***// Fig. 2.8: Addition.java An addition program***

```
import javax.swing.JOptionPane;

public class Addition {
   public static void main( String args[] )
   {
      String firstNumber,   secondNumber;
      int number1, number2, sum;

      // read in first number from user as a string
      firstNumber =  JOptionPane.showInputDialog( "Enter first integer" );

      // read in second number from user as a string
      secondNumber =   JOptionPane.showInputDialog( "Enter second integer" );

      // convert numbers from type String to type int
      number1 = Integer.parseInt( firstNumber );
      number2 = Integer.parseInt( secondNumber );

      sum = number1 + number2;

      // display the results
      JOptionPane.showMessageDialog( null, "The sum is " + sum, "Results", JOptionPane.PLAIN_MESSAGE );

      System.exit( 0 );                        // terminate the program
   }
}
```

**// Fig. 3.10: WelcomeLines.java Displaying text and lines**

```
import javax.swing.JApplet;
import java.awt.Graphics;

public class WelcomeLines extends JApplet {
   public void paint( Graphics g )
   {
      g.drawLine( 15, 10, 210, 10 );
      g.drawLine( 15, 30, 210, 30 );
      g.drawString( "Welcome to Java Programming!", 25, 25 );
   }
}
```

The class **JOptionPane** (javax.swing) allows you to display a dialog box containing info.

- showMessageDialog(), showInputDialog()

**Graphics** object:

- drawString(), drawLine(), drawOval(), drawRect()

**/ Fig. 3.12: AdditionApplet.java   Adding two floating-point numbers**

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

public class AdditionApplet extends JApplet {
  double sum;

  public void init()
  {
    String firstNumber,   secondNumber;
    double number1,  number2;

    // read in first number from user
    firstNumber =  JOptionPane.showInputDialog( "Enter first floating-point value" );

    // read in second number from user
    secondNumber = JOptionPane.showInputDialog( "Enter second floating-point value" );

    // convert numbers from type String to type double
    number1 = Double.parseDouble( firstNumber );
    number2 = Double.parseDouble( secondNumber );

     sum = number1 + number2;
  }


  public void paint( Graphics g )
  {
    // draw the results with g.drawString
    g.drawRect( 15, 10, 270, 20 );
    g.drawString( "The sum is " + sum, 25, 25 );
  }
}
```

```
// Fig. 5.6: Interest.java     Calculating compound interest

import java.text.DecimalFormat;
import javax.swing.JOptionPane;
import javax.swing.JTextArea;

public class Interest {
   public static void main( String args[] )
   {
      double amount, principal = 1000.0, rate = .05;

      DecimalFormat precisionTwo = new DecimalFormat( "0.00" );
      JTextArea outputTextArea = new JTextArea( 11, 20 );

      outputTextArea.append( "Year\tAmount on deposit\n" );

      for ( int year = 1; year <= 10; year++ ) {
         amount = principal * Math.pow( 1.0 + rate, year );
         outputTextArea.append( year + "\t" +
            precisionTwo.format( amount ) + "\n" );
      }

      JOptionPane.showMessageDialog(
         null, outputTextArea, "Compound Interest",
         JOptionPane.INFORMATION_MESSAGE );

      System.exit( 0 );                        // terminate the application
   }
}
```

**JTextArea** (javax.swing) is a GUI component that is capable of displaying many lines of text. The arguments determine the number of rows and columns as well as the size on the screen.

    append(),   setText()

**<u>Fig. 5.6</u>:** An interesting feature of class JOptionPane is that the message it displays with showMessageDialog can be a String or a GUI component such as a JtextArea.

```
JTextArea outTextArea = new JTA(11,20);
. . .
JOptionPane.showMessageDialog(null,
        outTextArea, “CompoundInterest”,
        JOptionPane.INFO_MSG);
```

**// Fig. 5.7: SwitchTest.java      Drawing shapes**

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


public class SwitchTest extends JApplet {
   int choice;

   public void init()
   {
      String input;

      input = JOptionPane.showInputDialog(
                 "Enter 1 to draw lines\n" +
                 "Enter 2 to draw rectangles\n" +
                 "Enter 3 to draw ovals\n" );

      choice = Integer.parseInt( input );
   }



   public void paint( Graphics g )
   {
      for ( int i = 0; i < 10; i++ ) {
         switch( choice ) {
            case 1:
               g.drawLine( 10, 10, 250, 10 + i * 10 );
               break;
            case 2:
               g.drawRect( 10 + i * 10, 10 + i * 10, 50 + i * 10, 50 + i * 10 );
               break;
            case 3:
               g.drawOval( 10 + i * 10, 10 + i * 10, 50 + i * 10, 50 + i * 10 );
               break;
            default:
               JOptionPane.showMessageDialog( null, "Invalid value entered" );

         }    // end switch
      }       // end for
   }         // end paint()
}            // end class SwitchTest
```

**/ Fig. 5.19: LogicalOperators.java      Demonstrating the logical operators**

```
import javax.swing.*;

public class LogicalOperators {
  public static void main( String args[] )
  {
    JTextArea outputArea = new JTextArea( 17, 20 );
    JScrollPane scroller = new JScrollPane( outputArea );
    String output = "";

    output += "Logical AND (&&)" +  "\nfalse && false: " + ( false && false ) +  "\nfalse && true: "
          + ( false && true ) +  "\ntrue && false: " + ( true && false ) +  "\ntrue && true: " + ( true && true );

    output += "\n\nLogical OR (||)" +
           "\nfalse || false: " + ( false || false ) +
           "\nfalse || true: " + ( false || true ) +
           "\ntrue || false: " + ( true || false ) +
           "\ntrue || true: " + ( true || true );

    output += "\n\nBoolean logical AND (&)" +
           "\nfalse & false: " + ( false & false ) +
           "\nfalse & true: " + ( false & true ) +
           "\ntrue & false: " + ( true & false ) +
           "\ntrue & true: " + ( true & true );

    output += "\n\nBoolean logical inclusive OR (|)" +
           "\nfalse | false: " + ( false | false ) +
           "\nfalse | true: " + ( false | true ) +
           "\ntrue | false: " + ( true | false ) +
           "\ntrue | true: " + ( true | true );

    output += "\n\nBoolean logical exclusive OR (^)" +
           "\nfalse ^ false: " + ( false ^ false ) +
           "\nfalse ^ true: " + ( false ^ true ) +
           "\ntrue ^ false: " + ( true ^ false ) +
           "\ntrue ^ true: " + ( true ^ true );

    output += "\n\nLogical NOT (!)" +
           "\n!false: " + ( !false ) +
           "\n!true: " + ( !true );

    outputArea.setText( output );

    JOptionPane.showMessageDialog( null, scroller,
      "Truth Tables", JOptionPane.INFORMATION_MESSAGE );

    System.exit( 0 );
  }
}
```

**Fig 5-19**: Class JScrollPane (javax.swing) provides a GUI component with scrolling functionality.  It is initialized with the GUI component for which it will provide scrolling.

```
JScrollPane scroller = new JSP( outputArea);
```

**/ Fig. 6.3: SquareInt.java    A programmer-defined square method**

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

public class SquareInt extends JApplet {
  public void init()
  {
    String output = "";

    JTextArea outputArea = new JTextArea( 10, 20 );

    // get the applet's GUI component display area
    Container c = getContentPane();

    // attach outputArea to Container c
    c.add( outputArea );

    int result;

    for ( int x = 1; x <= 10; x++ ) {
      result = square( x );
      output += "The square of " + x + " is " + result + "\n";
    }

    outputArea.setText( output );
  }

  // square method definition
  public int square( int y )
  {
    return y * y;
  }
}
```

**Fig 6.3:** Displays a GUI component on an applet. The on-screen display area for a Japplet has a *content pane* to which the GUI components must be attached so they can be displayed at execution time.

The content pane is an object of class **Container** (java.awt).

```
Container c = getContentPane();
c.add(outputArea);
```

Method **getContentPane()** returns a reference to the applet's content pane. The **add()** method places the JTextArea object on the applet so it can be displayed when executed.

**Fig 6.9**: Container and FlowLayout are imported from java.awt. The java.awt.event contains many data types that enable a program to process a user's interactions with a program (ActionListener, ActionEvent). JApplet, JLabel, JLextField and JButton are imported from javax.swing.

A **JLabel** contains a string of chars to display on the screen. It normally indicates the purpose of another gui element on the screen.

**JTextField**s are used to get a single line of info from the user at the keyboard or to display info on the screen.

- setEditable(): with argument **false** indicates that it is not editable (gray background).

Method **init()** creates the gui components and attaches them to the user interface.

```
c.setLayout (new FlowLayout() );
```

This method defines the *layout manager* for the applet's user interface. Layout managers are provided to arrange components on a **Container** for presentation purposes. They determine the position and size of every component attached.

**FlowLayout** is the most basic manager. GUI components are placed from left to right in the order in which they are attached to the Container with the method add().

Each Container can have only one layout manager at a time. Separate Containers in the same program can have different managers.

The interface **ActionListener** specifies that the class must define the method:

```
public void actionPerformed(ActionEvent e)
```

This method is used to process a user's interaction with the JButton. When the user presses the button, this method will be called automatically in response to the user interaction (*event*). This process is *called event handling.* The handler is the actionPerfomed() method.

```
roll.addActionListener( this );
```

The above method registers the applet (this) as the listener for events from the JButton roll.

When a GUI event occurs in a program, Java creates an object containing info about the event that occurred and automatically calls an appropriate event handling method.

Before any event can be processed, each GUI component must know which object in the program defines the event handling method that will be called when the event occurs.

To respond to an action event, we must define a class that implements ActionListener (that requires that the class also define method actionPerformed() ) and we must register the event handler.

actionPerformed() receives one argument – an **ActionEvent** – which contains info about the action that has occurred.

```
// Fig. 6.9: Craps.java      Craps

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

public class Craps extends JApplet implements ActionListener {

  final int WON = 0, LOST = 1, CONTINUE = 2;       // const varis for status of game

  boolean firstRoll = true;               // true if first roll
  int sumOfDice = 0;                      // sum of the dice
  int myPoint = 0;                        // point if no win/loss on first roll
  int gameStatus = CONTINUE;     // game not over yet

  JLabel die1Label, die2Label, sumLabel, pointLabel;
  JTextField firstDie, secondDie, sum, point;
  JButton roll;

  public void init()  {                               // setup graphical user interface components
    Container c = getContentPane();
    c.setLayout( new FlowLayout() );

    die1Label = new JLabel( "Die 1" );
    c.add( die1Label );
    firstDie = new JTextField( 10 );
    firstDie.setEditable( false );
    c.add( firstDie );

    die2Label = new JLabel( "Die 2" );
    c.add( die2Label );
    secondDie = new JTextField( 10 );
    secondDie.setEditable( false );
    c.add( secondDie );

    sumLabel = new JLabel( "Sum is" );
    c.add( sumLabel );
    sum = new JTextField( 10 );
    sum.setEditable( false );
    c.add( sum );

    pointLabel = new JLabel( "Point is" );
    c.add( pointLabel );
    point = new JTextField( 10 );
    point.setEditable( false );
    c.add( point );

    roll = new JButton( "Roll Dice" );
    roll.addActionListener( this );
    c.add( roll );
  }


 public void actionPerformed( ActionEvent e )  {        // call method play when button is pressed
    play();
  }
  public void play()   {                         // process one roll of the dice
    if ( firstRoll ) {                           // first roll of the dice
      sumOfDice = rollDice();

      switch ( sumOfDice ) {
        case 7: case 11:                         // win on first roll
          gameStatus = WON;
          point.setText( "" );                   // clear point text field
          break;
        case 2: case 3: case 12:                 // lose on first roll
          gameStatus = LOST;
          point.setText( "" );                    // clear point text field
          break;
        default:                                  // remember point
          gameStatus = CONTINUE;
          myPoint = sumOfDice;
          point.setText( Integer.toString( myPoint ) );
          firstRoll = false;
          break;
      }
    }
    else {
      sumOfDice = rollDice();
          if ( sumOfDice == myPoint )            // win by making point
        gameStatus = WON;
      else
        if ( sumOfDice == 7 )                    // lose by rolling 7
          gameStatus = LOST;
    }

    if ( gameStatus == CONTINUE )
      showStatus( "Roll again." );
    else {
      if ( gameStatus == WON )
        showStatus( "Player wins. " +  "Click Roll Dice to play again." );
      else
        showStatus( "Player loses. " +  "Click Roll Dice to play again." );

      firstRoll = true;
    }
  }

  public int rollDice()  {                                 // roll the dice
    int die1, die2, workSum;
    die1 = 1 + ( int ) ( Math.random() * 6 );
    die2 = 1 + ( int ) ( Math.random() * 6 );
    workSum = die1 + die2;
    firstDie.setText( Integer.toString( die1 ) );
    secondDie.setText( Integer.toString( die2 ) );
    sum.setText( Integer.toString( workSum ) );
    return workSum;
  }
}
```

**// Fig. 6.13: FibonacciTest.java        Recursive fibonacci method**

```
public class FibonacciTest extends Japplet   implements ActionListener {

  JLabel numLabel, resultLabel;
  JTextField num, result;

  public void init()
  {
    Container c = getContentPane();
    c.setLayout( new FlowLayout() );

    numLabel =  new JLabel( "Enter an integer and press Enter" );
    c.add( numLabel );

    num = new JTextField( 10 );
    num.addActionListener( this );
    c.add( num );

    resultLabel = new JLabel( "Fibonacci value is" );
    c.add( resultLabel );

    result = new JTextField( 15 );
    result.setEditable( false );
    c.add( result );
  }


  public void actionPerformed( ActionEvent e )
  {
    long number, fibonacciValue;

    number = Long.parseLong( num.getText() );
    showStatus( "Calculating ..." );
    fibonacciValue = fibonacci( number );
    showStatus( "Done." );
    result.setText( Long.toString( fibonacciValue ) );
  }


  public long fibonacci( long n )                          // Recursive definition of method fibonacci
  {
    if ( n == 0 || n == 1 )  // base case
      return n;
    else
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
  }
}
```

**Fig 6-13**: In this example, the user presses the *Enter* key while typing in the **JTextField num** to generate the action event.

Automatically, a message is sent to the applet indication that the user interacted with one of the program's GUI components.

The message **e.getActionCommand(),** when executed inside the handler, returns the string the user typed in the JTextField before pressing the *Enter* key.

```
// Fig. 8.5: TimeTest.java   Demonstrating the Time3 class set and get methods

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.deitel.jhtp3.ch08.Time3;

public class TimeTest extends Japplet    implements ActionListener {

  private Time3 t;
  private JLabel hourLabel, minuteLabel, secondLabel;
  private JTextField hourField, minuteField,  secondField, display;
  private JButton tickButton;

  public void init()  {
    t = new Time3();

    Container c = getContentPane();

    c.setLayout( new FlowLayout() );
    hourLabel = new JLabel( "Set Hour" );
    hourField = new JTextField( 10 );
    hourField.addActionListener( this );
    c.add( hourLabel );
    c.add( hourField );

    minuteLabel = new JLabel( "Set minute" );
    minuteField = new JTextField( 10 );
    minuteField.addActionListener( this );
    c.add( minuteLabel );
    c.add( minuteField );

    secondLabel = new JLabel( "Set Second" );
    secondField = new JTextField( 10 );
    secondField.addActionListener( this );
    c.add( secondLabel );
    c.add( secondField );

    display = new JTextField( 30 );
    display.setEditable( false );
    c.add( display );

    tickButton = new JButton( "Add 1 to Second" );
    tickButton.addActionListener( this );
    c.add( tickButton );

    updateDisplay();
  }
public void actionPerformed( ActionEvent e )
  {
    if ( e.getSource() == tickButton )
      tick();
    else if ( e.getSource() == hourField ) {
      t.setHour( Integer.parseInt( e.getActionCommand() ) );
      hourField.setText( "" );
    }
    else if ( e.getSource() == minuteField ) {
      t.setMinute( Integer.parseInt( e.getActionCommand() ) );
      minuteField.setText( "" );
    }
    else if ( e.getSource() == secondField ) {
      t.setSecond(  Integer.parseInt( e.getActionCommand() ) );
      secondField.setText( "" );
    }

    updateDisplay();
  }



 public void updateDisplay()
  {
    display.setText( "Hour: " + t.getHour() +
          "; Minute: " + t.getMinute() +
          "; Second: " + t.getSecond() );
    showStatus( "Standard time is: " + t.toString() +
          "; Universal time is: " + t.toUniversalString() );
  }



 public void tick()
  {
    t.setSecond( ( t.getSecond() + 1 ) % 60 );

    if ( t.getSecond() == 0 ) {
      t.setMinute( ( t.getMinute() + 1 ) % 60 );

      if ( t.getMinute() == 0 )
        t.setHour( ( t.getHour() + 1 ) % 24 );
    }
  }
}
```

**Fig 8-5:** Several GUI components that are being handled with the same method. The handler uses **e.getSource()** to determine which component generated the event. The ActionEvent parameter that is supplied to the handler contains a refernce to the source.

**/ Fig. 9.12: Time.java    Time class definition**

```
import java.text.DecimalFormat;     // used for number formatting

public class Time extends Object {
  private int hour;
  private int minute;
  private int second;

  public Time() { setTime( 0, 0, 0 ); }

  public void setTime( int h, int m, int s )  {
    setHour( h );   setMinute( m);  setSecond( s );
   }

  public void setHour( int h )
    { hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); }

  public void setMinute( int m )
    { minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); }

  public void setSecond( int s )
    { second = ( ( s >= 0 && s < 60 ) ? s : 0 ); }

  public int getHour() { return hour; }

  public int getMinute() { return minute; }

  public int getSecond() { return second; }


  public String toString()
  {
    DecimalFormat twoDigits = new DecimalFormat( "00" );

    return ( ( getHour() == 12 || getHour() == 0 ) ?
          12 : getHour() % 12 ) + ":" +
         twoDigits.format( getMinute() ) + ":" +
         twoDigits.format( getSecond() ) +
         ( getHour() < 12 ? " AM" : " PM" );
  }
}


// Fig. 9.12: TimeTestWindow.java    Demonstrating the Time class set and get methods

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

public class TimeTestWindow extends JFrame {
  private Time t;
  private JLabel hourLabel, minuteLabel, secondLabel;
  private JTextField hourField, minuteField, secondField, display;
  private JButton exitButton;
```

```
   public TimeTestWindow()   {
      super( "Inner Class Demonstration" );

      t = new Time();

      Container c = getContentPane();

      // create an instance of the inner class
      ActionEventHandler handler = new ActionEventHandler();

      c.setLayout( new FlowLayout() );
      hourLabel = new JLabel( "Set Hour" );
      hourField = new JTextField( 10 );
      hourField.addActionListener( handler );
      c.add( hourLabel );
      c.add( hourField );

      minuteLabel = new JLabel( "Set minute" );
      minuteField = new JTextField( 10 );
      minuteField.addActionListener( handler );
      c.add( minuteLabel );
      c.add( minuteField );

      secondLabel = new JLabel( "Set Second" );
      secondField = new JTextField( 10 );
      secondField.addActionListener( handler );
      c.add( secondLabel );
      c.add( secondField );

      display = new JTextField( 30 );
      display.setEditable( false );
      c.add( display );

      exitButton = new JButton( "Exit" );
      exitButton.addActionListener( handler );
      c.add( exitButton );
   }


public void displayTime() {
      display.setText( "The time is: " + t );
   }


 public static void main( String args[] )   {
      TimeTestWindow window = new TimeTestWindow();

      window.setSize( 400, 140 );
      window.show();
   }
```

```
// INNER CLASS DEFINITION FOR EVENT HANDLING

private class ActionEventHandler implements ActionListener {

    public void actionPerformed( ActionEvent e )
     {
       if ( e.getSource() == exitButton )
         System.exit( 0 );                                    // terminate the application
       else if ( e.getSource() == hourField ) {
         t.setHour( Integer.parseInt( e.getActionCommand() ) );
         hourField.setText( "" );
       }
       else if ( e.getSource() == minuteField ) {
         t.setMinute(Integer.parseInt( e.getActionCommand() ) );
         minuteField.setText( "" );
       }
       else if ( e.getSource() == secondField ) {
         t.setSecond( Integer.parseInt( e.getActionCommand() ) );
         secondField.setText( "" );
       }

       displayTime();
     }
   }
}
```

**<u>Fig 9-12</u>:** TimeTestWindow extends **JFrame** (javax.swing) rather than JApplet. This class provides the basic attributes and behavior of a window – a *title b*ar and buttons to *minimize, maximize* and *close* the window.

The init() method of the applet has been replaced by a constructor so that the GUI components are created as the application begins executing.

```
super ("Inner Class Demonstration")
```

The above calls the superclass constructors with a string that will be displayed in the title bar.

```
ActionEventHandler handler = new AEH()
```

Defines one instance of an *inner class* and assigns it to **handler**. This reference is passed to each of the calls to addActionListener() to register it as the event handler.

Each call to this method (addAL() ) requires an object of ActionListener to be passed as an argument. **handler** is an ActionListener since its class implements the interface:

```
private class ActionEventHandler implements ActionListener {
```

The inner class is defined as *private* because its only used in this class definition. An inner class object has a special relationship with the outer class object creates it. The inner class object is allowed to access directly all the instance variables and methods of the outer class object.

**// Fig. 9.13: TimeTestWindow.java   Demonstrating the Time class set and get methods**

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


public class TimeTestWindow extends JFrame {

   private Time t;
   private JLabel hourLabel, minuteLabel, secondLabel;
   private JTextField hourField, minuteField, secondField, display;



public TimeTestWindow()
   {
      super( "Inner Class Demonstration" );

      t = new Time();

      Container c = getContentPane();

      c.setLayout( new FlowLayout() );
      hourLabel = new JLabel( "Set Hour" );
      hourField = new JTextField( 10 );

      hourField.addActionListener(
         new ActionListener() {                                   // anonymous inner class
            public void actionPerformed( ActionEvent e )
            {
               t.setHour(
                  Integer.parseInt( e.getActionCommand() ) );
               hourField.setText( "" );
               displayTime();
            }
         }
      );
      c.add( hourLabel );
      c.add( hourField );

      minuteLabel = new JLabel( "Set minute" );
      minuteField = new JTextField( 10 );

      minuteField.addActionListener(
         new ActionListener() {                                   // anonymous inner class
            public void actionPerformed( ActionEvent e )
            {
               t.setMinute(
                  Integer.parseInt( e.getActionCommand() ) );
               minuteField.setText( "" );
               displayTime();
            }
         }
      );
```

```
    c.add( minuteLabel );
    c.add( minuteField );

    secondLabel = new JLabel( "Set Second" );
    secondField = new JTextField( 10 );

    secondField.addActionListener(
      new ActionListener() {                                // anonymous inner class
        public void actionPerformed( ActionEvent e )
        {
          t.setSecond(
            Integer.parseInt( e.getActionCommand() ) );
          secondField.setText( "" );
          displayTime();
        }
      }
    );
    c.add( secondLabel );
    c.add( secondField );

    display = new JTextField( 30 );
    display.setEditable( false );
    c.add( display );
  }



public void displayTime()
  {
    display.setText( "The time is: " + t );
  }



public static void main( String args[] )
  {
    TimeTestWindow window = new TimeTestWindow();

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

    window.setSize( 400, 120 );
    window.show();
  }
}
```

**<u>Fig 9-13</u>:** Modifies the previous example to use an *anonymous inner class*. Since it has no name, one object of the anonymous class must be created at the point where the class is defined in the program.

Each of the JTextFields that generate events have a similar anonymous inner class to handle events.

```
hourField.addActionListener(
    new ActionListener() {                    // aic
        public void actionPerformed(ActionEvent e) {
        t.setHour( Integer.parseInt( e.getActionCommand() ) );
        hourField.setText( " " );
        displayTime();
     }
        }
    );
```

The argument to addActionListener(), that must be an object that is an ActionListener, is an object of an anonymous class created by the line **new ActionListener ()** (the parenthesis indicate a call to the default constructor of the aic).

The body of the class (between { } ) defines one method to be called when the user presses *Enter* while typing in **hourField**.

Windows generate a variety of events. This example deals with the event generated when the user clicks the window's close box – a *window closing event*.

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

The method addWindowListener() registers the window event listener. The argument must be a reference to an object the is a **WindowListener** (java.awt.event).

However, there are seven different methods that must be defined in every class that implements the interface, and we only need one for the window closing event.

For event handling interfaces with more than one method, Java provides a corresponding class (*adapter class*) that already implements all the methods in the interface for you. All you need to do is extend the adapter class and override the methods you require.

The syntax **WindowAdapter( )** begins the definition of an anonymous inner class that extends class WindowAdapter. This is similar to beginning a class definition with:

```
    public class MyHandler extends WindowAdapter {
```

Class WA implements the interface WListener, so every object of WA is a WL.

**<u>Notes on Inner Classes</u>**

1. Compiling a class that contains inner classes results in a separate **.class** file for every class. Inner classes have the file name *OuterClassName*$*InnerClassName***.class**

   Anonymous inner classes have the file name *OuterClassName***$***#InnerClassName***.class** where # starts at 1 and is incremented for each AIC encountered during compilation.

2. Inner classes with class names can be defined *public*, *private*, package access, or *protected* and are subject to the same usage restrictions as other members of a class.

3. To access the outer class's **this** reference, use *OuterClassName***.this**

4. The outer class is responsible for creating objs of its inner classes. To create an obj of another class's inner class, first create obj of the outer class and assign it to a refernce (lets call it **ref**). Then use a statement of the following form to create an inner class obj:

   *OuterClassName.InnerClassName* **innerRef** = **ref.new** *InnerClassName* ( ) ;

5. An inner class can be declared **static**. These do not require an obj of its outer class to be defined. A static IC does not have access to the outer class's non-static members.

## Swing Overview

Swing components are written, manipulated and displayed completely in Java.

The original components from the AWT are tied directly to the local's platform's graphical user interface capabilities. So a program executing on different platforms has a different appearance and sometimes even different user interactions.

The Swing components allow the programmer to specify a different look and feel for each platform, or a uniform one across all, or even change the look and feel while running.

Hierarchy:

```
java.lang.Object
    java.awt.Component
        ava.awt.Container
            javax.swing.JComponent
```

Swing components that subclass JComponent have the following features:

1. A *pluggable look and feel* that can be used to customize the l-a-f when the program executes on different platforms.
2. Shortcut keys (called *mnemonics*) for direct access to components through the keyboard.
3. Common event handling capabilities for cases where several components initiate the same actions in a program.
4. Brief descriptions of a component's purpose (called *tool tips*) that are displayed when the mouse cursor is positioned over the compt.
5. Support for assistive technologies.
6. Support for user interface *localization*.

**Fig 12-4:** Example using labels with and without icons and different alignments. Many swing components can display images by specifying an **Icon** as an argument to their constructor or by using a method that is normally called **setIcon()**.

```
Icon bug = new ImageIcon("bug.gif");
label2  = new JLabel ("Text and icon", bug, SwingConstants.LEFT).
```

The horizontal and vertical positiojns can be also be set:

```
label3.setHorizontalPosition(SwingConstants.CENTER)
```

```
// Fig. 12.4: LabelTest.java.    Demonstrating the JLabel class.

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

public class LabelTest extends JFrame {

  private JLabel label1, label2, label3;

  public LabelTest()
  {
    super( "Testing JLabel" );

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


    label1 = new JLabel( "Label with text" );             // JLabel constructor with a string argument
    label1.setToolTipText( "This is label1" );
    c.add( label1 );


    // JLabel constructor with string, Icon and alignment arguments

    Icon bug = new ImageIcon( "bug1.gif" );
    label2 = new JLabel( "Label with text and icon",  bug, SwingConstants.LEFT );
    label2.setToolTipText( "This is label2" );
    c.add( label2 );


    label3 = new JLabel();                                             // JLabel constructor no arguments
    label3.setText( "Label with icon and text at bottom" );
    label3.setIcon( bug );
    label3.setHorizontalTextPosition( SwingConstants.CENTER );
    label3.setVerticalTextPosition(SwingConstants.BOTTOM );
    label3.setToolTipText( "This is label3" );
    c.add( label3 );

    setSize( 275, 170 );
    show();
  }


  public static void main( String args[] )
 {
    LabelTest app = new LabelTest();

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

**// Fig. 12.7: TextFieldTest.java** ***Demonstrating the JTextField class.***

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

public class TextFieldTest extends JFrame {
  private JTextField text1, text2, text3;
  private JPasswordField password;

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

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

    text2 = new JTextField( "Enter text here" );           // construct textfield with default text
    c.add( text2 );

    text3 = new JTextField( "Uneditable text field", 20 ); // construct textfield with default text and 20
    text3.setEditable( false );                            //  visible elements and no event handler
    c.add( text3 );

    password = new JPasswordField( "Hidden text" );        // construct textfield with default text
    c.add( password );

    TextFieldHandler handler = new TextFieldHandler();
    text1.addActionListener( handler );
    text2.addActionListener( handler );   text3.addActionListener( handler );
    password.addActionListener( handler );
    setSize( 325, 100 );
    show();
  }

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

  // inner class for event handling
  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();
      else if ( e.getSource() == text3 )     s = "text3: " + e.getActionCommand();
      else if ( e.getSource() == password ) {
        JPasswordField pwd = (JPasswordField) e.getSource();
        s = "password: " + new String( pwd.getPassword() );
      }
      JOptionPane.showMessageDialog( null, s );
    }  }   }
```

**Fig 12-7:** When the user presses Enter in the active text field a message dialog box containing the text in the field is displayed. When an event occurs in the password field, the password is revealed. A **JPasswordField** shows that a char was typed as the user enters chars, but hides the characters.

The inner class event handler determines the component using getSource(), and gets the text entered with getActionCommand(). For the password field it must first cast the component to be able to use the getPassword() method.

```
JPasswordField = pwd (JPasswordField) e.getSource( );
```

**Fig 12-10:** The icon on the button changes as the mouse moves in and out of the button's area on the screen. A **JButton** can display Icons and can also have a *rollover* Icon – one that is displayed when the mouse is positioned over the button.

```
fancyButton.setRolloverIcon(bug2);
```

Buttons generate ActionEvents that can be processed by any ActionListener object with the method actionPerformed(). The inner class handler displays a message dialog box containing the label for the button that was pressed by the user. The ActionEvent method e.getActionCommand() returns the label.

**Fig 12-11**: Changes the style of the text field by selecting check buttons. One **JCheckButton** applies a bold style and the other italic. When the user clicks a JCheckButton an **ItemEvent** is generated that can be handled by an **ItemListener**.

The ItemListener obj must define the method **itemStateChanged( ).** This example uses an inner class as handler that determines the button checked with getSource() and the state of the button as follows:

```
if ( e.getStateChange() == ItemEvent.SELECTED )
      valBold = Font.BOLD;
   else
      valBold = Font.PLAIN;
```

**Fig 12-12:** Uses JRadioButtons that permit only a single font style to be selected. The constructor of the radio button specifies the label and the initial state.

```
radioGroup = new ButtonGroup();
```

The above object is the "glue" that binds the four radio buttons together to form a logical relationship that allows only one of the buttons to be selected at a time. Class RadioButtonHandler implements the ItemListener interface so it can handle events generated by the radio buttons.

**// Fig. 12.10: ButtonTest.java** ***Creating JButtons.***

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

public class ButtonTest extends JFrame {
  private JButton plainButton, fancyButton;

  public ButtonTest()
  {
    super( "Testing Buttons" );

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


    plainButton = new JButton( "Plain Button" );                    // create buttons
    c.add( plainButton );

    Icon bug1 = new ImageIcon( "bug1.gif" );
    Icon bug2 = new ImageIcon( "bug2.gif" );
    fancyButton = new JButton( "Fancy Button", bug1 );
    fancyButton.setRolloverIcon( bug2 );
    c.add( fancyButton );


    ButtonHandler handler = new ButtonHandler();         // create an instance of inner class ButtonHandler
    fancyButton.addActionListener( handler );            // to use for button event handling
    plainButton.addActionListener( handler );

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

  public static void main( String args[] )
  {
    ButtonTest app = new ButtonTest();

    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 )
    {
      JOptionPane.showMessageDialog( null,
        "You pressed: " + e.getActionCommand() );
    }
  }
}
```

**// Fig. 12.11: CheckBoxTest.java** ***Creating Checkbox buttons.***

```
public class CheckBoxTest extends JFrame {
  private JTextField t;
  private JCheckBox bold, italic;

  public CheckBoxTest()  {
    super( "JCheckBox Test" );

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

    t = new JTextField( "Watch the font style change", 20 );
    t.setFont( new Font( "TimesRoman", Font.PLAIN, 14 ) );
    c.add( t );

    bold = new JCheckBox( "Bold" );                    // create checkbox objects
    c.add( bold );
    italic = new JCheckBox( "Italic" );
    c.add( italic );

    CheckBoxHandler handler = new CheckBoxHandler();
    bold.addItemListener( handler );
    italic.addItemListener( handler );
    setSize( 275, 100 );
    show();
  }

  public static void main( String args[] )  {
    CheckBoxTest app = new CheckBoxTest();

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

  private class CheckBoxHandler implements ItemListener {
    private int valBold = Font.PLAIN;
    private int valItalic = Font.PLAIN;

    public void itemStateChanged( ItemEvent e )  {
      if ( e.getSource() == bold )
        if ( e.getStateChange() == ItemEvent.SELECTED )
          valBold = Font.BOLD;
        else
          valBold = Font.PLAIN;

      if ( e.getSource() == italic )
        if ( e.getStateChange() == ItemEvent.SELECTED )
          valItalic = Font.ITALIC;
        else
          valItalic = Font.PLAIN;

      t.setFont( new Font( "TimesRoman", valBold + valItalic, 14 ) );
      t.repaint();
    }   }  }
```

**// Fig. 12.12: RadioButtonTest.java  Creating radio buttons using ButtonGroup and JRadioButton**.

```
public class RadioButtonTest extends JFrame {
  private JTextField t;
  private Font plainFont,  boldFont,  italicFont,  boldItalicFont;
  private JRadioButton plain,  bold,  italic,  boldItalic;
  private ButtonGroup radioGroup;

  public RadioButtonTest()  {
    super( "RadioButton Test" );

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

    t = new JTextField( "Watch the font style change", 25 );
    c.add( t );

    plain = new JRadioButton( "Plain", true );     c.add( plain );                // Create radio buttons
    bold = new JRadioButton( "Bold", false);       c.add( bold );
    italic = new JRadioButton( "Italic", false );  c.add( italic );
    boldItalic = new JRadioButton( "Bold/Italic", false );  c.add( boldItalic );

    RadioButtonHandler handler = new RadioButtonHandler();              // register events
    plain.addItemListener( handler ); bold.addItemListener( handler );
    italic.addItemListener( handler ); boldItalic.addItemListener( handler );

    radioGroup = new ButtonGroup();       // create logical relationship between JRadioButtons
    radioGroup.add( plain );
     radioGroup.add( bold );  radioGroup.add( italic ); radioGroup.add( boldItalic );

    plainFont = new Font( "TimesRoman", Font.PLAIN, 14 );
    boldFont = new Font( "TimesRoman", Font.BOLD, 14 );
    italicFont = new Font( "TimesRoman", Font.ITALIC, 14 );
    boldItalicFont = new Font( "TimesRoman", Font.BOLD + Font.ITALIC, 14 );
    t.setFont( plainFont );

    setSize( 300, 100 );
    show();
  }

  public static void main( String args[] )   {
    RadioButtonTest app = new RadioButtonTest();

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

  private class RadioButtonHandler implements ItemListener {
    public void itemStateChanged( ItemEvent e )   {
      if ( e.getSource() == plain )          t.setFont( plainFont );
      else if ( e.getSource() == bold )      t.setFont( boldFont );
      else if ( e.getSource() == italic )    t.setFont( italicFont );
      else if ( e.getSource() == boldItalic ) t.setFont( boldItalicFont );
       t.repaint();
    }  }  }
```

**// Fig. 12.13: ComboBoxTest.java      Using a JComboBox to select an image to display.**

```
public class ComboBoxTest extends JFrame {

  private JComboBox images;
  private JLabel label;

  private String names[] ={ "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };

  private Icon icons[] =
    { new ImageIcon( names[ 0 ] ),
      new ImageIcon( names[ 1 ] ),
      new ImageIcon( names[ 2 ] ),
      new ImageIcon( names[ 3 ] ) };

  public ComboBoxTest()   {
    super( "Testing JComboBox" );

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

    images = new JComboBox( names );
    images.setMaximumRowCount( 3 );

    images.addItemListener(   new ItemListener() {
        public void itemStateChanged( ItemEvent e )
        {
          label.setIcon( icons[ images.getSelectedIndex() ] );
        }
      }
    );

    c.add( images );

    label = new JLabel( icons[ 0 ] );
    c.add( label );

    setSize( 350, 100 );
    show();
  }


  public static void main( String args[] )
  {
    ComboBoxTest app = new ComboBoxTest();

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

**Fig 12-13:** Uses a JComboBox to provide a list of four image file names, that when selected its displayed as an Icon on a JLabel.

```
images = new JComboBox (names);
```

Creates a combo box obj using the string in array **names** as the elements in the list. A numeric *index* keeps track of the ordering of items.

```
images.setMaximumRowCount(3);
```

Sets the maximum number of elements that are displayed when the user clicks the combo box. If there are more items than the max, the combo box automatically provides a *scrollbar*.

The handler obtains the index of the item selected with method **getSelectedIndex()**.

## Mouse Event Handling

Mouse events are handled by the **MouseListener** and **MouseMotionListener** event-listener interfaces.

Each mouse event handling methods takes a **MouseEvent** obj argument. Contains info about the event that occurred and the **x** and **y** coordinates of the location where it occurred.

**Fig 12-17**: Each mouse event results in a string displayed in a **JLabel** at the bottom of the window. Uses the default layout manager, **BorderLayout**, that divides the content pane's area into five regions - north, south, east, west and center.

```
GetContentPane( ).add( statusBar, BorderLayout.SOUTH );
```

Attaches the JLabel **statusBar** to the south region which extends across the entire bottom of the content pane.

The application window is registered as the listener for its own mouse events. When the mouse enters or exits the application area the corresponding event handlers display a message in **statusBar** indicating the event.

When any of the other five events occur, they display a message describing the event and the coordinates where the event occurred. With the **getX()** and **getY()** MouseEvent methods.

**// Fig. 12.17: MouseTracker.java       Demonstrating mouse events.**

```
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 );                // application listens to its own mouse events
    addMouseMotionListener( this );

    setSize( 275, 100 );
    show();
  }
                                                         // MouseListener event handlers
  public void mouseClicked( MouseEvent e )  {
    statusBar.setText( "Clicked at [" + e.getX() + ", " + e.getY() + "]" );
  }

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

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

  public void mouseMoved( MouseEvent e )  {
    statusBar.setText( "Moved at [" + 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 );
        }
      }
    );
  }
}
```

```
// Fig. 12.19: Painter.java       Using class MouseMotionAdapter.

public class Painter extends JFrame {
  private int xValue = -10, yValue = -10;

  public Painter()
  {
    super( "A simple paint program" );

    getContentPane().add(
      new Label( "Drag the mouse to draw" ),
      BorderLayout.SOUTH );

    addMouseMotionListener(
      new MouseMotionAdapter() {
        public void mouseDragged( MouseEvent e )
        {
          xValue = e.getX();
          yValue = e.getY();
          repaint();
        }
      }
    );
    setSize( 300, 150 );
    show();
  }


 public void paint( Graphics g )  {
    g.fillOval( xValue, yValue, 4, 4 );  }


public static void main( String args[] )  {
    Painter app = new Painter();

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

**Fig 12-19**: Uses the **mouseDragged** event handler to create a simple drawing program. The user can draw pictures with by dragging the mouse on the background of the window.

Since method **mouseMoved** is not used, the **MouseMotionListener** is defined as a subclass of **MouseMotionAdapter** and then we override **mouseDragged**.

The handler captures the **x** and **y** coordinates of the mouse-dragged event and stores them in instance variables **xValue** and **yValue**, then calls **repaint()** to initiate drawing of the next oval on the background.

As you drag the mouse all ovals remain on the window. This is due to a special feature of Swing GUI components called *double buffering* in which all drawing actually occurs in an image stored in memory, then the entire image is displayed on the window. This helps present smooth graphics display in a GUI.

**// Fig. 12.20: MouseDetails.java  Demonstrating mouse clicks and distinguishing between mouse buttons.**

```
public class MouseDetails extends JFrame {
  private String s = "";
  private int xPos, yPos;

  public MouseDetails()  {
    super( "Mouse clicks and buttons" );
    addMouseListener( new MouseClickHandler() );
    setSize( 350, 150 );
    show();
  }

  public void paint( Graphics g ) {
    g.drawString( "Clicked @ [" + xPos + ", " + yPos + "]", xPos, yPos );
  }

  public static void main( String args[] )  {
    MouseDetails app = new MouseDetails();

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

  private class MouseClickHandler extends MouseAdapter {
    public void mouseClicked( MouseEvent e )  {
      xPos = e.getX();
      yPos = e.getY();
      String s = "Clicked " + e.getClickCount() + " time(s)";

      if ( e.isMetaDown() )                          // Right mouse button
        s += " with right mouse button";
      else if ( e.isAltDown() )                      // Middle mouse button
        s += " with center mouse button";
      else                                           // Left mouse button
        s += " with left mouse button";

      setTitle( s );                                 // set the title bar of the window
      repaint();
    }
  }
}
```

## Keyboard Event Handling

Key events are generated when keys are pressed and released. To implement the **KeyListener** interface a class must provide definitions for methods **keyPressed**, **keyReleased** and **keyTyped**. Each of these receive a **KeyEvent** as its argument.

keyPressed: called in response to pressing any key

keyTyped: called in response to pressing any key that is not an *action* key (arrow key, *Home*, *End*, *Page Up*, *Page Down*, function key, *Num Lock*, *Print Screen*, *Scroll Lock*, *Caps Lock* and *Pause*).

keyReleased: called when the key is released after any keyPressed or keyTyped event.

**Fig 12-22:** The constructor registers the application to handle its own key events with method **addKeyListener( ).** This method is defined in class **Component**, so every subclass can notify **KeyListeners** of key events for that Component.

A **JTextArea** is added to display the output. It occupies the entire window. When a single **Component** is added to a **BorderLayout**, it occupies the entire **Container** by default.

Methods keyPressed and keyReleased use KeyEvent method **getKeyCode** to get the *virtual key code* of the key that was pressed. The value returned is passed to KeyEvent method **getKeyText** which returns a string containing the name of the key that was pressed.

Method keyTyped uses KeyEvent method **getKeyChar** to get the Unicode value of the character typed.

All three handlers call method **setLines2and3( )** passing it the event object. This method uses the KeyEvent method **isActionKey** to determine if the key was an action key.

InputEvent mehtod **getModifiers** is called to determine if any modifier keys (*Shift*, *Alt* and *Ctrl*) where pressed. The result is passed to KeyEvent method **getKeyModifiersText**, which produces a string containing the names of the pressed modifier keys.

```
// Fig. 12.22: KeyDemo.java       Demonstrating keystroke events.

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

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

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

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

    getContentPane().add( textArea );

    setSize( 350, 100 );
    show();
  }

  public void keyPressed( KeyEvent e )  {
    line1 = "Key pressed: " +  e.getKeyText( e.getKeyCode() );
    setLines2and3( e );
  }

  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 );
        }
      }
    );
  }
}
```

## Layout Managers

The layout managers provide basic layout capabilities that are easier to use than determining the exact position and size of every GUI component.

**FlowLayout**: the most basic layout manager. Components are placed on a container from left to right in the order in which they are added. When the edge is reached, components aer continued on the next line. Allows components to be *left-aligned*, *centered* (default) and *right-aligned*.

**Fig 12-24:** Creates three buttons and adds them to the application using a FlowLayout. The user can click the **Left**, **Right** or **Center** buttons to change the alignment.

```
        layout.layoutContainer( c );
```

Uses the LayoutManager's interface method to specify that the content pane should be rearranged based on the adjusted layout.

**BorderLayout**: Arranges components into five regions. Up to five components can be added directly - one for each region. If all five regions are occupied, the entire container's space is covered. If a region is not occupied (except center), the other regions expand to fill the remaining space.

**Fig 12-25**: Places five buttons on a BorderLayout. When a button is clicked, it is hidden and the window is adjusted.

```
        layout = new BorderLayout(5, 5);
```

The arguments specify the number of pixels between components. The horizontal gap and the vertical gap (default is 0).

The handler uses **setVisible** to hide the button. The LayoutManager method **layoutContainer** is used to recalculate the layout of the content pane.

**// Fig. 12.24: FlowLayoutDemo.java  Demonstrating FlowLayout alignments.**

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

  public FlowLayoutDemo()   {
    super( "FlowLayout Demo" );
    layout = new FlowLayout();

    c = getContentPane();
    c.setLayout( layout );

    left = new JButton( "Left" );
    left.addActionListener(
      new ActionListener() {
        public void actionPerformed( ActionEvent e )  {
          layout.setAlignment( FlowLayout.LEFT );

          layout.layoutContainer( c );                    // re-align attached components
        }
      }
    );
    c.add( left );
    center = new JButton( "Center" );
    center.addActionListener(
      new ActionListener() {
        public void actionPerformed( ActionEvent e )  {
          layout.setAlignment( FlowLayout.CENTER );

          layout.layoutContainer( c );                    // re-align attached components
        }
      }  );
    c.add( center );

    right = new JButton( "Right" );
    right.addActionListener(
      new ActionListener() {
        public void actionPerformed( ActionEvent e )  {
          layout.setAlignment( FlowLayout.RIGHT );

          layout.layoutContainer( c );                    / re-align attached components
        }
      }  );
    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 );
        }
      }   ); } }
```

**// Fig. 12.25: BorderLayoutDemo.java        Demonstrating BorderLayout.**

```
public class BorderLayoutDemo extends Jframe  implements ActionListener {

   private JButton b[];
   private String names[] = { "Hide North", "Hide South", "Hide East",  "Hide West", "Hide Center" };
   private BorderLayout layout;

   public BorderLayoutDemo()  {
      super( "BorderLayout Demo" );

      layout = new BorderLayout( 5, 5 );

      Container c = getContentPane();
      c.setLayout( layout );

      b = new JButton[ names.length ];                        // instantiate button objects

      for ( int i = 0; i < names.length; i++ ) {
         b[ i ] = new JButton( names[ i ] );
         b[ i ].addActionListener( this );
      }

      // order not important
      c.add( b[ 0 ], BorderLayout.NORTH );          // North position
      c.add( b[ 1 ], BorderLayout.SOUTH );           // South position
      c.add( b[ 2 ], BorderLayout.EAST );             // East position
      c.add( b[ 3 ], BorderLayout.WEST );             // West position
      c.add( b[ 4 ], BorderLayout.CENTER );         // Center position

      setSize( 300, 200 );
      show();
   }

   public void actionPerformed( ActionEvent e )  {
      for ( int i = 0; i < b.length; i++ )
         if ( e.getSource() == b[ i ] )   b[ i ].setVisible( false );
         else   b[ i ].setVisible( true );

      // re-layout the content pane
      layout.layoutContainer( getContentPane() );
   }

   public static void main( String args[] )  {
      BorderLayoutDemo app = new BorderLayoutDemo();

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

**GridLayout**: divides the container into a grid so that components can be placed in rows and columns. Every component has the same width and height. They are added starting at the top left cell of the grid and proceeding left to right.

**Fig 12-26**: Adds six button to a GridLayout. Defines two GridLayout objects which are switched when a button is pressed.

```
grid1 = new GridLayout(2, 3, 5, 5);
grid2 = new GridLayout(3, 2);
```

The object **grid1** is a layout with 2 rows, 3 columns and 5 pixels at the gaps. Object **grid2** specifies 3 rows, 2 columns and no gap space.

Every call to **actionPerformed** toggles the layout between **grid2** and **grid1**. The method **validate()** illustrates another way to re-layout a container for which the layout has changed. It re-computes the container's layout based on the current layout manager and the current set of displayed components.

**Panels**: Complex GUIs require that each component be placed in an exact location. They often consist of multiple *panels* with each panel's components arranged in a specific layout. A **JPanel** is a subclass of **JComponent** that inherits from class **Container**. Thus, JPanels may have components, including other panels, added to them.

**Fig 12-27:** A JPanel is created with a GridLayout. Five buttons are added to the panel which is then added to the south region of a BorderLayout.

```
buttonPanel.setLayout(new GridLayout(1, buttons.lenght) );

c.add( buttonPanel, BorderLayout.SOUTH);
```

The buttons are added directly to the **JPanel** - class JPanel does not have a content pane like an applet or a **JFrame**.

**/ Fig. 12.26: GridLayoutDemo.java        Demonstrating GridLayout.**

```
public class GridLayoutDemo extends Jframe  implements ActionListener {

  private JButton b[];
  private String names[] =  { "one", "two", "three", "four", "five", "six" };
  private boolean toggle = true;
  private Container c;
  private GridLayout grid1, grid2;

  public GridLayoutDemo()  {
    super( "GridLayout Demo" );

    grid1 = new GridLayout( 2, 3, 5, 5 );
    grid2 = new GridLayout( 3, 2 );

    c = getContentPane();
    c.setLayout( grid1 );

    b = new JButton[ names.length ];             // create and add buttons

    for (int i = 0; i < names.length; i++ ) {
      b[ i ] = new JButton( names[ i ] );
      b[ i ].addActionListener( this );
      c.add( b[ i ] );
    }

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

  public void actionPerformed( ActionEvent e )  {
    if ( toggle )  c.setLayout( grid2 );
    else  c.setLayout( grid1 );

    toggle = !toggle;
    c.validate();
  }

  public static void main( String args[] )   {
    GridLayoutDemo app = new GridLayoutDemo();

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

```
// Fig. 12.27: PanelDemo.java   Using a JPanel to help lay out components.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PanelDemo extends JFrame {

   private JPanel buttonPanel;
   private JButton buttons[];


 public PanelDemo()
   {
      super( "Panel Demo" );

      Container c = getContentPane();
      buttonPanel = new JPanel();
      buttons = new JButton[ 5 ];

      buttonPanel.setLayout(  new GridLayout( 1, buttons.length ) );

      for ( int i = 0; i < buttons.length; i++ ) {
         buttons[ i ] = new JButton( "Button " + (i + 1) );
         buttonPanel.add( buttons[ i ] );
      }

      c.add( buttonPanel, BorderLayout.SOUTH );

      setSize( 425, 150 );
      show();
   }


 public static void main( String args[] )
   {
      PanelDemo app = new PanelDemo();

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