</td>
<td>returns the Label text</td>
</tr>
<tr>
<td>void setText(String s)</td>
<td>sets the Label text</td>
</tr>
<tr>
<td>void setHorizontalAlignment(int alignment)</td>
<td>sets the JLabel alignment to SwingConstants.LEFT, SwingConstants.CENTER, and SwingConstants.RIGHT</td>
</tr>
</tbody>
</table>
The following program illustrates how to use a JLabel:

```java
// Demonstrating LabelTest
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class LabelTest extends JFrame {
    private JLabel label1;

    public LabelTest() {
        super( "Testing JLabel");
        Container c = getContentPane();
        c.setLayout ( new FlowLayout () );
        // with a string argument
        label1 = new JLabel("Label with text");
        c.add( label1 );
        setSize(300,300);
        show();
    }

    public static void main ( String args[] )
    {
        LabelTest app =new LabelTest();
        app.addWindowListener ( new WindowAdapter() {
            public void windowClosing ( WindowEvent e )
            {
                System.exit(0);
            }
        });
    }
}
```

Figure 3.13

You may refer to Figure 13.4 of your textbook. The example will create a JLabel using three different constructors. You will also learn how to display an image using ImageIcon class.

**Reading**

Deitel and Deitel, section 13.3, pages 616–19.
Self-test 3.5

1. What are the three constructors of JLabel?
2. What are the two arguments required by the JLabel’s constructor?
3. What happens after clicking a label?

JTextField

A JTextField is a single-line area in which the user can enter text from the keyboard, or text can simply be displayed. When the user types data into a JTextField and presses the Enter key, an action event is generated.

Four JTextField constructors are given below:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTextField()</td>
<td>constructs a JTextField object</td>
</tr>
<tr>
<td>JTextField(int columns)</td>
<td>constructs an empty JTextField object with the specified number of columns</td>
</tr>
<tr>
<td>JTextField(String s)</td>
<td>constructs a JTextField object displaying s</td>
</tr>
<tr>
<td>JTextField(String s, int columns)</td>
<td>constructs a JTextField object displaying s in the specified number of columns</td>
</tr>
</tbody>
</table>

Some of the methods of TextField:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getText()</td>
<td>This method is inherited from JTextComponent. It returns the text in the text field.</td>
</tr>
<tr>
<td>void setEditable(boolean b)</td>
<td>This method is inherited from JTextComponent. It sets the specified boolean parameter to indicate whether or not this TextField should be editable. A PropertyChange event (‘editable’) is fired when the state is changed.</td>
</tr>
</tbody>
</table>

The following program illustrates how to use a JTextField:
// Demonstrating the JTextField class.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MyTextField extends JFrame {
    private JTextField text1, text2;
    private TextFieldHandler handler;

    public MyTextField()
    {
        super ( "Testing JTextField");
        Container c = getContentPane();
        c.setLayout ( new FlowLayout() );

        // constructor textfield with default
        // sizing
text1 = new JTextField ( 10 );
c.add ( text1 );

        // constructor textfield with default text
        // and sizing
text2 = new JTextField ("Uneditable",10);
text2.setEditable ( false );
c.add ( text2 );

        TextFieldHandler handler = new TextFieldHandler();
text1.addActionListener ( handler );
text2.addActionListener ( handler );

        setSize ( 325, 100 );
        show ();
    }

    public static void main ( String args[] )
    {
        MyTextField app = new MyTextField ();
        app.addWindowListener ( new WindowAdapter () {
            public void windowClosing ( WindowEvent e )
            {
                System.exit(0);
            }
        });
    }

    private class TextFieldHandler implements ActionListener {
        public void actionPerformed ( ActionEvent e )
        {
            String s = "";
            if ( e.getSource() == text1 )
                s = "text1 : " + e.getActionCommand();
            else if ( e.getSource() == text2 )
                s = "text2 : " + e.getActionCommand();

            JOptionPane.showMessageDialog ( null, s );
        }
    }
}
Points to note in the program in Figure 3.13:

- Two text fields were created — one editable and the other not editable.

- The same event handler is used for the action events occurring in the two textfields. Therefore, you need to use the getSource method to determine the source of the event.

Now you may refer to the program in Figure 13.7 of your textbook. In that program, besides JTextField, it illustrates how to use JPasswordField.

**Reading**

Deitel and Deitel, section 13.5, pages 620–25.

**Self-test 3.6**

What are the arguments of the TextField’s constructor?

**JButton**

A button is a component the user clicks to trigger a specific action.

A Java program can use several types of button, including command buttons, check boxes, toggle buttons and radio buttons. A command button generates a button event when the user clicks the button with the mouse. The following is a more complicated example in using JButton:
// Creating command buttons.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MyButton extends JFrame {
    private Button commandButton1, commandButton2;

    public MyButton ()
    {
        super ( "Testing buttons");

        Container c = getContentPane ();
        c.setLayout ( new FlowLayout () );

        // create buttons
        commandButton1 = new Button( "Button 1" );
        c.add ( commandButton1 );

        commandButton2 = new Button( "Button 2" );
        c.add ( commandButton2 );

        ButtonHandler handler = new ButtonHandler();
        commandButton1.addActionListener ( handler );
        commandButton2.addActionListener ( handler );

        setSize ( 275, 100 );
        show ();
    }

    public static void main ( String args[] )
    {
        MyButton app = new MyButton();
        app.addWindowListener ( new WindowAdapter() {
            public void windowClosing(WindowEvent e )
            {
                System.exit(0);
            }
        });
    }

    private class ButtonHandler implements ActionListener
    {
        public void actionPerformed ( ActionEvent e )
        {
            JOptionPane.showMessageDialog ( null,
                "You pressed: " + e.getActionCommand() );
        }
    }
}
Points to note:

- Two buttons will be displayed.

- The same event handler ButtonHandler is used for the action events generated from the two buttons. Therefore, the `getActionCommand` method of `ActionEvent` is used to get the identity of the event source.

In Figure 13.10 of your textbook, the program demonstrates the command buttons and action events. Figure 13.9 gives a button hierarchy. You should know that besides `JButton`, the other types of button available are: command buttons, check boxes, toggle buttons and radio buttons. Not all of these buttons are covered in this unit, but as a Java programmer, you should know of their existence.

Reading


Activity 3.3

Write an application with three buttons: 'Disable Button A', 'Button A' and 'Enable Button A'. When the user presses 'Disable Button A' both 'Disable Button A' and 'Button A' will be disabled, and only 'Enable Button' will be enabled. Then, if the user presses 'Enable Button' it will disable 'Enable Button' and enable the other two.

Self-test 3.7

What is the single argument of the `JButton`'s constructor?
**JCheckBox**

Check boxes provides a way to make a single on/off choice. It consists of a tiny box and a label. You’ll normally create a JCheckBox using a constructor that takes the label as an argument. You can get and set the state, and get and set the label if you want to read, or change it after the JCheckBox has been created.

Whenever a JCheckBox is set or cleared, an ItemEvent occurs, which you can capture in the same way you do for a button, by using an ItemListener. ItemEvent method getStateChange returns a Boolean representing the state of a JCheckbox. The following example illustrates how to capture events from a check box.

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

public class MyCheckBox extends JFrame {
    private JTextField t;
    private JCheckBox cb1, cb2, cb3;

    public MyCheckBox ()
    {
        super ( "JCheckBox Test");
        Container c = getContentPane();
        c.setLayout ( new FlowLayout() );
        t = new JTextField ("Watch the changes ", 30 );
        c.add ( t );

        cb1 = new JCheckBox("Check Box 1");
        cb2 = new JCheckBox("Check Box 2");
        cb3 = new JCheckBox("Check Box 3");

        c.add ( cb1 );
        c.add ( cb2 );
        c.add ( cb3 );

        CheckBoxHandler handler = new CheckBoxHandler();
        cb1.addItemListener ( handler );
        cb2.addItemListener ( handler );
        cb3.addItemListener ( handler );

        addWindowListener(
            new WindowAdapter() {
                public void windowClosing ( WindowEvent e )
                {
                    System.exit (0);
                }
            });
        setSize ( 350, 150 );
        show ();
    }
}
public static void main ( String args[] )
{
    new MyCheckBox();
}

private class CheckBoxHandler implements ItemListener {
    public void itemStateChanged ( ItemEvent e )
    {
        if ( e.getSource() == cb1 )
            if ( e.getStateChange() == ItemEvent.SELECTED )
                t.setText("Box 1 Set\n");
            else
                t.setText("Box 1 Cleared\n");
        else if ( e.getSource() == cb2 )
            if ( e.getStateChange() == ItemEvent.SELECTED )
                t.setText("Box 2 Set\n");
            else
                t.setText("Box 2 Cleared\n");
        else if ( e.getSource() == cb3 )
            if ( e.getStateChange() == ItemEvent.SELECTED )
                t.setText("Box 3 Set\n");
            else
                t.setText("Box 3 Cleared\n");
    }
}

Figure 3.16

The String passed to the constructor is the check box label that appears to the right of the JCheckBox by default.

    cb1 = new JCheckBox(“Check Box 1”),
    cb2 = new JCheckBox(“Check Box 2”),
    cb3 = new JCheckBox(“Check Box 3”);

When the user clicks a JCheckBox, an ItemListener method sends the name of the selected JCheckBox and its current state to the JTextField. Hence, the changes of state of the check box are captured by the text field.

You may refer to Section 13.8 of your textbook. That section also covers JRadioButton. Both JRadioButton and JCheckBox generate ItemEvents when they are clicked. You may refer to the textbook for details.
**Self-test 3.8**

In a container with several checkboxes, how can you determine which checkbox is being clicked?

**JList and JScrollPane**

In Java, scrolling is supported by JScrollPane. Components are simply wrapped in a JScrollPane to control scrolling when too much text is placed on the screen.

A JList occupies a fixed number of lines on a screen all the time and doesn’t change. A JList allows single-selection list or multiple selection list. To perform multiple-selection, you have to press the Control key while clicking the selected items. List method getSelectedValues returns the item clicked by the user.
// Demonstrating JList

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

public class MyJList extends JFrame {
    private JTextField t;
    private JList items;

    private String itemsNames[] = {
        "one", "two", "three", "four", "five" 
    };

    public MyJList () {
        super ( "JList Test");
        Container c = getContentPane();
        c.setLayout ( new FlowLayout() );

        t = new JTextField ("Watch the changes ", 20 );
        c.add ( t );

        items = new JList ( itemsNames );
        items.setVisibleRowCount (3);

        // allow single selection
        items.setSelectionMode (ListSelectionModel.SINGLE_SELECTION );

        // add a JScrollPane contains the JList
        c.add ( new JScrollPane ( items ));

        items.addListSelectionListener (new JListHandler());

        // addWindowListener
        new WindowAdapter() {
            public void windowClosing ( WindowEvent e ) {
                System.exit (0);
            }
        };

        setSize ( 300, 150 );
        show ();
    }

    public static void main ( String args[] ) {
        new MyJList();
    }

    private class JListHandler implements ListSelectionListener {
        public void valueChanged ( ListSelectionEvent e ) {
            t.setText( items.getSelectedValues().toString() );
        }
    }
}

Figure 3.17
The following line adds a new instance of class JScrollPane to the content pane.

```java
c.add ( new JScrollPane ( items ));
```

The following line uses JList method addListSelectionListener to register an object that implements ListSelectionListener.

```java
item.addListSelectionListener ( JListHandler );
```

You can refer to Fig. 13.15 of your textbook. The example will create a multiple-selection list using SINGLE_INTERVAL_SELECTION and MULTIPLE_INTERVAL_SELECTION. You can try to execute the program. Try using the selection techniques described.

**Reading**


**Activity 3.4**

You may modify the example in Fig.13.15 so that it uses MULTIPLE_INTERVAL_SELECTION as its setSelectionMode to feel the difference.

**Mouse events handling**

Up until now, you have responded to events generated by Java components like buttons, text fields, and list boxes. You’ve yet to examine how to respond to events generated by the most basic of a computer’s controls, the mouse and the keyboard. Because virtually every computer has these important hardware controls, you can confidently take advantage of them in your applets to collect various types of input.

In this section, you learn how to handle mouse and keyboard events in Java applications. A mouse event occurs when the user interacts with the mouse. Mouse events are broken into two groups — mouse motion events and all other mouse events — so that an object can listen for mouse events such as clicks without requiring the system overhead necessary for generating and forwarding mouse motion events, which tend to occur frequently. Interfaces MouseListener, MouseMotionListener, are provided for handling mouse events.
Some useful methods in interface `MouseListener` are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void mousePressed (MouseEvent e)</code></td>
<td>called when a mouse button is pressed with the mouse cursor on a component</td>
</tr>
<tr>
<td><code>void mouseClicked (MouseEvent e)</code></td>
<td>called when a mouse button is pressed and released on a component without moving the mouse cursor</td>
</tr>
<tr>
<td><code>void mouseReleased (MouseEvent e)</code></td>
<td>called when a mouse button is released after being dragged; this event is always preceded by a <code>mouseDragged</code> event</td>
</tr>
<tr>
<td><code>void mouseEntered (MouseEvent e)</code></td>
<td>called when the mouse cursor enters the bounds of a component</td>
</tr>
<tr>
<td><code>void mouseExited (MouseEvent e)</code></td>
<td>called when the mouse cursor leaves the bounds of a component</td>
</tr>
</tbody>
</table>

Methods in interface `MouseMotionListener` are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void mouseDragged (MouseEvent e)</code></td>
<td>Called when a mouse button is pressed and the mouse is moved; this event is always preceded by a call to <code>mousePressed</code>.</td>
</tr>
<tr>
<td><code>void mouseMoved (MouseEvent e)</code></td>
<td>Called when the mouse is moved with the mouse cursor on a component.</td>
</tr>
</tbody>
</table>

Methods `addMouseListener` and `addMouseMotionListener` of class `Component` register `MouseListener` and `MouseMotionListener` event listeners for any `GUI` component. `MouseEvent` methods `getX` and `getY` return information about the mouse event that occurred including the x and y coordinates where the event occurred.

```
// Demonstrating mouse events.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseTracker extends JFrame
    implements MouseListener, MouseMotionListener
{
    private JLabel statusBar;

    public MouseTracker()
    {
        super ("Demonstrating Mouse Events");
        statusBar = new JLabel();
        getContentPane().add(statusBar,BorderLayout.SOUTH );
    }
```
addMouseListener(this);
addMouseMotionListener(this);

setSize ( 275, 100 );
show();

// MouseListener event handlers
public void mouseClicked( MouseEvent e )
{ statusBar.setText("Clicked [" + e.getX() + "," +
e.getY() + "]"); }

public void mousePressed( MouseEvent e )
{ statusBar.setText("Pressed [" + e.getX() + "," +
e.getY() + "]"); }

public void mouseReleased( MouseEvent e )
{ statusBar.setText("Released [" + e.getX() + "," +
e.getY() + "]"); }

public void mouseEntered( MouseEvent e )
{ statusBar.setText("Mouse in window"); }

public void mouseExited( MouseEvent e )
{ statusBar.setText("Mouse outside window"); }

// MouseMotionListener event handlers
public void mouseDragged( MouseEvent e )
{ statusBar.setText("Dragging [" + e.getX() + "," +
e.getY() + "]"); }

public void mouseMoved( MouseEvent e )
{ statusBar.setText("Moving [" + e.getX() + "," +
e.getY() + "]"); }

public static void main ( String args[])
{
MouseTracker app = new MouseTracker();
app.addWindowListener ( new WindowAdapter() {
public void windowClosing ( WindowEvent e)
{ System.exit(0); }
}
);
}

Figure 3.18

The lines register MouseTracker window object as the listener for its own mouse events.

addMouseListener(this);
addMouseMotionListener(this);
When any of the five events occur, they display a message in `statusBar`.

In your textbook, Fig 13.16 of section 12.16 summarizes the `MouseListener` and `MouseMotionListener` interface methods. Make sure you understand the functions of each of these methods.

**Reading**

Deitel and Deitel, section 13.12, pages 642–46.

---

**Activity 3.5**

Write a simple program with a blank display area initially. A line will be drawn as the user drags the mouse across the Frame. (Hint: you only need to modify the `mouseDragged` and `mousePressed` methods and give an empty method for other unused methods of `MouseListener` interface and `MouseMotionListener` interface.)

---

**Self-test 9.9**

When a mouse-related message is received, how do you determine the coordinates of the mouse at the time the event happens?

Key events are generated when keys on the keyboard are pressed and released. A class that implements `KeyListener` must provide definitions for methods `keyPressed`, `keyReleased` and `keyTyped`, each of which receives a `KeyEvent` as its argument. Method `keyPressed` is called in response to pressing an action key (an arrow key, Home, End, Page Up, Page Down, a function key, Num Lock, Print Screen, Scroll Lock, Caps Lock, and Pause). Method `keyTyped` is called in response to pressing any other key on the keyboard. Method `keyReleased` is called after the key is released in any `keyPressed` and `keyTyped` event.

Method `addKeyListener` of `Component` registers a `KeyListener` for a component.

The following are useful methods of `KeyEvent` <http://java.sun.com/products/j2se/1.4/docs/api/java/awt/event/KeyEvent.html>. 


int getKeyCode() returns the code of the key pressed

static String getKeyText(int code) returns the name of the key corresponding to the input parameter.

boolean isActionKey() returns whether or not the pressed key is an action key like Page Up, F1, etc.

static String getKeyModifierText(int code) returns a String describing the modifier key(s), such as 'Shift', or 'Ctrl+Shift'

The following are useful methods of KeyEvent which are inherited from InputEvent:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int getModifiers()</td>
<td>returns the modifiers flag for this event</td>
</tr>
<tr>
<td>boolean isAltDown()</td>
<td>These two methods return whether the Alt or Shift key is pressed.</td>
</tr>
<tr>
<td>boolean isShiftDown()</td>
<td></td>
</tr>
</tbody>
</table>

```java
// Demonstrating keystroke events.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class KeyDemo extends JFrame implements KeyListener {
    private String line1 = "";
    private String line2 = "";
    private String line3 = "";
    private JTextArea textArea;

    public KeyDemo() {
        super ("Demonstrating Keystroke Events");

        textArea = new JTextArea ( 10, 15 );
        textArea.setText ("Press any key");
        textArea.setEnabled (false);

        // allow frame to process Key events
        addKeyListener( this );

        getContentPane().add( textArea );
        setSize ( 350, 100 );
        show();
    }

    public void keyPressed( KeyEvent e ) {
        line1 = "Key pressed: " + e.getKeyText( e.getKeyCode() ) + 
        setLines2and3( e );
    }
}
```
```java
public void keyReleased( KeyEvent e )
{
    line1 = "Key released: " +
              e.getKeyText( e.getKeyCode() );
    setLines2and3( e );
}
public void keyTyped( KeyEvent e )
{
    line1 = "Key typed: " + e.getKeyChar();
    setLines2and3( e );
}
private void setLines2and3( KeyEvent e )
{
    line2 = "This key is " +
            ( e.isActionKey() ? "" : "not " ) +
            "an action key";
    String temp =
                   e.getKeyModifiersText(e.getModifiers());
    line3 = "Modifier keys pressed: " +
            ( temp.equals( "" ) ? "none" : temp );
    textarea.setText (line1 + "\n" + line2 + "\n"
          + line3 + "\n");
}
public static void main ( String args[] )
{
    KeyDemo app = new KeyDemo();
    app.addWindowListener(
        new WindowAdapter() {
            public void windowClosing (WindowEvent e )
            {
                System.exit(0);
            }
        });
}
```

**Figure 3.19**

You may refer to Section 13.14 of your textbook for a detailed description. Besides giving a sample program, it gives you the sample output. You should make sure you agree with the details presented.

**Reading**

Summary

In this unit, you first learned Java’s capabilities of drawing 2D shapes, including lines, rectangles, ovals and arcs. You can also set the color of the Graphics class. You can create fonts you need by using methods from the Font and FontMetrics classes.

Besides drawing figures, several graphical user interfaces components are presented. Java supports different layout managers when you create a layout for the components as they are added to the container. As Java is an event-driven programming, make sure you understand Event Source, Event and Event Listener.

Since the keyboard and the mouse are important devices for accepting input from the user, capturing the event of these devices is illustrated.

Should you want to learn more about Java’s graphical user interface components, you may refer to the textbook.
Feedback on activities

Activity 3.1

We need to add a JPanel at the position of the original JButton8. Then, four jButtons are added into this new JPanel. A GridLayout manager would be used in the JPanel.

The code is shown below:

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

public class Frame1 extends JFrame {
    JPanel contentPane;
    BorderLayout borderLayout1 = new BorderLayout();
    JPanel jPanel1 = new JPanel();
    JPanel jPanel2 = new JPanel();
    JPanel jPanel3 = new JPanel();
    JPanel jPanel4 = new JPanel();
    JButton jButton1 = new JButton("jButton1");
    JButton jButton2 = new JButton("jButton2");
    JButton jButton3 = new JButton("jButton3");
    JButton jButton4 = new JButton("jButton4");
    GridLayout gridLayout1 = new GridLayout();
    GridLayout gridLayout2 = new GridLayout();
    JButton jButton5 = new JButton("jButton5");
    JButton jButton6 = new JButton("jButton6");
    JButton jButton7 = new JButton("jButton7");
    JButton jButton8 = new JButton("jButton8");
    JButton jButton9 = new JButton("jButton9");
    JButton jButton10 = new JButton("jButton10");
    JButton jButton11 = new JButton("jButton11");

    //Construct the frame
    public Frame1() {
        contentPane = (JPanel) this.getContentPane();
        contentPane.setLayout(borderLayout1);
        this.setSize(new Dimension(400, 300));
        jPanel1.setBackground(Color.green);
        jPanel2.setBackground(Color.orange);
        gridLayout1.setColumns(2);
        gridLayout1.setRows(2);
        contentPane.add(jPanel1, BorderLayout.NORTH);
        jPanel1.add(jButton1, null);
        jPanel1.add(jButton2, null);
        contentPane.add(jPanel2, BorderLayout.SOUTH);
        jPanel2.add(jButton3, null);
        jPanel2.add(jButton4, null);
        contentPane.add(jPanel3, BorderLayout.CENTER);
        jPanel3.add(jButton5, null);
        jPanel3.add(jButton7, null);
        jPanel3.add(jButton6, null);
        jPanel3.add(jPanel4, null);
    }
}````
gridLayout2.setRows(2);
gridLayout2.setColumns(2);
jPanel4.setLayout(gridLayout2);
jPanel4.add(jButton8, null);
jPanel4.add(jButton9, null);
jPanel4.add(jButton10, null);
jPanel4.add(jButton11, null);
show();
}
public static void main(String args[]) {
    Frame1 frame=new Frame1();
    frame.addWindowListener( new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        } 
    });
}

Activity 3.2

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

public class BorderLayoutAct extends JFrame {
    private JButton north, east, south, west;
    private BorderLayout layout;
    private Container c;
    private int x = 100;
    private int y = 100;
    private CustomPanel panel;

    public BorderLayoutAct() {
        super ("BorderLayout Demo");
        layout = new BorderLayout();
        c = getContentPane();
        c.setLayout (layout);
        ButtonHandler handler = new ButtonHandler();
        north = new JButton("North");
        c.add(north, BorderLayout.NORTH);
        north.addActionListener(handler);
        east = new JButton("East");
        c.add(east, BorderLayout.EAST);
        east.addActionListener(handler);
        south = new JButton("South");
        c.add(south, BorderLayout.SOUTH);
        south.addActionListener(handler);
    }
}
public static void main( String args[] )
{
    MultipleSelection app = new MultipleSelection();

    app.addWindowListener(
        new WindowAdapter()
        {
            public void windowClosing( WindowEvent e )
            {
                System.exit( 0 );
            }
        });
    }
}

Activity 3.5

//you need to modify the following two methods and
//give an empty methods for other unused methods in the
//MouseLister and MouseMotionLister interfaces.
//Also, you need to include two private
//int last_x and last_y

public void mousePressed(MouseEvent e )
{
    last_x = e.getX();
    last_y = e.getY();
}

public void mouseDragge(MouseEvent e )
{
    Graphics g = this.getGraphics();
    int x = e.getX();
    int y = e.getY();
    g.drawLine ( last_x, last_y, x );
    last_x = x;
    last_y = y;
}
Solutions to self-tests

Self-test 3.1

1. The Graphics class’s getFont() method returns a reference to the currently active font object, y.

2. The Font class’s getName() method returns the font’s name.

3. The Font class’s getHeight() method returns the font’s height.

4. In order to properly space lines of text.

5. The Graphics class’s getFontMetrics() method returns a reference to a FontMetrics object. The Font object for which you want the metrics is the method’s single argument.

6. A FontMetrics object returns more detailed information about a font. The Font class only gives general information about a font.

Self-test 3.2

1. The origin of Java’s graphical coordinate system is in the upper left corner with values of X increasing to the right and values of Y increasing downwards.

2. The drawLine() method draws a line.

3. ```java
public void paint( Graphics g )
{
    //set color to red
    g.setColor( Color.red)
    // draw a line from (0,0) to (200,100)
    g.drawLine( 0, 0, 200, 100 );
    //set color to blue
    g.setColor( Color.blue)
    // draw a line from (20,120) to (250,120)
    g.drawLine( 20, 120, 250, 120 );
}
```  

Self-test 3.3

1. The drawRect() method draws a hollow rectangle, whereas the fillRect() method draws a filled (solid) rectangle.

2. The four arguments for the drawRect() method are the X,Y coordinates of the rectangle’s upper left corner and the width and height of the rectangle.
public void paint( Graphics g )
{
    //set color to red
    g.setColor ( Color.red)
    // draw a rectangle
    g.drawRect(120,100,200,120);
}

Self-test 3.4
1 The layout managers are FlowLayout, GridLayout, BorderLayout.

2 The default layout manager is BorderLayout.

3 The BorderLayout manager places the components you create for the application/applet into each cell of the grid, working from left to right and top to bottom.

4 The GridLayout constructor’s four arguments are the number of columns and rows in the grid, and the horizontal and vertical spacing of the cells in the grid.

5 The component positions you can use with the BorderLayout manager are North, South, East, West, and Center.

6 To add a component to an applet using the BorderLayout manager, you call a special version of the add() method that has the position string (North, South, etc.) and a reference to the component as arguments.

Self-test 3.5
1 // Constructs an empty Label
public JLabel()
    // Constructs a Label that displays the text s with
    // default left-justified alignment
public JLabel(String s)
    // Constructs a Label that displays the text s with
    // the specified alignment. Possible alignments are
    // Label.LEFT, Label.CENTER, and Label.Right
public JLabel(String s, int alignment)

2 The two arguments required by JLabel are the text for the label and the alignment.

3 Since label objects do not generate events, nothing would happen.

Self-test 3.6
The two arguments are the default text and the width in characters.
Self-test 3.7

The argument is the text label for the button.

Self-test 3.8

To determine which checkbox generated an event, you can use the `getSource()` method to return the checkbox’s name.

Self-test 3.9

You can determine the coordinates from the Event object’s `getX` and `getY` methods.
west = new JButton("West");
c.add (west, BorderLayout.WEST );
west.addActionListener ( handler );

panel = new CustomPanel();
c.add (panel, BorderLayout.CENTER);
setSize ( 400, 400 );
show ();
}

// inner class for customed Panels
private class CustomPanel extends JPanel {
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        g.setColor ( new Color( 0, 0, 255 ));
        g.fillArc( x , y, 80, 80, 0, 360 );
    }
}

// inner class for button event handling
private class ButtonHandler implements ActionListener
{
    public void actionPerformed ( ActionEvent e )
    {
        JButton name = (JButton)e.getSource();

        if ( name == north )
            y--;
        if ( name == south )
            y++;
        if ( name == east )
            x++;
        if ( name == west )
            x--;
        panel.repaint();
    }
}

public static void main ( String args[] )
{
    BorderLayoutAct app =new BorderLayoutAct();
    app.addWindowListener ( new WindowAdapter() {
        public void windowClosing ( WindowEvent e )
        {
            System.exit(0);
        }
    });
}
}
Activity 3.3

```java
// Only import coding are shown
public class ButtonA extends JFrame implements ActionListener {
    static final String DISABLE = "Disable";
    static final String ENABLE = "enable";

    // initialize initial status
    b1 = new Button("Disable Button A");
    b2 = new Button("Button A");
    b3 = new Button("Enable Button");
    b3.setEnable ( false );
    b3.setActionCommand (ENABLE);
    b1.addActionListener(handler);
    b2.addActionListener(handler);

    // other coding are skipped
    public void actionPerformed (ActionEvent e ){
        String command = e.getActionCommand();
        if ( command == DISABLE ) {
            b2.setEnable(false);
            b1.setEnable(false);
            b3.setEnable(true);
        } else {    b2.setEnable(true);
            b1.setEnable(true);
            b3.setEnable(false);
        }
    }
}
```
Activity 3.4

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

public class MultipleSelection extends JFrame {
    private JList colorList, copyList;
    private JButton copy;
    private String colorNames[] = {
        "Black", "Blue", "Cyan", "Dark Gray", "Gray",
        "Green", "Light Gray", "Magenta", "Orange",
        "Pink", "Red", "White", "Yellow" }

    public MultipleSelection() {
        super( "Multiple Selection Lists" );
        Container c = getContentPane();
        c.setLayout( new FlowLayout() );

        colorList = new JList( colorNames );
        colorList.setVisibleRowCount( 5 );
        colorList.setFixedCellHeight( 15 );
        colorList.setSelectionMode( ListSelectionModel.SINGLE_INTERVAL_SELECTION );
        c.add( new JScrollPane( colorList ) );

        // create copy button
        copy = new JButton( "Copy >>>" );
        copy.addActionListener( new ActionListener() {
            public void actionPerformed( ActionEvent e ) {
                // place selected values in copyList
                copyList.setListData( colorList.getSelectedValues() );
            }
        });
        c.add( copy );

        copyList = new JList( );
        copyList.setVisibleRowCount( 5 );
        copyList.setFixedCellWidth( 100 );
        copyList.setFixedCellHeight( 15 );
        copyList.setSelectionMode( ListSelectionModel.SINGLE_INTERVAL_SELECTION );
        c.add( new JScrollPane( copyList ) );

        setSize( 300, 120 );
        show();
    }
}
```