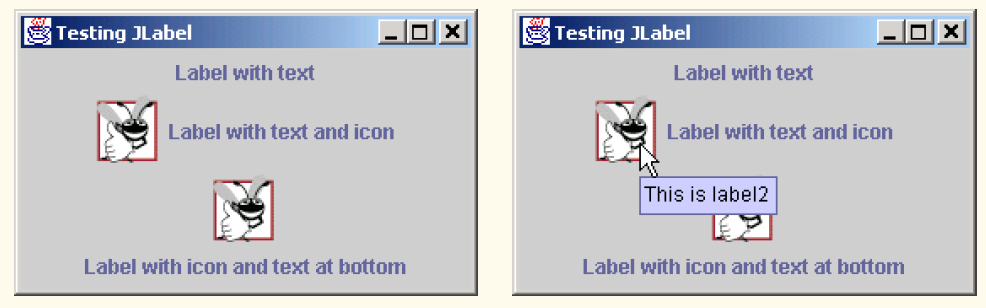
**// Demonstrating the JLabel class**

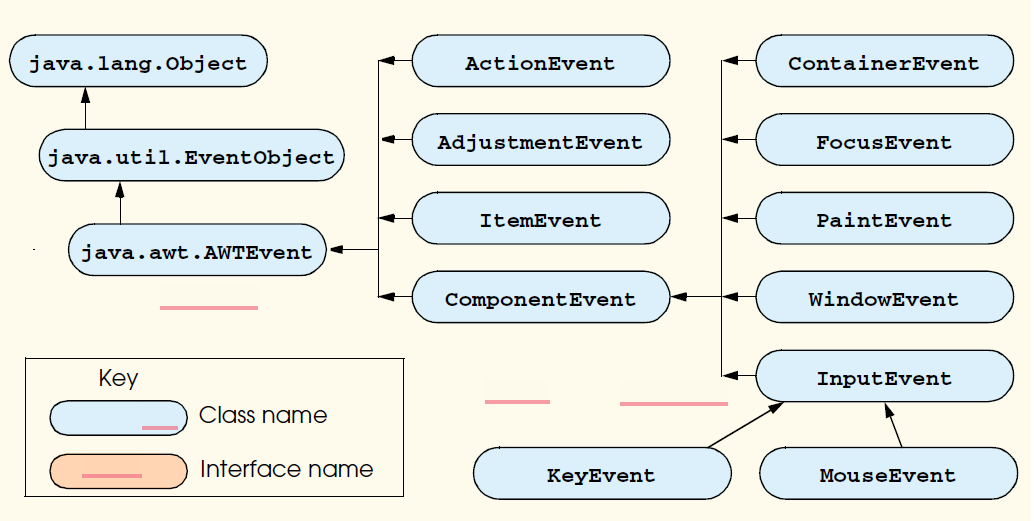
|  |
| --- |
| **1 // Fig. 12.4: LabelTest.java**  **2 // Demonstrating the JLabel class.**  **3**  **4// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class LabelTest extends JFrame {**  **12 private JLabel label1, label2, label3;**  **13**  **14 // set up GUI**  **15 public LabelTest()**  **16 {**  **17 super( "Testing JLabel" );**  **18**  **19 // get content pane and set its layout**  **20 Container container = getContentPane();**  **21 container.setLayout( new FlowLayout() );**  **22**  **23 // JLabel constructor with a string argument**  **24 label1 = new JLabel( "Label with text" );**  **25 label1.setToolTipText( "This is label1" );**  **26 container.add( label1 );**  **27**  **28 // JLabel constructor with string, Icon and**  **29 // alignment arguments**  **30 Icon bug = new ImageIcon( "bug1.gif" );**  **31 label2 = new JLabel( "Label with text and icon",**  **32 bug, SwingConstants.LEFT );**  **33 label2.setToolTipText( "This is label2" );**  **34 container.add( label2 );**  **35**  **36 // JLabel constructor no arguments**  **37 label3 = new JLabel();**  **38 label3.setText( "Label with icon and text at bottom" );**  **39 label3.setIcon( bug );**  **40 label3.setHorizontalTextPosition( SwingConstants.CENTER );**  **41 label3.setVerticalTextPosition( SwingConstants.BOTTOM );**  **42 label3.setToolTipText( "This is label3" );**  **43 container.add( label3 );**  **44**  **45 setSize( 275, 170 );**  **46 setVisible( true );**  **47 }**  **48**  **49 // execute application**  **50 public static void main( String args[] )**  **51 {**  **52 LabelTest application = new LabelTest();**  **53**  **54 application.setDefaultCloseOperation(**  **55 JFrame.EXIT\_ON\_CLOSE );**  **56 }**  **57**  **58 } // end class LabelTest** |



**Event-Handling Model**

GUIs are *event driven* (i.e., they generate *events* when the user of the program interacts with the GUI). Some common interactions are moving the mouse, clicking the mouse, clicking a button, typing in a text field, selecting an item from a menu, closing a window, etc. When a user interaction occurs, an event is sent to the program. GUI

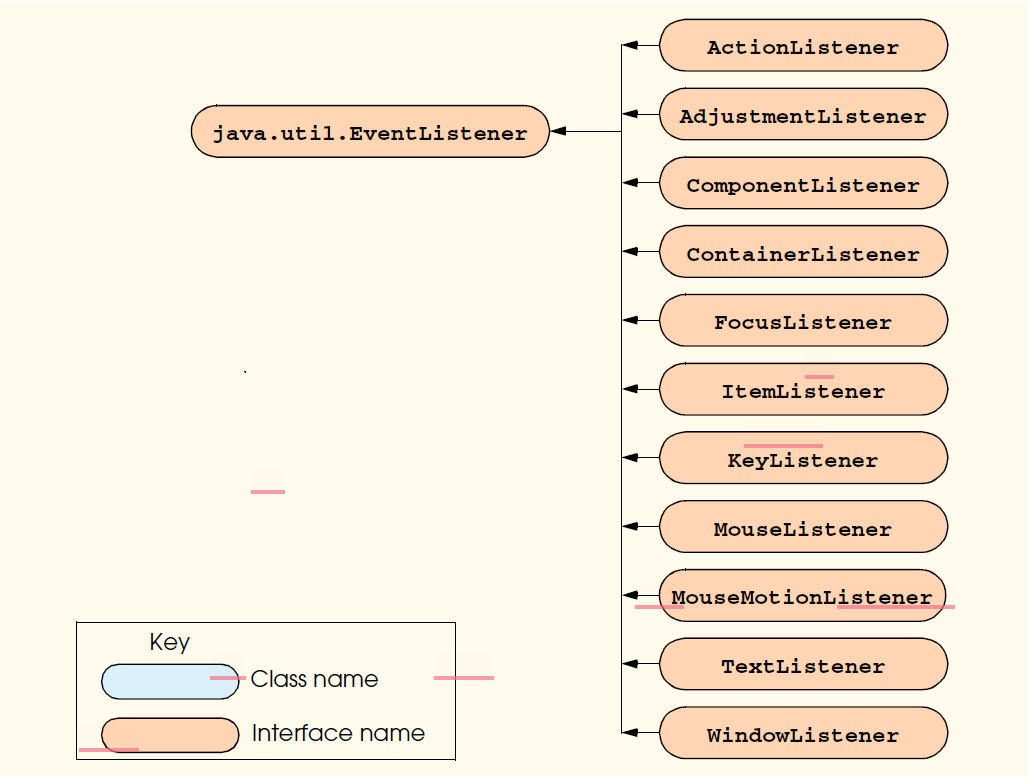
event information is stored in an object of a class that extends **AWTEvent**. The following Figure illustrates a hierarchy containing many of the event classes we use from package **java.** **awt.event**. The event types from package ***java.awt.event*** are used with both AWT and Swing components. Additional event types have also been added that are specific to several types of Swing components. New Swing-component event types are defined in package ***javax.swing.event***.



Some event classes of package **java.awt.event**.

There are three parts to the event-handling mechanism—the *event source*, the *event object* and the *event listener*. The event source is the particular GUI component with whichthe user interacts. The event object encapsulates information about the event that occurred.This information includes a reference to the event source and any event-specific informationthat may be required by the event listener to handle the event. The event listener is anobject that is notified by the event source when an event occurs. The event listener receivesan event object when it is notified of the event, then uses the object to respond to the event.The event source is required to provide methods that enable listeners to be registered andunregistered. The event source also is required to maintain a list of its registered listenersand be able to notify its listeners when an event occurs.

The programmer must perform two key tasks to process a graphical user interface event in a program—register an *event listener* for the GUI component that is expected to generate the event, and implement an *event handling method* (or set of event-handling methods). Commonly, event-handling methods are called *event handlers*. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from package **java.awt.event** and package **javax.swing.event**. Many of the event-listener types are common to both Swing and AWT components. Such types are defined in package **java.awt.event**, and many of these are shown in the following Figure Additional event-listener types that are specific to Swing components are defined in package **javax.swing.event**.



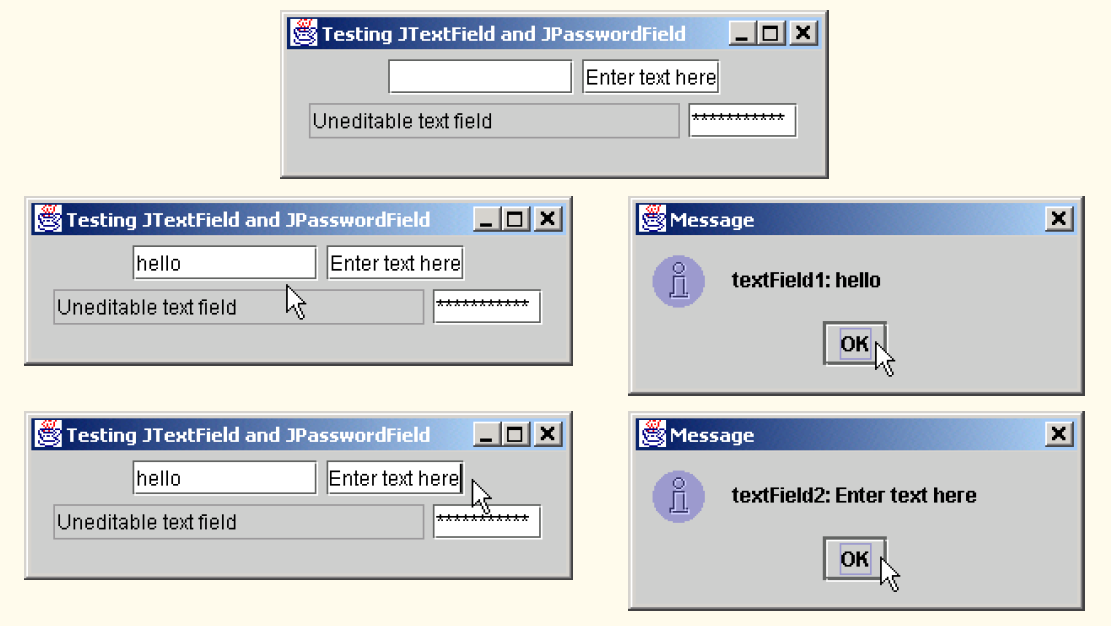
Event-listener interfaces of package **java.awt.event**.

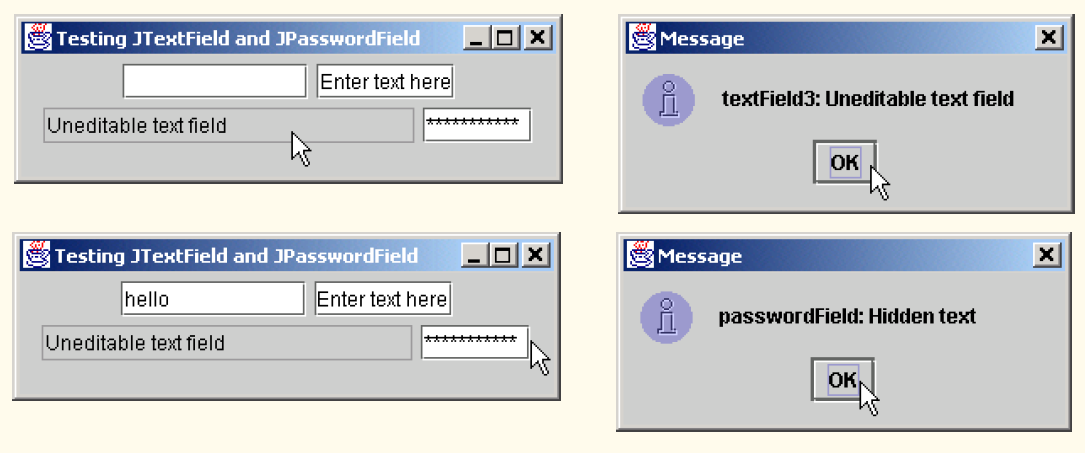
An event listener object “listens” for specific types of events generated by event sources (normally GUI components) in a program. An event handler is a method that is called in response to a particular type of event. Each event-listener interface specifies one or more event-handling methods that *must* be defined in the class that implements the event-listener interface. Remember that interfaces define **abstract** methods. Any class that implements an interface must define all the methods of that interface; otherwise, the class is an **abstract** class and cannot be used to create objects. The use of event listeners in event handling is known as the *delegation event model*—the processing of an event is delegated to a particular object (the listener) in the program.

When an event occurs, the GUI component with which the user interacted notifies its registered listeners by calling each listener’s appropriate event handling method. For example, when the user presses the *Enter* key in a **JTextField**, the registered listener’s **actionPerformed** method is called. How did the event handler get registered? How does the GUI component know to call **actionPerformed** as opposed to some other event handling method? We answer these questions and diagram the interaction as part of the next example.

**JTextField and JPasswordField**

|  |
| --- |
| **1 // Fig. 12.7: TextFieldTest.java**  **2 // Demonstrating the JTextField class.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **7**  **8// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class TextFieldTest extends JFrame {**  **12 private JTextField textField1, textField2, textField3;**  **13 private JPasswordField passwordField;**  **14**  **15 // set up GUI**  **16 public TextFieldTest()**  **17 {**  **18 super( "Testing JTextField and JPasswordField" );**  **19**  **20 Container container = getContentPane();**  **21 container.setLayout( new FlowLayout() );**  **22**  **23 // construct textfield with default sizing**  **24 textField1 = new JTextField( 10 );**  **25 container.add( textField1 );**  **26**  **27 // construct textfield with default text**  **28 textField2 = new JTextField( "Enter text here" );**  **29 container.add( textField2 );**  **30**  **31 // construct textfield with default text and**  **32 // 20 visible elements and no event handler**  **33 textField3 = new JTextField( "Uneditable text field", 20 );**  **34 textField3.setEditable( false );**  **35 container.add( textField3 );**  **36**  **37 // construct textfield with default text**  **38 passwordField = new JPasswordField( "Hidden text" );**  **39 container.add( passwordField );**  **40**  **41 // register event handlers**  **42 TextFieldHandler handler = new TextFieldHandler();**  **43 textField1.addActionListener( handler );**  **44 textField2.addActionListener( handler );**  **45 textField3.addActionListener( handler );**  **46 passwordField.addActionListener( handler );**  **47**  **48 setSize( 325, 100 );**  **49 setVisible( true );**  **50 }**  **51**  **52 // execute application**  **53 public static void main( String args[] )**  **54 {**  **55 TextFieldTest application = new TextFieldTest();**  **56**  **57 application.setDefaultCloseOperation(**  **58 JFrame.EXIT\_ON\_CLOSE );**  **59 }**  **60**  **61 // private inner class for event handling**  **62 private class TextFieldHandler implements ActionListener {**  **63**  **64 // process text field events**  **65 public void actionPerformed( ActionEvent event )**  **66 {**  **67 String string = "";**  **68**  **69 // user pressed Enter in JTextField textField1**  **70 if ( event.getSource() == textField1 )**  **71 string = "textField1: " + event.getActionCommand();**  **72**  **73 // user pressed Enter in JTextField textField2**  **74 else if ( event.getSource() == textField2 )**  **75 string = "textField2: " + event.getActionCommand();**  **76**  **77 // user pressed Enter in JTextField textField3**  **78 else if ( event.getSource() == textField3 )**  **79 string = "textField3: " + event.getActionCommand();**  **80**  **81 // user pressed Enter in JTextField passwordField**  **82 else if ( event.getSource() == passwordField ) {**  **83 JPasswordField pwd =**  **84 ( JPasswordField ) event.getSource();**  **85 string = "passwordField: " +**  **86 new String( passwordField.getPassword() );**  **87 }**  **88**  **89 JOptionPane.showMessageDialog( null, string );**  **90 }**  **91**  **92 } // end private inner class TextFieldHandler**  **93**  **94 } // end class TextFieldTest** |





**How Event Handling Works**

Let us illustrate how the event-handling mechanism works using **textField1** from the preceding example. We have two remaining open questions from Section 12.4:

1. How did the event handler get registered?

2. How does the GUI component know to call **actionPerformed** as opposed to some other event handling method?

The first question is answered by the event registration performed in lines 43–46 of the program. Figure 12.8 diagrams **JTextField** reference **textField1**, the **JTextField** object to which it refers and the listener object that is registered to handle the **JText- Field**’s event.

Every **JComponent** has an object of class ***EventListenerList*** (package **javax.swing.event**) called ***listenerList*** as an instance variable. All registered listeners are stored in the **listenerList** (diagramed as an array in the following Figure). When the statement

**textField1.addActionListener( handler );** executes, a new entry is placed in the **listenerList** for **JTextField textField1**, indicating both the reference to the listener object and the type of listener(in this case **ActionListener**).

The type is important in answering the second question—how does the GUI component know to call **actionPerformed** rather than another event handling method? Every **JComponent** actually supports several different event types, including *mouse events*, *key* *events* and others. When an event occurs, the event is *dispatched* only to the event listeners of the appropriate type. The dispatching of an event is simply calling the event handling

method for each registered listener for that event type.

Each event type has a corresponding event-listener interface. For example, **ActionEvent**s are handled by **ActionListener**s, ***MouseEvent****s* are handled by ***MouseListener****s*(and ***MouseMotionListener****s* as we will see) and ***KeyEvent****s* are handled by ***KeyListener****s*. When an event is generated by a user interaction with a component, thecomponent is handed a unique *event ID* specifying the event type that occurred. The GUIcomponent uses the event ID to decide the type of listener to which the event should be dispatchedand the method to call. In the case of an **ActionEvent**, the event is dispatched toevery registered **ActionListener**’s **actionPerformed** method (the only method ininterface **ActionListener**). In the case of a **MouseEvent**, the event is dispatched toevery registered **MouseListener** (or **MouseMotionListener**, depending on theevent that occurs). The event ID of the **MouseEvent** determines which of the seven different

mouse event handling methods are called. All of this decision logic is handled for you by the GUI components. We discuss other event types and event-listener interfaces as they are needed with each new component we cover.

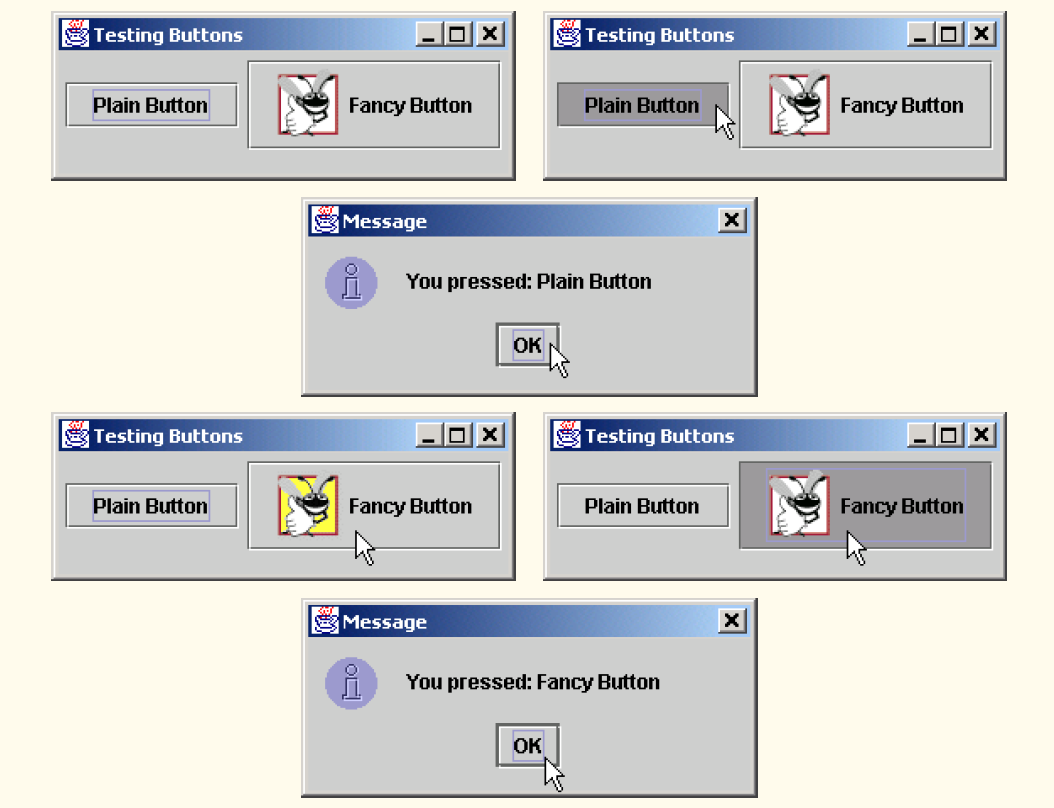
A screenshot of a computer program

Description automatically generated

Event registration for **JTextField textField1**.

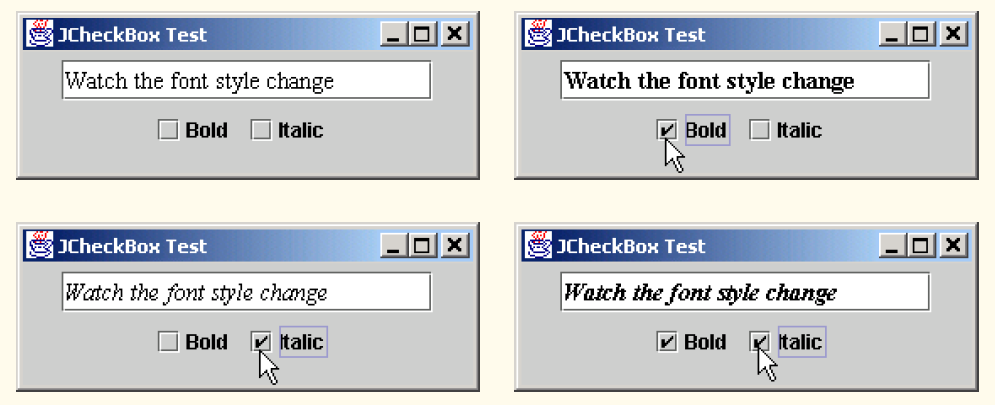
**JButton**

|  |
| --- |
| **1 // Fig. 12.10: ButtonTest.java**  **2 // Creating JButtons.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class ButtonTest extends JFrame {**  **12 private JButton plainButton, fancyButton;**  **13**  **14 // set up GUI**  **15 public ButtonTest()**  **16 {**  **17 super( "Testing Buttons" );**  **18**  **19 // get content pane and set its layout**  **20 Container container = getContentPane();**  **21 container.setLayout( new FlowLayout() );**  **22**  **23 // create buttons**  **24 plainButton = new JButton( "Plain Button" );**  **25 container.add( plainButton );**  **26**  **27 Icon bug1 = new ImageIcon( "bug1.gif" );**  **28 Icon bug2 = new ImageIcon( "bug2.gif" );**  **29 fancyButton = new JButton( "Fancy Button", bug1 );**  **30 fancyButton.setRolloverIcon( bug2 );**  **31 container.add( fancyButton );**  **32**  **33 // create an instance of inner class ButtonHandler**  **34 // to use for button event handling**  **35 ButtonHandler handler = new ButtonHandler();**  **36 fancyButton.addActionListener( handler );**  **37 plainButton.addActionListener( handler );**  **38**  **39 setSize( 275, 100 );**  **40 setVisible( true );**  **41 }**  **42**  **43 // execute application**  **44 public static void main( String args[] )**  **45 {**  **46 ButtonTest application = new ButtonTest();**  **47**  **48 application.setDefaultCloseOperation(**  **49 JFrame.EXIT\_ON\_CLOSE );**  **50 }**  **51**  **52 // inner class for button event handling**  **53 private class ButtonHandler implements ActionListener {**  **54**  **55 // handle button event**  **56 public void actionPerformed( ActionEvent event )**  **57 {**  **58 JOptionPane.showMessageDialog( null,**  **59 "You pressed: " + event.getActionCommand() );**  **60 }**  **61**  **62 } // end private inner class ButtonHandler**  **63**  **64 } // end class ButtonTest** |



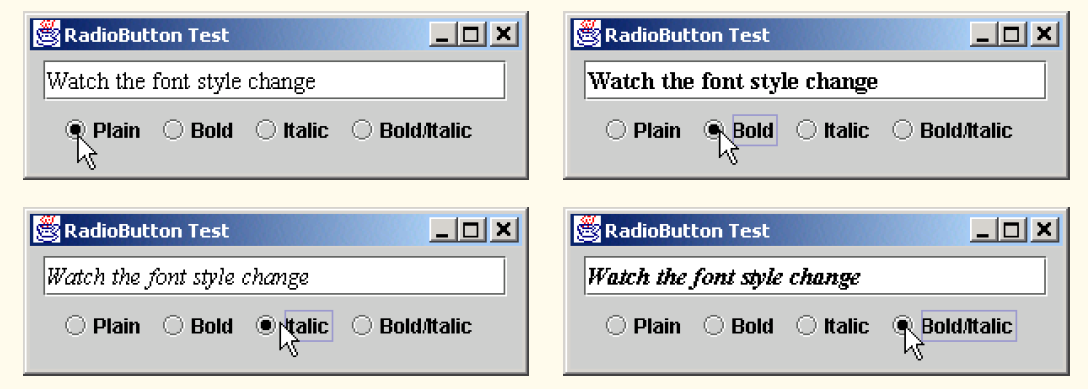
**JCheckBox and JRadioButton**

|  |
| --- |
| **1 // Fig. 12.11: CheckBoxTest.java**  **2 // Creating Checkbox buttons.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class CheckBoxTest extends JFrame {**  **12 private JTextField field;**  **13 private JCheckBox bold, italic;**  **14**  **15 // set up GUI**  **16 public CheckBoxTest()**  **17 {**  **18 super( "JCheckBox Test" );**  **19**  **20 // get content pane and set its layout**  **21 Container container = getContentPane();**  **22 container.setLayout( new FlowLayout() );**  **23**  **24 // set up JTextField and set its font**  **25 field =**  **26 new JTextField( "Watch the font style change", 20 );**  **27 field.setFont( new Font( "Serif", Font.PLAIN, 14 ) );**  **28 container.add( field );**  **29**  **30 // create checkbox objects**  **31 bold = new JCheckBox( "Bold" );**  **32 container.add( bold );**  **33**  **34 italic = new JCheckBox( "Italic" );**  **35 container.add( italic );**  **36**  **37 // register listeners for JCheckBoxes**  **38 CheckBoxHandler handler = new CheckBoxHandler();**  **39 bold.addItemListener( handler );**  **40 italic.addItemListener( handler );**  **41**  **42 setSize( 275, 100 );**  **43 setVisible( true );**  **44 }**  **45**  **46 // execute application**  **47 public static void main( String args[] )**  **48 {**  **49 CheckBoxTest application = new CheckBoxTest();**  **50**  **51 application.setDefaultCloseOperation(**  **52 JFrame.EXIT\_ON\_CLOSE );**  **53 }**  **54**  **55 // private inner class for ItemListener event handling**  **56 private class CheckBoxHandler implements ItemListener {**  **57 private int valBold = Font.PLAIN;**  **58 private int valItalic = Font.PLAIN;**  **59**  **60 // respond to checkbox events**  **61 public void itemStateChanged( ItemEvent event )**  **62 {**  **63 // process bold checkbox events**  **64 if ( event.getSource() == bold )**  **65**  **66 if ( event.getStateChange() == ItemEvent.SELECTED )**  **67 valBold = Font.BOLD;**  **68 else**  **69 valBold = Font.PLAIN;**  **70**  **71 // process italic checkbox events**  **72 if ( event.getSource() == italic )**  **73**  **74 if ( event.getStateChange() == ItemEvent.SELECTED )**  **75 valItalic = Font.ITALIC;**  **76 else**  **77 valItalic = Font.PLAIN;**  **78**  **79 // set text field font**  **80 field.setFont(**  **81 new Font( "Serif", valBold + valItalic, 14 ) );**  **82 }**  **83**  **84 } // end private inner class CheckBoxHandler**  **85**  **86 } // end class CheckBoxTest** |



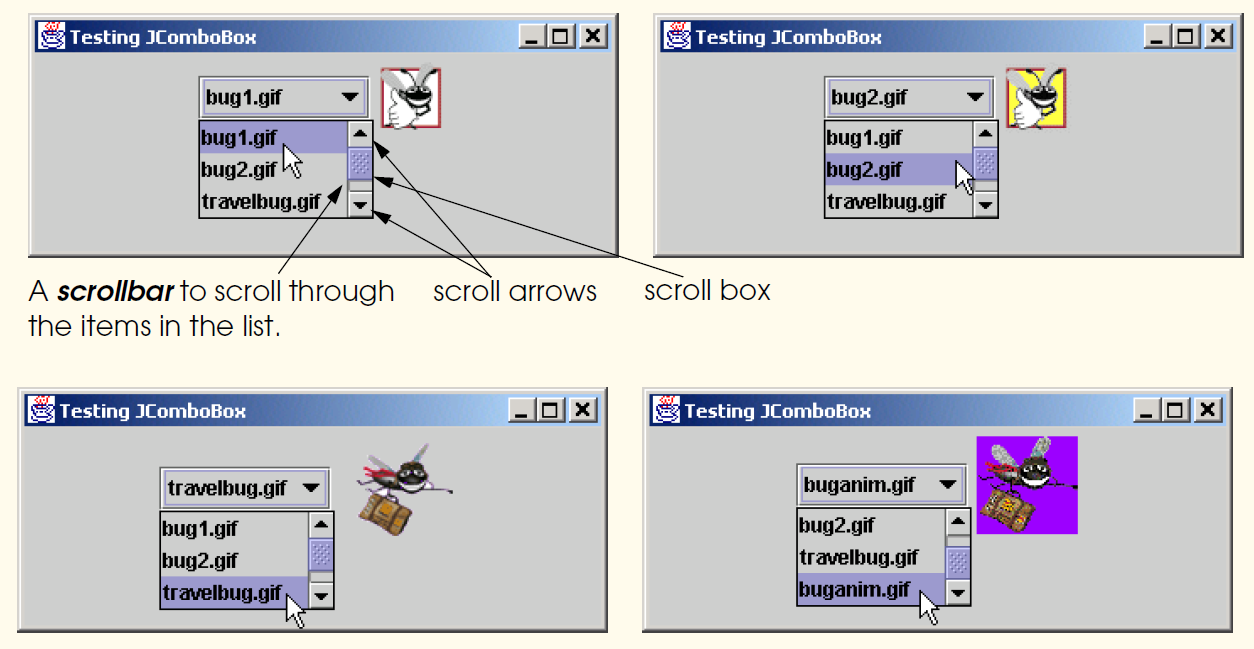
**Creating radio buttons using ButtonGroup and JRadioButton**

|  |
| --- |
| **// Fig. 12.12: RadioButtonTest.java**  **2 // Creating radio buttons using ButtonGroup and JRadioButton.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class RadioButtonTest extends JFrame {**  **12 private JTextField field;**  **13 private Font plainFont, boldFont, italicFont, boldItalicFont;**  **14 private JRadioButton plainButton, boldButton, italicButton,**  **15 boldItalicButton;**  **16 private ButtonGroup radioGroup;**  **17**  **18 // create GUI and fonts**  **19 public RadioButtonTest()**  **20 {**  **21 super( "RadioButton Test" );**  **22**  **23 // get content pane and set its layout**  **24 Container container = getContentPane();**  **25 container.setLayout( new FlowLayout() );**  **26**  **27 // set up JTextField**  **28 field =**  **29 new JTextField( "Watch the font style change", 25 );**  **30 container.add( field );**  **31**  **32 // create radio buttons**  **33 plainButton = new JRadioButton( "Plain", true );**  **34 container.add( plainButton );**  **35**  **36 boldButton = new JRadioButton( "Bold", false);**  **37 container.add( boldButton );**  **38**  **39 italicButton = new JRadioButton( "Italic", false );**  **40 container.add( italicButton );**  **41**  **42 boldItalicButton = new JRadioButton(**  **43 "Bold/Italic", false );**  **44 container.add( boldItalicButton );**  **45**  **46 // register events for JRadioButtons**  **47 RadioButtonHandler handler = new RadioButtonHandler();**  **48 plainButton.addItemListener( handler );**  **49 boldButton.addItemListener( handler );**  **50 italicButton.addItemListener( handler );**  **51 boldItalicButton.addItemListener( handler );**  **52**  **53 // create logical relationship between JRadioButtons**  **54 radioGroup = new ButtonGroup();**  **55 radioGroup.add( plainButton );**  **56 radioGroup.add( boldButton );**  **57 radioGroup.add( italicButton );**  **58 radioGroup.add( boldItalicButton );**  **59**  **60 // create font objects**  **61 plainFont = new Font( "Serif", Font.PLAIN, 14 );**  **62 boldFont = new Font( "Serif", Font.BOLD, 14 );**  **63 italicFont = new Font( "Serif", Font.ITALIC, 14 );**  **64 boldItalicFont =**  **65 new Font( "Serif", Font.BOLD + Font.ITALIC, 14 );**  **66 field.setFont( plainFont );**  **67**  **68 setSize( 300, 100 );**  **69 setVisible( true );**  **70 }**  **71**  **72 // execute application**  **73 public static void main( String args[] )**  **74 {**  **75 RadioButtonTest application = new RadioButtonTest();**  **76**  **77 application.setDefaultCloseOperation(**  **78 JFrame.EXIT\_ON\_CLOSE );**  **79 }**  **80**  **81 // private inner class to handle radio button events**  **82 private class RadioButtonHandler implements ItemListener {**  **83**  **84 // handle radio button events**  **85 public void itemStateChanged( ItemEvent event )**  **86 {**  **87 // user clicked plainButton**  **88 if ( event.getSource() == plainButton )**  **89 field.setFont( plainFont );**  **90**  **91 // user clicked boldButton**  **92 else if ( event.getSource() == boldButton )**  **93 field.setFont( boldFont );**  **94**  **95 // user clicked italicButton**  **96 else if ( event.getSource() == italicButton )**  **97 field.setFont( italicFont );**  **98**  **99 // user clicked boldItalicButton**  **100 else if ( event.getSource() == boldItalicButton )**  **101 field.setFont( boldItalicFont );**  **102 }**  **103**  **104 } // end private inner class RadioButtonHandler**  **105**  **106 } // end class RadioButtonTest** |



**JComboBox**

|  |
| --- |
| **// Fig. 12.13: ComboBoxTest.java**  **2 // Using a JComboBox to select an image to display.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class ComboBoxTest extends JFrame {**  **12 private JComboBox imagesComboBox;**  **13 private JLabel label;**  **14**  **15 private String names[] =**  **16 { "bug1.gif", "bug2.gif", "travelbug.gif", "buganim.gif" };**  **17 private Icon icons[] = { new ImageIcon( names[ 0 ] ),**  **18 new ImageIcon( names[ 1 ] ), new ImageIcon( names[ 2 ] ),**  **19 new ImageIcon( names[ 3 ] ) };**  **20**  **21 // set up GUI**  **22 public ComboBoxTest()**  **23 {**  **24 super( "Testing JComboBox" );**  **25**  **26 // get content pane and set its layout**  **27 Container container = getContentPane();**  **28 container.setLayout( new FlowLayout() );**  **29**  **30 // set up JComboBox and register its event handler**  **31 imagesComboBox = new JComboBox( names );**  **32 imagesComboBox.setMaximumRowCount( 3 );**  **33**  **34 imagesComboBox.addItemListener(**  **35**  **36 // anonymous inner class to handle JComboBox events**  **37 new ItemListener() {**  **38**  **39 // handle JComboBox event**  **40 public void itemStateChanged( ItemEvent event )**  **41 {**  **42 // determine whether check box selected**  **43 if ( event.getStateChange() == ItemEvent.SELECTED )**  **44 label.setIcon( icons[**  **45 imagesComboBox.getSelectedIndex() ] );**  **46 }**  **47**  **48 } // end anonymous inner class**  **49**  **50 ); // end call to addItemListener**  **51**  **52 container.add( imagesComboBox );**  **53**  **54 // set up JLabel to display ImageIcons**  **55 label = new JLabel( icons[ 0 ] );**  **56 container.add( label );**  **57**  **58 setSize( 350, 100 );**  **59 setVisible( true );**  **60 }**  **61**  **62 // execute application**  **63 public static void main( String args[] )**  **64 {**  **65 ComboBoxTest application = new ComboBoxTest();**  **66**  **67 application.setDefaultCloseOperation(**  **68 JFrame.EXIT\_ON\_CLOSE );**  **69 }**  **70**  **71 } // end class ComboBoxTest** |



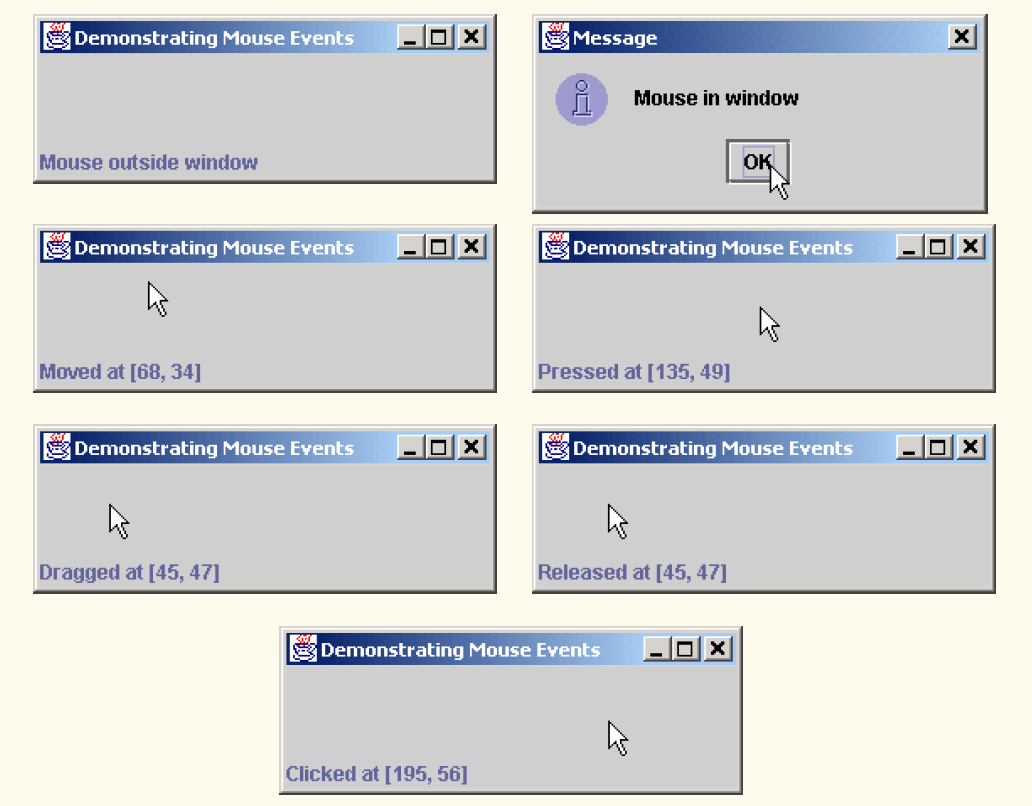
**JList**

|  |
| --- |
| **1 // Fig. 12.14: ListTest.java**  **2 // Selecting colors from a JList.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **67**  **// Java extension packages**  **8 import javax.swing.\*;**  **9 import javax.swing.event.\*;**  **10**  **11 public class ListTest extends JFrame {**  **12 private JList colorList;**  **13 private Container container;**  **14**  **15 private String colorNames[] = { "Black", "Blue", "Cyan",**  **16 "Dark Gray", "Gray", "Green", "Light Gray", "Magenta",**  **17 "Orange", "Pink", "Red", "White", "Yellow" };**  **18**  **19 private Color colors[] = { Color.black, Color.blue,**  **20 Color.cyan, Color.darkGray, Color.gray, Color.green,**  **21 Color.lightGray, Color.magenta, Color.orange, Color.pink,**  **22 Color.red, Color.white, Color.yellow };**  **23**  **24 // set up GUI**  **25 public ListTest()**  **26 {**  **27 super( "List Test" );**  **28**  **29 // get content pane and set its layout**  **30 container = getContentPane();**  **31 container.setLayout( new FlowLayout() );**  **32**  **33 // create a list with items in colorNames array**  **34 colorList = new JList( colorNames );**  **35 colorList.setVisibleRowCount( 5 );**  **36**  **37 // do not allow multiple selections**  **38 colorList.setSelectionMode(**  **39 ListSelectionModel.SINGLE\_SELECTION );**  **40**  **41 // add a JScrollPane containing JList to content pane**  **42 container.add( new JScrollPane( colorList ) );**  **43**  **44 // set up event handler**  **45 colorList.addListSelectionListener(**  **46**  **47 // anonymous inner class for list selection events**  **48 new ListSelectionListener() {**  **49**  **50 // handle list selection events**  **51 public void valueChanged( ListSelectionEvent event )**  **52 {**  **53 container.setBackground(**  **54 colors[ colorList.getSelectedIndex() ] );**  **55 }**  **56**  **57 } // end anonymous inner class**  **58**  **59 ); // end call to addListSelectionListener**  **60**  **61 setSize( 350, 150 );**  **62 setVisible( true );**  **63 }**  **64**  **65 // execute application**  **66 public static void main( String args[] )**  **67 {**  **68 ListTest application = new ListTest();**  **69**  **70 application.setDefaultCloseOperation(**  **71 JFrame.EXIT\_ON\_CLOSE );**  **72 }**  **73**  **74 } // end class ListTest** |



**Mouse Event Handling**

|  |
| --- |
| **1 // Fig. 12.17: MouseTracker.java**  **2 // Demonstrating mouse events.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class MouseTracker extends JFrame**  **12 implements MouseListener, MouseMotionListener {**  **13**  **14 private JLabel statusBar;**  **15**  **16 // set up GUI and register mouse event handlers**  **17 public MouseTracker()**  **18 {**  **19 super( "Demonstrating Mouse Events" );**  **20**  **21 statusBar = new JLabel();**  **22 getContentPane().add( statusBar, BorderLayout.SOUTH );**  **23**  **24 // application listens to its own mouse events**  **25 addMouseListener( this );**  **26 addMouseMotionListener( this );**  **27**  **28 setSize( 275, 100 );**  **29 setVisible( true );**  **30 }**  **31**  **32 // MouseListener event handlers**  **33**  **34 // handle event when mouse released immediately after press**  **35 public void mouseClicked( MouseEvent event )**  **36 {**  **37 statusBar.setText( "Clicked at [" + event.getX() +**  **38 ", " + event.getY() + "]" );**  **39 }**  **40**  **41 // handle event when mouse pressed**  **42 public void mousePressed( MouseEvent event )**  **43 {**  **44 statusBar.setText( "Pressed at [" + event.getX() +**  **45 ", " + event.getY() + "]" );**  **46 }**  **47**  **48 // handle event when mouse released after dragging**  **49 public void mouseReleased( MouseEvent event )**  **50 {**  **51 statusBar.setText( "Released at [" + event.getX() +**  **52 ", " + event.getY() + "]" );**  **53 }**  **54**  **55 // handle event when mouse enters area**  **56 public void mouseEntered( MouseEvent event )**  **57 {**  **58 JOptionPane.showMessageDialog( null, "Mouse in window" );**  **59 }**  **60**  **61 // handle event when mouse exits area**  **62 public void mouseExited( MouseEvent event )**  **63 {**  **64 statusBar.setText( "Mouse outside window" );**  **65 }**  **66**  **67 // MouseMotionListener event handlers**  **68**  **69 // handle event when user drags mouse with button pressed**  **70 public void mouseDragged( MouseEvent event )**  **71 {**  **72 statusBar.setText( "Dragged at [" + event.getX() +**  **73 ", " + event.getY() + "]" );**  **74 }**  **75**  **76 // handle event when user moves mouse**  **77 public void mouseMoved( MouseEvent event )**  **78 {**  **79 statusBar.setText( "Moved at [" + event.getX() +**  **80 ", " + event.getY() + "]" );**  **81 }**  **82**  **83 // execute application**  **84 public static void main( String args[] )**  **85 {**  **86 MouseTracker application = new MouseTracker();**  **87**  **88 application.setDefaultCloseOperation(**  **89 JFrame.EXIT\_ON\_CLOSE );**  **90 }**  **91**  **92 } // end class MouseTracker** |



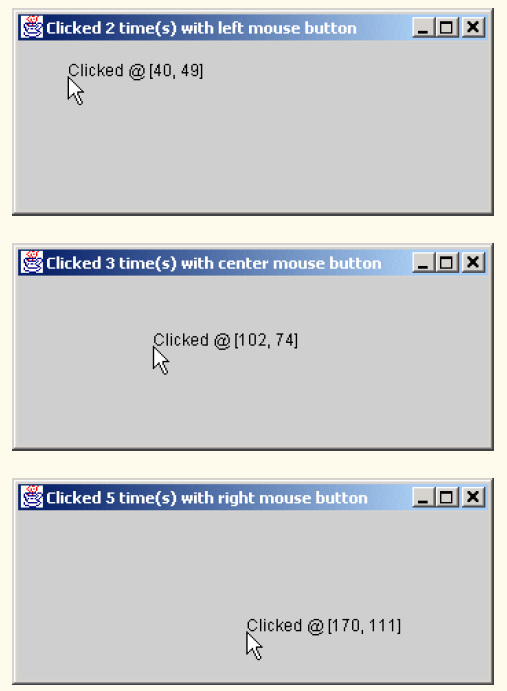
**Painter.java**

|  |
| --- |
| **1 // Fig. 12.19: Painter.java**  **2 // Using class MouseMotionAdapter.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class Painter extends JFrame {**  **12 private int xValue = -10, yValue = -10;**  **13**  **14 // set up GUI and register mouse event handler**  **15 public Painter()**  **16 {**  **17 super( "A simple paint program" );**  **18**  **19 // create a label and place it in SOUTH of BorderLayout**  **20 getContentPane().add(**  **21 new Label( "Drag the mouse to draw" ),**  **22 BorderLayout.SOUTH );**  **23**  **24 addMouseMotionListener(**  **25**  **26 // anonymous inner class**  **27 new MouseMotionAdapter() {**  **28**  **29 // store drag coordinates and repaint**  **30 public void mouseDragged( MouseEvent event )**  **31 {**  **32 xValue = event.getX();**  **33 yValue = event.getY();**  **34 repaint();**  **35 }**  **36**  **37 } // end anonymous inner class**  **38**  **39 ); // end call to addMouseMotionListener**  **40**  **41 setSize( 300, 150 );**  **42 setVisible( true );**  **43 }**  **44**  **45 // draw oval in a 4-by-4 bounding box at the specified**  **46 // location on the window**  **47 public void paint( Graphics g )**  **48 {**  **49 // we purposely did not call super.paint( g ) here to**  **50 // prevent repainting**  **51**  **52 g.fillOval( xValue, yValue, 4, 4 );**  **53 }**  **54**  **55 // execute application**  **56 public static void main( String args[] )**  **57 {**  **58 Painter application = new Painter();**  **59**  **60 application.addWindowListener(**  **61**  **62 // adapter to handle only windowClosing event**  **63 new WindowAdapter() {**  **64**  **65 public void windowClosing( WindowEvent event )**  **66 {**  **67 System.exit( 0 );**  **68 }**  **69**  **70 } // end anonymous inner class**  **71**  **72 ); // end call to addWindowListener**  **73 }**  **74**  **75 } // end class Painter** |

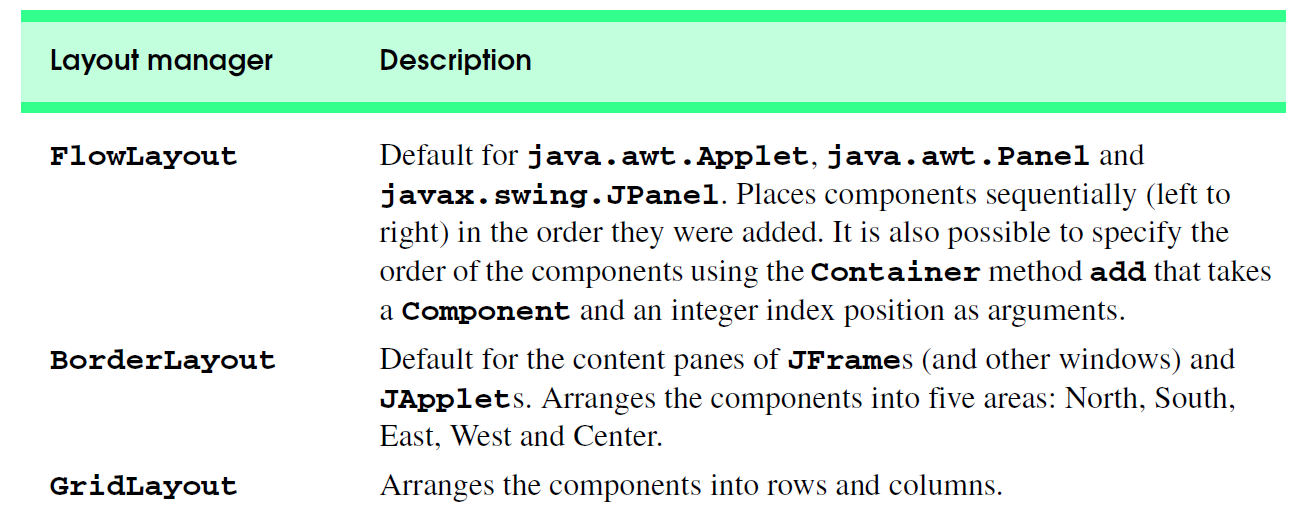


**MouseDetails.java**

|  |
| --- |
| **1 // Fig. 12.20: MouseDetails.java**  **2 // Demonstrating mouse clicks and**  **3 // distinguishing between mouse buttons.**  **45**  **// Java core packages**  **6 import java.awt.\*;**  **7 import java.awt.event.\*;**  **89**  **// Java extension packages**  **10 import javax.swing.\*;**  **11**  **12 public class MouseDetails extends JFrame {**  **13 private int xPos, yPos;**  **14**  **15 // set title bar String, register mouse listener and size**  **16 // and show window**  **17 public MouseDetails()**  **18 {**  **19 super( "Mouse clicks and buttons" );**  **20**  **21 addMouseListener( new MouseClickHandler() );**  **22**  **23 setSize( 350, 150 );**  **24 setVisible( true );**  **25 }**  **26**  **27 // draw String at location where mouse was clicked**  **28 public void paint( Graphics g )**  **29 {**  **30 // call superclass's paint method**  **31 super.paint( g );**  **32**  **33 g.drawString( "Clicked @ [" + xPos + ", " + yPos + "]",**  **34 xPos, yPos );**  **35 }**  **36**  **37 // execute application**  **38 public static void main( String args[] )**  **39 {**  **40 MouseDetails application = new MouseDetails();**  **41**  **42 application.setDefaultCloseOperation(**  **43 JFrame.EXIT\_ON\_CLOSE );**  **44 }**  **45**  **46 // inner class to handle mouse events**  **47 private class MouseClickHandler extends MouseAdapter {**  **48**  **49 // handle mouse click event and determine which mouse**  **50 // button was pressed**  **51 public void mouseClicked( MouseEvent event )**  **52 {**  **53 xPos = event.getX();**  **54 yPos = event.getY();**  **55**  **56 String title =**  **57 "Clicked " + event.getClickCount() + " time(s)";**  **58**  **59 // right mouse button**  **60 if ( event.isMetaDown() )**  **61 title += " with right mouse button";**  **62**  **63 // middle mouse button**  **64 else if ( event.isAltDown() )**  **65 title += " with center mouse button";**  **66**  **67 // left mouse button**  **68 else**  **69 title += " with left mouse button";**  **70**  **71 setTitle( title ); // set title bar of window**  **72 repaint();**  **73 }**  **74**  **75 } // end private inner class MouseClickHandler**  **76**  **77 } // end class MouseDetails** |

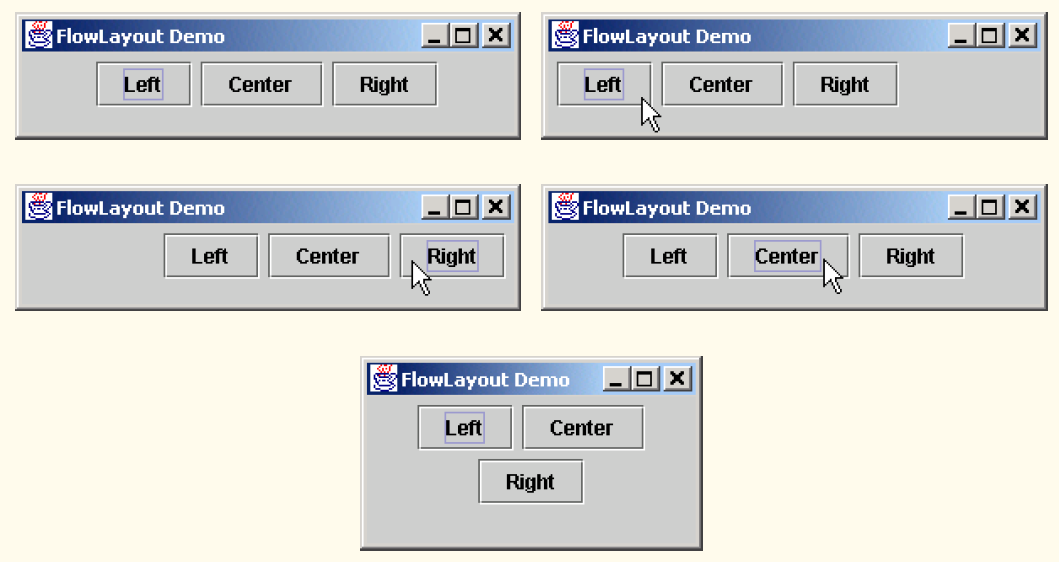


**Layout Managers**



**FlowLayout**

|  |
| --- |
| **1 // Fig. 12.24: FlowLayoutDemo.java**  **2 // Demonstrating FlowLayout alignments.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class FlowLayoutDemo extends JFrame {**  **12 private JButton leftButton, centerButton, rightButton;**  **13 private Container container;**  **14 private FlowLayout layout;**  **15**  **16 // set up GUI and register button listeners**  **17 public FlowLayoutDemo()**  **18 {**  **19 super( "FlowLayout Demo" );**  **20**  **21 layout = new FlowLayout();**  **22**  **23 // get content pane and set its layout**  **24 container = getContentPane();**  **25 container.setLayout( layout );**  **26**  **27 // set up leftButton and register listener**  **28 leftButton = new JButton( "Left" );**  **29**  **30 leftButton.addActionListener(**  **31**  **32 // anonymous inner class**  **33 new ActionListener() {**  **34**  **35 // process leftButton event**  **36 public void actionPerformed( ActionEvent event )**  **37 {**  **38 layout.setAlignment( FlowLayout.LEFT );**  **39**  **40 // re-align attached components**  **41 layout.layoutContainer( container );**  **42 }**  **43**  **44 } // end anonymous inner class**  **45**  **46 ); // end call to addActionListener**  **47**  **48 container.add( leftButton );**  **49**  **50 // set up centerButton and register listener**  **51 centerButton = new JButton( "Center" );**  **52**  **53 centerButton.addActionListener(**  **54**  **55 // anonymous inner class**  **56 new ActionListener() {**  **57**  **58 // process centerButton event**  **59 public void actionPerformed( ActionEvent event )**  **60 {**  **61 layout.setAlignment( FlowLayout.CENTER );**  **62**  **63 // re-align attached components**  **64 layout.layoutContainer( container );**  **65 }**  **66 }**  **67 );**  **68**  **69 container.add( centerButton );**  **70**  **71 // set up rightButton and register listener**  **72 rightButton = new JButton( "Right" );**  **73**  **74 rightButton.addActionListener(**  **75**  **76 // anonymous inner class**  **77 new ActionListener() {**  **78**  **79 // process rightButton event**  **80 public void actionPerformed( ActionEvent event )**  **81 {**  **82 layout.setAlignment( FlowLayout.RIGHT );**  **83**  **84 // re-align attached components**  **85 layout.layoutContainer( container );**  **86 }**  **87 }**  **88 );**  **89**  **90 container.add( rightButton );**  **91**  **92 setSize( 300, 75 );**  **93 setVisible( true );**  **94 }**  **95**  **96 // execute application**  **97 public static void main( String args[] )**  **98 {**  **99 FlowLayoutDemo application = new FlowLayoutDemo();**  **100**  **101 application.setDefaultCloseOperation(**  **102 JFrame.EXIT\_ON\_CLOSE );**  **103 }**  **104**  **105 } // end class FlowLayoutDemo** |



**BorderLayout**

|  |
| --- |
| **1 // Fig. 12.25: BorderLayoutDemo.java**  **2 // Demonstrating BorderLayout.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class BorderLayoutDemo extends JFrame**  **12 implements ActionListener {**  **13**  **14 private JButton buttons[];**  **15 private String names[] = { "Hide North", "Hide South",**  **16 "Hide East", "Hide West", "Hide Center" };**  **17 private BorderLayout layout;**  **18**  **19 // set up GUI and event handling**  **20 public BorderLayoutDemo()**  **21 {**  **22 super( "BorderLayout Demo" );**  **23**  **24 layout = new BorderLayout( 5, 5 );**  **25**  **26 // get content pane and set its layout**  **27 Container container = getContentPane();**  **28 container.setLayout( layout );**  **29**  **30 // instantiate button objects**  **31 buttons = new JButton[ names.length ];**  **32**  **33 for ( int count = 0; count < names.length; count++ ) {**  **34 buttons[ count ] = new JButton( names[ count ] );**  **35 buttons[ count ].addActionListener( this );**  **36 }**  **37**  **38 // place buttons in BorderLayout; order not important**  **39 container.add( buttons[ 0 ], BorderLayout.NORTH );**  **40 container.add( buttons[ 1 ], BorderLayout.SOUTH );**  **41 container.add( buttons[ 2 ], BorderLayout.EAST );**  **42 container.add( buttons[ 3 ], BorderLayout.WEST );**  **43 container.add( buttons[ 4 ], BorderLayout.CENTER );**  **44**  **45 setSize( 300, 200 );**  **46 setVisible( true );**  **47 }**  **48**  **49 // handle button events**  **50 public void actionPerformed( ActionEvent event )**  **51 {**  **52 for ( int count = 0; count < buttons.length; count++ )**  **53**  **54 if ( event.getSource() == buttons[ count ] )**  **55 buttons[ count ].setVisible( false );**  **56 else**  **57 buttons[ count ].setVisible( true );**  **58**  **59 // re-layout the content pane**  **60 layout.layoutContainer( getContentPane() );**  **61 }**  **62**  **63 // execute application**  **64 public static void main( String args[] )**  **65 {**  **66 BorderLayoutDemo application = new BorderLayoutDemo();**  **67**  **68 application.setDefaultCloseOperation(**  **69 JFrame.EXIT\_ON\_CLOSE );**  **70 }**  **71**  **72 } // end class BorderLayoutDemo** |



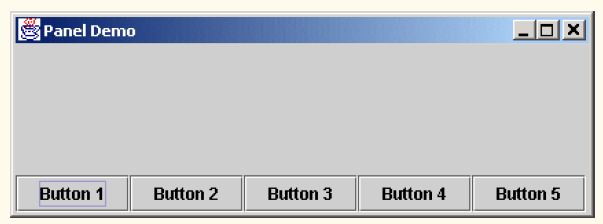
**GridLayout**

|  |
| --- |
| **1 // Fig. 12.26: GridLayoutDemo.java**  **2 // Demonstrating GridLayout.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class GridLayoutDemo extends JFrame**  **12 implements ActionListener {**  **13**  **14 private JButton buttons[];**  **15 private String names[] =**  **16 { "one", "two", "three", "four", "five", "six" };**  **17 private boolean toggle = true;**  **18 private Container container;**  **19 private GridLayout grid1, grid2;**  **20**  **21 // set up GUI**  **22 public GridLayoutDemo()**  **23 {**  **24 super( "GridLayout Demo" );**  **25**  **26 // set up layouts**  **27 grid1 = new GridLayout( 2, 3, 5, 5 );**  **28 grid2 = new GridLayout( 3, 2 );**  **29**  **30 // get content pane and set its layout**  **31 container = getContentPane();**  **32 container.setLayout( grid1 );**  **33**  **34 // create and add buttons**  **35 buttons = new JButton[ names.length ];**  **36**  **37 for ( int count = 0; count < names.length; count++ ) {**  **38 buttons[ count ] = new JButton( names[ count ] );**  **39 buttons[ count ].addActionListener( this );**  **40 container.add( buttons[ count ] );**  **41 }**  **42**  **43 setSize( 300, 150 );**  **44 setVisible( true );**  **45 }**  **46**  **47 // handle button events by toggling between layouts**  **48 public void actionPerformed( ActionEvent event )**  **49 {**  **50 if ( toggle )**  **51 container.setLayout( grid2 );**  **52 else**  **53 container.setLayout( grid1 );**  **54**  **55 toggle = !toggle; // set toggle to opposite value**  **56 container.validate();**  **57 }**  **58**  **59 // execute application**  **60 public static void main( String args[] )**  **61 {**  **62 GridLayoutDemo application = new GridLayoutDemo();**  **63**  **64 application.setDefaultCloseOperation(**  **65 JFrame.EXIT\_ON\_CLOSE );**  **66 }**  **67**  **68 } // end class GridLayoutDemo** |



**Panels**

|  |
| --- |
| **1 // Fig. 12.27: PanelDemo.java**  **2 // Using a JPanel to help lay out components.**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class PanelDemo extends JFrame {**  **12 private JPanel buttonPanel;**  **13 private JButton buttons[];**  **14**  **15 // set up GUI**  **16 public PanelDemo()**  **17 {**  **18 super( "Panel Demo" );**  **19**  **20 // get content pane**  **21 Container container = getContentPane();**  **22**  **23 // create buttons array**  **24 buttons = new JButton[ 5 ];**  **25**  **26 // set up panel and set its layout**  **27 buttonPanel = new JPanel();**  **28 buttonPanel.setLayout(**  **29 new GridLayout( 1, buttons.length ) );**  **30**  **31 // create and add buttons**  **32 for ( int count = 0; count < buttons.length; count++ ) {**  **33 buttons[ count ] =**  **34 new JButton( "Button " + ( count + 1 ) );**  **35 buttonPanel.add( buttons[ count ] );**  **36 }**  **37**  **38 container.add( buttonPanel, BorderLayout.SOUTH );**  **39**  **40 setSize( 425, 150 );**  **41 setVisible( true );**  **42 }**  **43**  **44 // execute application**  **45 public static void main( String args[] )**  **46 {**  **47 PanelDemo application = new PanelDemo();**  **48**  **49 application.setDefaultCloseOperation(**  **50 JFrame.EXIT\_ON\_CLOSE );**  **51 }**  **52**  **53 } // end class PanelDemo** |



**Using Menus with Frames**

|  |
| --- |
| **1 // Fig. 13.10: MenuTest.java**  **2 // Demonstrating menus**  **34**  **// Java core packages**  **5 import java.awt.\*;**  **6 import java.awt.event.\*;**  **78**  **// Java extension packages**  **9 import javax.swing.\*;**  **10**  **11 public class MenuTest extends JFrame {**  **12 private Color colorValues[] =**  **13 { Color.black, Color.blue, Color.red, Color.green };**  **14**  **15 private JRadioButtonMenuItem colorItems[], fonts[];**  **16 private JCheckBoxMenuItem styleItems[];**  **17 private JLabel displayLabel;**  **18 private ButtonGroup fontGroup, colorGroup;**  **19 private int style;**  **20**  **21 // set up GUI**  **22 public MenuTest()**  **23 {**  **24 super( "Using JMenus" );**  **25**  **26 // set up File menu and its menu items**  **27 JMenu fileMenu = new JMenu( "File" );**  **28 fileMenu.setMnemonic( 'F' );**  **29**  **30 // set up About... menu item**  **31 JMenuItem aboutItem = new JMenuItem( "About..." );**  **32 aboutItem.setMnemonic( 'A' );**  **33**  **34 aboutItem.addActionListener(**  **35**  **36 // anonymous inner class to handle menu item event**  **37 new ActionListener() {**  **38**  **39 // display message dialog when user selects About...**  **40 public void actionPerformed( ActionEvent event )**  **41 {**  **42 JOptionPane.showMessageDialog( MenuTest.this,**  **43 "This is an example\nof using menus",**  **44 "About", JOptionPane.PLAIN\_MESSAGE );**  **45 }**  **46**  **47 } // end anonymous inner class**  **48**  **49 ); // end call to addActionListener**  **50**  **51 fileMenu.add( aboutItem );**  **52**  **53 // set up Exit menu item**  **54 JMenuItem exitItem = new JMenuItem( "Exit" );**  **55 exitItem.setMnemonic( 'x' );**  **56**  **57 exitItem.addActionListener(**  **58**  **59 // anonymous inner class to handle exitItem event**  **60 new ActionListener() {**  **61**  **62 // terminate application when user clicks exitItem**  **63 public void actionPerformed( ActionEvent event )**  **64 {**  **65 System.exit( 0 );**  **66 }**  **67**  **68 } // end anonymous inner class**  **69**  **70 ); // end call to addActionListener**  **71**  **72 fileMenu.add( exitItem );**  **73**  **74 // create menu bar and attach it to MenuTest window**  **75 JMenuBar bar = new JMenuBar();**  **76 setJMenuBar( bar );**  **77 bar.add( fileMenu );**  **78**  **79 // create Format menu, its submenus and menu items**  **80 JMenu formatMenu = new JMenu( "Format" );**  **81 formatMenu.setMnemonic( 'r' );**  **82**  **83 // create Color submenu**  **84 String colors[] = { "Black", "Blue", "Red", "Green" };**  **85**  **86 JMenu colorMenu = new JMenu( "Color" );**  **87 colorMenu.setMnemonic( 'C' );**  **88**  **89 colorItems = new JRadioButtonMenuItem[ colors.length ];**  **90 colorGroup = new ButtonGroup();**  **91 ItemHandler itemHandler = new ItemHandler();**  **92**  **93 // create color radio button menu items**  **94 for ( int count = 0; count < colors.length; count++ ) {**  **95 colorItems[ count ] =**  **96 new JRadioButtonMenuItem( colors[ count ] );**  **97**  **98 colorMenu.add( colorItems[ count ] );**  **99 colorGroup.add( colorItems[ count ] );**  **100**  **101 colorItems[ count ].addActionListener( itemHandler );**  **102 }**  **103**  **104 // select first Color menu item**  **105 colorItems[ 0 ].setSelected( true );**  **106**  **107 // add format menu to menu bar**  **108 formatMenu.add( colorMenu );**  **109 formatMenu.addSeparator();**  **110**  **111 // create Font submenu**  **112 String fontNames[] = { "Serif", "Monospaced", "SansSerif" };**  **113**  **114 JMenu fontMenu = new JMenu( "Font" );**  **115 fontMenu.setMnemonic( 'n' );**  **116**  **117 fonts = new JRadioButtonMenuItem[ fontNames.length ];**  **118 fontGroup = new ButtonGroup();**  **119**  **120 // create Font radio button menu items**  **121 for ( int count = 0; count < fonts.length; count++ ) {**  **122 fonts[ count ] =**  **123 new JRadioButtonMenuItem( fontNames[ count ] );**  **124**  **125 fontMenu.add( fonts[ count ] );**  **126 fontGroup.add( fonts[ count ] );**  **127**  **128 fonts[ count ].addActionListener( itemHandler );**  **129 }**  **130**  **131 // select first Font menu item**  **132 fonts[ 0 ].setSelected( true );**  **133**  **134 fontMenu.addSeparator();**  **135**  **136 // set up style menu items**  **137 String styleNames[] = { "Bold", "Italic" };**  **138**  **139 styleItems = new JCheckBoxMenuItem[ styleNames.length ];**  **140 StyleHandler styleHandler = new StyleHandler();**  **141**  **142 // create style checkbox menu items**  **143 for ( int count = 0; count < styleNames.length; count++ ) {**  **144 styleItems[ count ] =**  **145 new JCheckBoxMenuItem( styleNames[ count ] );**  **146**  **147 fontMenu.add( styleItems[ count ] );**  **148**  **149 styleItems[ count ].addItemListener( styleHandler );**  **150 }**  **151**  **152 // put Font menu in Format menu**  **153 formatMenu.add( fontMenu );**  **154**  **155 // add Format menu to menu bar**  **156 bar.add( formatMenu );**  **157**  **158 // set up label to display text**  **159 displayLabel = new JLabel(**  **160 "Sample Text", SwingConstants.CENTER );**  **161 displayLabel.setForeground( colorValues[ 0 ] );**  **162 displayLabel.setFont(**  **163 new Font( "TimesRoman", Font.PLAIN, 72 ) );**  **164**  **165 getContentPane().setBackground( Color.cyan );**  **166 getContentPane().add( displayLabel, BorderLayout.CENTER );**  **167**  **168 setSize( 500, 200 );**  **169 setVisible( true );**  **170**  **171 } // end constructor**  **172**  **173 // execute application**  **174 public static void main( String args[] )**  **175 {**  **176 MenuTest application = new MenuTest();**  **177**  **178 application.setDefaultCloseOperation(**  **179 JFrame.EXIT\_ON\_CLOSE );**  **180 }**  **181**  **182 // inner class to handle action events from menu items**  **183 private class ItemHandler implements ActionListener {**  **184**  **185 // process color and font selections**  **186 public void actionPerformed( ActionEvent event )**  **187 {**  **188 // process color selection**  **189 for ( int count = 0; count < colorItems.length; count++ )**  **190**  **191 if ( colorItems[ count ].isSelected() ) {**  **192 displayLabel.setForeground( colorValues[ count ] );**  **193 break;**  **194 }**  **195**  **196 // process font selection**  **197 for ( int count = 0; count < fonts.length; count++ )**  **198**  **199 if ( event.getSource() == fonts[ count ] ) {**  **200 displayLabel.setFont( new Font(**  **201 fonts[ count ].getText(), style, 72 ) );**  **202 break;**  **203 }**  **204**  **205 repaint();**  **206 }**  **207**  **208 } // end class ItemHandler**  **209**  **210 // inner class to handle item events from check box menu items**  **211 private class StyleHandler implements ItemListener {**  **212**  **213 // process font style selections**  **214 public void itemStateChanged( ItemEvent e )**  **215 {**  **216 style = 0;**  **217**  **218 // check for bold selection**  **219 if ( styleItems[ 0 ].isSelected() )**  **220 style += Font.BOLD;**  **221**  **222 // check for italic selection**  **223 if ( styleItems[ 1 ].isSelected() )**  **224 style += Font.ITALIC;**  **225**  **226 displayLabel.setFont( new Font(**  **227 displayLabel.getFont().getName(), style, 72 ) );**  **228**  **229 repaint();**  **230 }**  **231**  **232 } // end class StyleHandler**  **233**  **234 } // end class MenuTest** |

A screenshot of a computer

Description automatically generated