Capitolo 2 – I componenti GUI di base

Outline

Capitolo 2 – I componenti GUI di base

Outline

2.1 Introduction

- Graphical User Interface ("Goo-ee")
  - Pictorial interface to a program
    - Distinctive "look" and "feel"
  - Different applications with consistent GUIs improve productivity
- GUIs built from components
  - Component: object with which user interacts
  - Examples: Labels, Text fields, Buttons, Checkboxes

2.1 Introduction

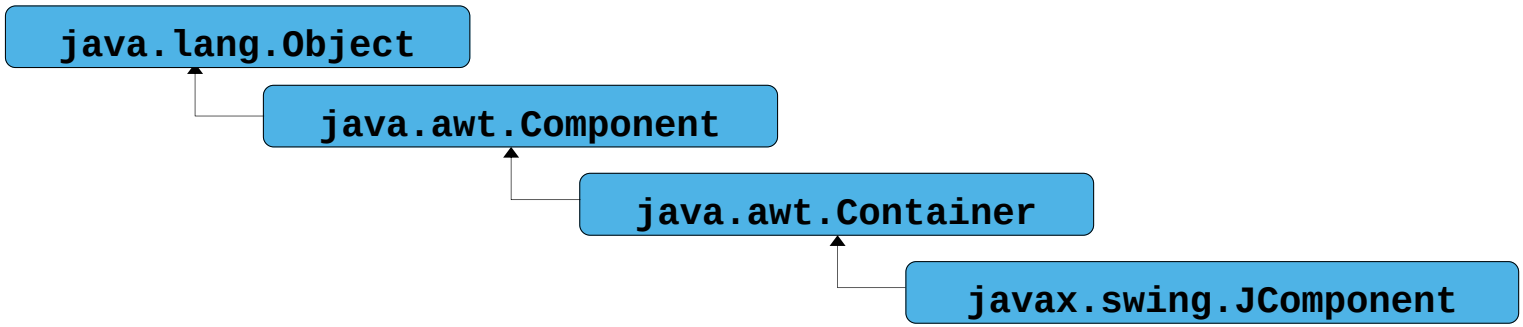
- Example GUI: Netscape Communicator

2.2 Swing Overview

- Swing GUI components
  - Defined in package **`javax.swing`**
  - Original GUI components from Abstract Windowing Toolkit in **`java.awt`**
    - Heavyweight components - rely on local platform's windowing system for look and feel
  - Swing components are lightweight
    - Written in Java, not weighed down by complex GUI capabilities of platform
    - More portable than heavyweight components
  - Swing components allow programmer to specify look and feel
    - Can change depending on platform
    - Can be same across all platforms

2.2 Swing Overview

- Swing component inheritance hierarchy



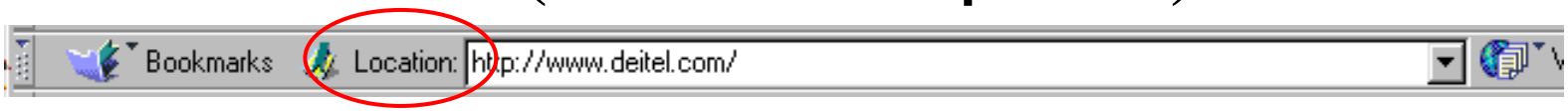
  - **`Component`** defines methods that can be used in its subclasses (for example, **`paint`** and **`repaint`**)
  - **`Container`** - collection of related components
    - When using **`JFrames`**, attach components to the content pane (a **`Container`**)
    - Method **`add`**
  - **`JComponent`** - superclass to most Swing components
  - Much of a component's functionality inherited from these classes

1.2 Swing Overview

- Some capabilities of subclasses of **`JComponent`**
  - Pluggable look and feel
  - Shortcut keys (mnemonics)
    - Direct access to components through keyboard
  - Common event handling
    - If several components perform same actions
  - Tool tips
    - Description of component that appears when mouse over it

2.3 JLabel

- Labels
  - Provide text instructions on a GUI
  - Read-only text
  - Programs rarely change a label's contents
  - Class **JLabel** (subclass of **JComponent**)



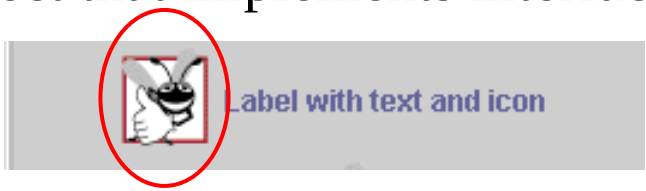
- Methods

```
18        label1 = new JLabel( "Label with text" );
```

    - Can declare label text in constructor
  - **myLabel.setToolTipText("Text")**
    - Displays **"Text"**in a tool tip when mouse over label
  - **myLabel.setText("Text")**
  - **myLabel.getText()**

2.3 JLabel

- **Icon**
  - Object that implements interface **Icon**



  - One class is **ImageIcon** (**.gif** and **.jpeg** images)

```
24        Icon bug = new ImageIcon( "bug1.gif" );
```

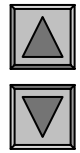
    - Assumed same directory as program (more Chapter 16)
  - Display an icon with **setIcon** method (of class **JLabel**)

```
33        label3.setIcon( bug );
```

    - **myLabel.setIcon(myIcon);**
    - **myLabel.getIcon //returns current Icon**

2.3 JLabel

- Alignment
  - By default, text appears to right of image
  - **JLabel** methods **setHorizontalTextPosition** and **setVerticalTextPosition**
    - Specify where text appears in label
    - Use integer constants defined in interface **SwingConstants** (**javax.swing**)
      - **SwingConstants.LEFT, RIGHT, BOTTOM, CENTER**
- Another **JLabel** constructor
  - **JLabel("Text", ImageIcon, Text_Alignment_CONSTANT)**



Outline

1. import

1.1 Class Labeltest (extends JFrame)

2. Initialize JLabels

```
1  // Fig. 2.4: LabelTest.java
2  // Demonstrating the JLabel class.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class LabelTest extends JFrame {
8     private JLabel label1, label2,
9
10    public LabelTest()
11    {
12       super( "Testing JLabel" );
13
14       Container c = getContentPane();
15       c.setLayout( new FlowLayout() );
16
17       // JLabel constructor with a string argument
18       label1 = new JLabel( "Label with text" );
19       label1.setToolTipText( "This is label1" );
20       c.add( label1 );
21
22       // JLabel constructor with string, Icon and
23       // alignment arguments
24       Icon bug = new ImageIcon( "bug1.gif" );
25       label2 = new JLabel( "Label with text and icon",
26                            bug, SwingConstants.LEFT );
27       label2.setToolTipText( "This is label2" );
28       c.add( label2 );
29
30       // JLabel constructor no arguments
```

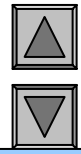
Create a **Container** object, to which we attach **JLabel** objects (subclass of **JComponent**).

Initialize text in **JLabel** constructor.

Set the tool tip text, and attach component to **Container c**.

Create a new **ImageIcon** (assumed to be in same directory as program). More Chapter 16.

Set **ImageIcon** and alignment of text in **JLabel** constructor.



Outline

```
31       label3 = new JLabel();
32       label3.setText( "Label with icon and text at bottom" );
33       label3.setIcon( bug );
34       label3.setHorizontalTextPosition(
35          SwingConstants.CENTER );
36       label3.setVerticalTextPosition(
37          SwingConstants.BOTTOM );
38       label3.setToolTipText( "This is label3" );
39       c.add( label3 );
40
41       setSize( 275, 170 );
42       show();
43    }
44
45    public static void main( String args[] )
46    {
47       LabelTest app = new LabelTest();
48
49       app.addWindowListener(
50          new WindowAdapter() {
51             public void windowClosing( WindowEvent e )
52             {
53                System.exit( 0 );
54             }
55          }
56       );
57    }
58 }
```

Use a no-argument constructor. Set text, icon, and alignment using methods.

ontalText Position

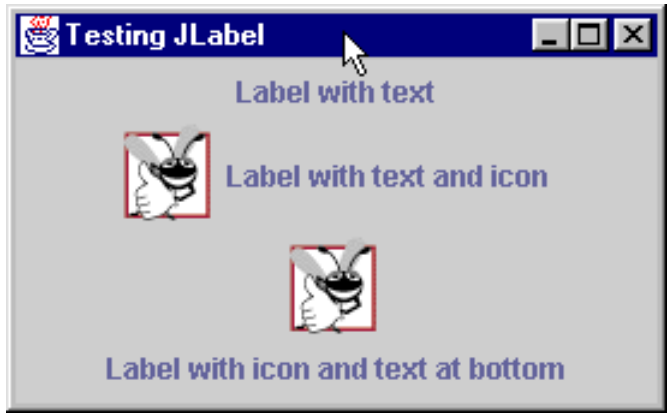
2.3 setVerticalText Position

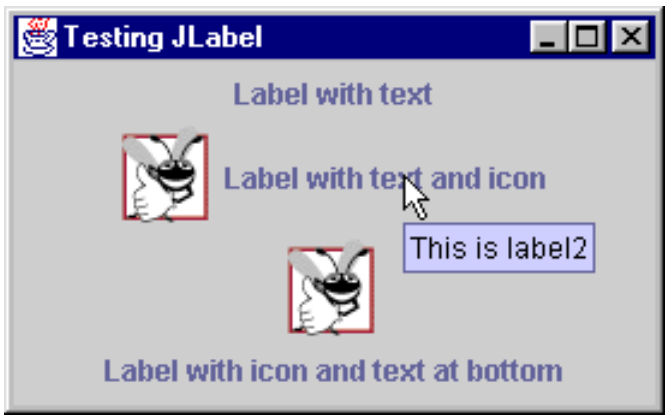
2.3 setToolTipText

3. main

Outline

Program Output





2.4 Event Handling Model

- GUIs are event driven
  - Generate events when user interacts with GUI
    - Mouse movements, mouse clicks, typing in a text field, etc.
  - Event information stored in object that extends **`AWTEvent`**
- To process an event
  - Register an event listener
    - Object from a class that implements an event-listener interface (from **`java.awt.event`** or **`javax.swing.event`**)
    - "Listens" for events
  - Implement event handler
    - Method called in response to event
    - Event handling interface has one or more methods that must be defined

2.4 Event Handling Model

- Delegation event model
  - Use of event listeners in event handling
  - Processing of event delegated to particular object
- When an event occurs
  - GUI component notifies its listeners
    - Calls listener's event handling method
- Example:
  - *Enter* pressed in a **`JTextField`**
  - Method **`actionPerformed`** called for registered listener
  - Details in following sections

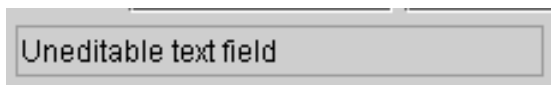
2.5 JTextField and JPasswordField

- **JTextField**s and **JPasswordField**s
  - Single line areas in which text can be entered or displayed
  - **JPasswordField**s show inputted text as an asterisk *

  - **JTextField** extends **JTextComponent**
    - **JPasswordField** extends **JTextField**
- When **Enter** pressed
  - **ActionEvent** occurs
  - Currently active field "has the focus"

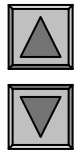
2.5 JTextField and JPasswordField

- Methods
  - Constructors
    - **JTextField(10)**
      - Textfield with 10 columns of text
      - Takes average character width, multiplies by 10
    - **JTextField("Hi")**
      - Sets text, width determined automatically
    - **JTextField("Hi", 20)**
  - **setEditable(boolean)**
    - If **false**, user cannot edit text
    - Can still generate events
  - **getPassword**
    - Class **JPasswordField**
    - Returns password as an **array** of type **char**



2.5 JTextField and JPasswordField

- Class **ActionEvent**
  - Method **getActionCommand**
    - Returns text in **JTextField** that generated event
  - Method **getSource**
    - **getSource** returns **Component** reference
- Example
  - Create **JTextField**s and a **JPasswordField**
  - Create and register an event handler
    - Use **getSource** to determine which component had event
    - Display a dialog box when *Enter* pressed



Outline

1. `import`

1.1 Declarations

1.2 Constructor

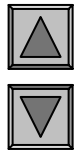
1.3 GUI components

ds

```
1  // Fig. 2.7: TextFieldTest.java
2  // Demonstrating the JTextField class.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TextFieldTest extends JFrame {
8     private JTextField text1, text2, text3;
9     private JPasswordField password;
10
11    public TextFieldTest()
12    {
13       super( "Testing JTextField and JPasswordField" );
14
15       Container c = getContentPane();
16       c.setLayout( new FlowLayout() );
17
18       // construct textfield with default sizing
19       text1 = new JTextField( 10 );
20       c.add( text1 );
21
22       // construct textfield with default text
23       text2 = new JTextField( "Enter text here"
24       c.add( text2 );
25
26       // construct textfield with default text and
27       // 20 visible elements and no event handler
28       text3 = new JTextField( "Uneditable text field", 20 );
29       text3.setEditable( false );
30       c.add( text3 );
```

Create new **`JTextField`** objects using the various constructors.

This text field cannot be modified (has a gray background). It can still generate events.



Outline

```
31
32       // construct textfield with default text
33       password = new JPasswordField( "Hidden text" );
34       c.add( password );
35
36       TextFieldHandler handler = new TextFieldH
37       text1.addActionListener( handler );
38       text2.addActionListener( handler );
39       text3.addActionListener( handler );
40       password.addActionListener( handler );
41
42       setSize( 325, 100 );
43       show();
44    }
45
46    public static void main( String args[] )
47    {
48       TextFieldTest app = new TextFieldTest();
49
50       app.addWindowListener(
51          new WindowAdapter() {
52             public void windowClosing( WindowEvent e )
53             {
54                System.exit( 0 );
55             }
56          }
57       );
58    }
59
60    // inner class for event handling
```

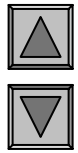
JPasswordField initialized with text, which appears as asterisks.

Register event handlers. Good practice to use an inner class as an event handler.

dField

2.3 Event handler

3 main



Outline

```
61    private class TextFieldHandler implements ActionListener {
62       public void actionPerformed( ActionEvent e )
63       {
64          String s = "";
65
66          if ( e.getSource() == text1 )
67             s = "text1: " + e.getActionCommand();
68          else if ( e.getSource() == text2 )
69             s = "text2: " + e.getActionCommand();
70          else if ( e.getSource() == text3 )
71             s = "text3: " + e.getActionCommand();
72          else if ( e.getSource() == password ) {
73             JPasswordField pwd =
74                (JPasswordField) e.getSource();
75             s = "password: " +
76                 new String( pwd.getPass
77          }
78
79          JOptionPane.showMessageDialog( null, s );
80       }
81    }
82 }
```

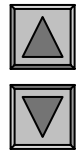
4. Inner class `TextFieldHandler` (event handler)

`getActionCommand`

4.3 Downcast reference

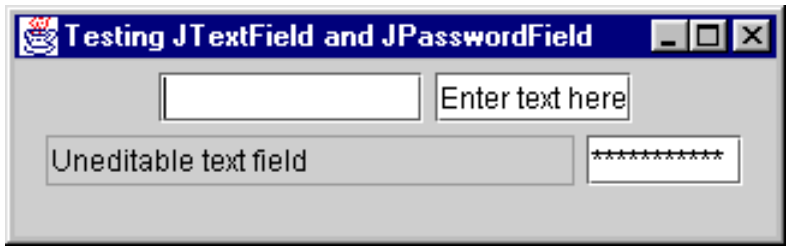
Use **`getActionCommand`** to get the text in the text field that had the event.

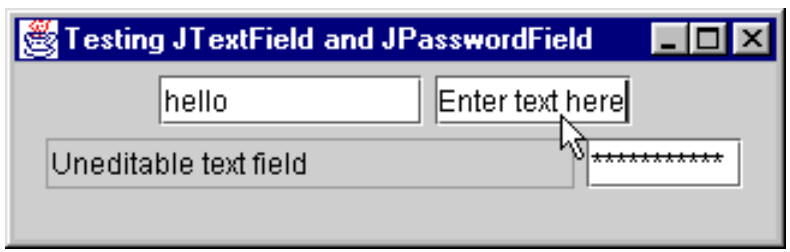
`e.getSource()` returns a **`Component`** reference, which is cast to a **`JPasswordField`**.

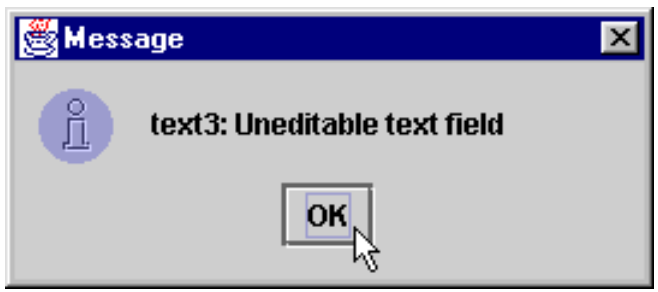


Outline

Program Output







2.5.1 How Event Handling Works

- Registering event listeners
  - All **`JComponents`** contain an object of class **`EventListenerList`** called **`listenerList`**
  - When **`text1.addActionListener( handler )`** executes
    - New entry placed into **`listenerList`**
- Handling events
  - When event occurs, has an event ID
    - Component uses this to decide which method to call
    - If **`ActionEvent`**, then **`actionPerformed`** called (in all registered **`ActionListener`**s)

2.6 JButton

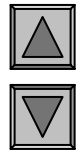
- Button
  - Component user clicks to trigger an action
  - Several types of buttons
    - Command buttons, toggle buttons, check boxes, radio buttons
- Command button
  - Generates **`ActionEvent`** when clicked
  - Created with class **`JButton`**
    - Inherits from class **`AbstractButton`**
    - Defines many features of Swing buttons
- **`JButton`**
  - Text on face called button label
  - Each button should have a different label
  - Can display **`Icon`**s

2.6 JButton

- Methods of class **JButton**
  - Constructors

```
JButton myButton = new JButton( "Label" );
JButton myButton = new JButton( "Label",
  myIcon );
```

  - **setRolloverIcon(myIcon)**
    - Sets image to display when mouse over button
- Class **ActionEvent**
  - **getActionCommand**
    - Returns label of button that generated event



Outline

1. `import`

1.1 Declarations

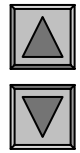
2. Initialize buttons and `Icons`

2.2 Register event handler

```
1  // Fig. 2.11: ButtonTest.java
2  // Creating JButtons.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class ButtonTest extends JFrame {
8     private JButton plainButton, fancyButton;
9
10    public ButtonTest()
11    {
12       super( "Testing Buttons" );
13
14       Container c = getContentPane();
15       c.setLayout( new FlowLayout() );
16
17       // create buttons
18       plainButton = new JButton( "Plain Button" );
19       c.add( plainButton );
20
21       Icon bug1 = new ImageIcon( "bug1.gif" );
22       Icon bug2 = new ImageIcon( "bug2.gif" );
23       fancyButton = new JButton( "Fancy Button", bug1 );
24       fancyButton.setRolloverIcon( bug2 );
25       c.add( fancyButton );
26
27       // create an instance of inner class Butto
28       // to use for button event handling
29       ButtonHandler handler = new ButtonHandler();
30       fancyButton.addActionListener( handler );
```

Create **JButtons**. Initialize **fancyButton** with an **ImageIcon**.

Set a different icon to appear when the mouse is over the **JButton**.



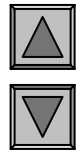
Outline

3. `main`

4. Inner class event handler

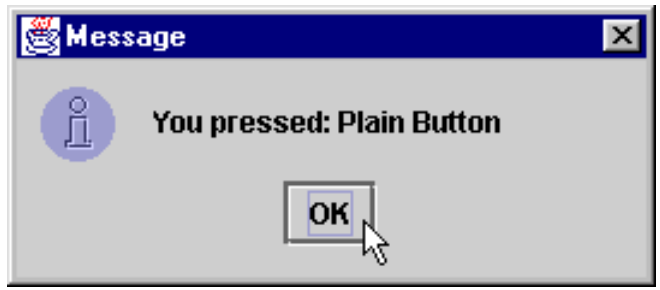
```
31       plainButton.addActionListener( handler );
32
33       setSize( 275, 100 );
34       show();
35    }
36
37    public static void main( String args[] )
38    {
39       ButtonTest app = new ButtonTest();
40
41       app.addWindowListener(
42          new WindowAdapter() {
43             public void windowClosing( WindowEvent e )
44             {
45                System.exit( 0 );
46             }
47          }
48       );
49    }
50
51    // inner class for button event handling
52    private class ButtonHandler implements ActionListener {
53       public void actionPerformed( ActionEvent e )
54       {
55          JOptionPane.showMessageDialog( null,
56             "You pressed: " + e.getActionCommand() );
57       }
58    }
59 }
```

`getActionCommand` returns label of button that generated event.



Outline

Program Output



2.7 JCheckBox and JRadioButton

- State buttons
  - **JToggleButton**
    - Subclasses **JCheckBox**, **JRadioButton**
  - Have on/off (true/false) values

- Class **JCheckBox**
  - Text appears to right of checkbox

  - Constructor

```
JCheckBox myBox = new JCheckBox( "Title" );
```


2.7 JCheckBox and JRadioButton

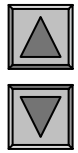
- When **JCheckBox** changes
  - **ItemEvent** generated
    - Handled by an **ItemListener**, which must define **itemStateChanged**
  - Register handlers with with **addItemListener**

```
51    private class CheckBoxHandler implements ItemListener {
55       public void itemStateChanged( ItemEvent e )
56       {
```

- Class **ItemEvent**
  - **getStateChange**
    - Returns **ItemEvent.SELECTED** or **ItemEvent.DESELECTED**

2.7 JCheckBox and JRadioButton

- **JTextField**
  - Method **setText(fontObject)**
    - **new Font(name, style_CONSTANT, size)**
    - **style_CONSTANT - FONT.PLAIN, BOLD, ITALIC**
      - Can add to get combinations
- Example
  - Use **JCheckBox**es to change the font of a **JTextField**



Outline

```
1  // Fig. 2.12: CheckBoxTest.java
2  // Creating Checkbox buttons.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class CheckBoxTest extends JFrame {
8     private JTextField t;
9     private JCheckBox bold, italic;
10
11    public CheckBoxTest()
12    {
13       super( "JCheckBox Test" );
14
15       Container c = getContentPane();
16       c.setLayout(new FlowLayout());
17
18       t = new JTextField( "Watch the font style change", 20 );
19       t.setFont( new Font( "TimesRoman", Font.PLAIN, 14 ) );
20       c.add( t );
21
22       // create checkbox objects
23       bold = new JCheckBox( "Bold" );
24       c.add( bold );
25
26       italic = new JCheckBox( "Italic" );
27       c.add( italic );
28
29       CheckBoxHandler handler = new CheckBoxHandler();
30       bold.addItemListener( handler );
```

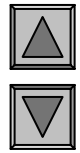
Create **JCheckBox**es

1. `import`

1.1 Declarations

1.2 Initialize JCheckBoxes

1.3 Register event handler



Outline

2. main

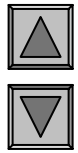
3. Inner class (event handler)

3.1 `getStateChange`

```
31       italic.addItemListener( handler );
32
33       addWindowListener(
34          new WindowAdapter() {
35             public void windowClosing( WindowEvent e )
36             {
37                System.exit( 0 );
38             }
39          }
40       );
41
42       setSize( 275, 100 );
43       show();
44    }
45
46    public static void main( String
47    {
48       new CheckBoxTest();
49    }
50
51    private class CheckBoxHandler implements ItemListen
52       private int valBold = Font.PLAIN;
53       private int valItalic = Font.PLAIN;
54
55       public void itemStateChanged( ItemEvent e )
56       {
57          if ( e.getSource() == bold )
58             if ( e.getStateChange() == ItemEvent.SELECTED )
59                valBold = Font.BOLD;
60             else
61                valBold = Font.PLAIN;
```

Because **`CheckBoxHandler`** implements **`ItemListener`**, it must define method **`itemStateChanged`**

`getStateChange` returns **`ItemEvent.SELECTED`** or **`ItemEvent.DESELECTED`**

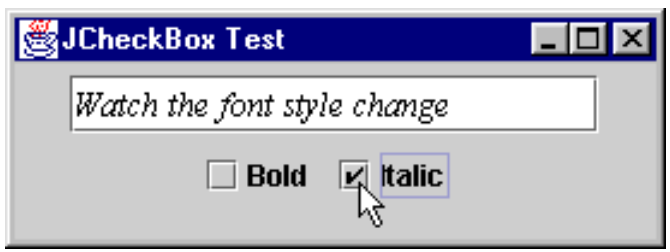


Outline

```
62
63          if ( e.getSource() == italic )
64             if ( e.getStateChange() == ItemEvent.SELECTED )
65                valItalic = Font.ITALIC;
66             else
67                valItalic = Font.PLAIN;
68
69          t.setFont(
70             new Font( "TimesRoman", valBold + valItalic, 14 ) );
71          t.repaint();
72       }
73    }
74 }
```

Use **`setFont`** to update the **`JTextField`**.

Program Output



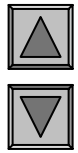
2.7 JCheckBox and JRadioButton

- Radio buttons
  - Have two states: selected and deselected
  - Normally appear as a group
    - Only one radio button in the group can be selected at time
    - Selecting one button forces the other buttons off
  - Mutually exclusive options
  - **`ButtonGroup`** - maintains logical relationship between radio buttons
- Class **`JRadioButton`**
  - Constructor
    - **`JRadioButton( "Label", selected )`**
    - If selected **`true`**, **`JRadioButton`** initially selected

2.7 JCheckBox and JRadioButton

- Class **JRadioButton**
  - Generates **ItemEvents** (like **JCheckBox**)
- Class **ButtonGroup**
  - **ButtonGroup myGroup = new ButtonGroup();**
  - Binds radio buttons into logical relationship
  - Method **add**
    - Associate a radio button with a group

```
myGroup.add( myRadioButton )
```

Outline

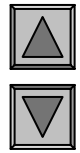
1. `import`

1.1 Declarations

1.2 Initialization

```
1  // Fig. 2.12: RadioButtonTest.java
2  // Creating radio buttons using ButtonGroup and JRadioButton.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class RadioButtonTest extends JFrame {
8     private JTextField t;
9     private Font plainFont, boldFont,
10                 italicFont, boldItalicFont;
11    private JRadioButton plain, bold, italic, boldItalic;
12    private ButtonGroup radioGroup;
13
14    public RadioButtonTest()
15    {
16       super( "RadioButton Test" );
17
18       Container c = getContentPane();
19       c.setLayout( new FlowLayout() );
20
21       t = new JTextField( "Watch the font sty
22       c.add( t );
23
24       // Create radio buttons
25       plain = new JRadioButton( "Plain", true );
26       c.add( plain );
27       bold = new JRadioButton( "Bold", false);
28       c.add( bold );
29       italic = new JRadioButton( "Italic", false );
30       c.add( italic );
```

Initialize radio buttons. Only one is initially selected.



Outline

```
31       boldItalic = new JRadioButton( "Bold/Italic", false );
32       c.add( boldItalic );
33
34       // register events
35       RadioButtonHandler handler = new RadioButtonHandler();
36       plain.addItemListener( handler );
37       bold.addItemListener( handler );
38       italic.addItemListener( handler );
39       boldItalic.addItemListener( handler );
40
41       // create logical relationship between JRadioButtons
42       radioGroup = new ButtonGroup();
43       radioGroup.add( plain );
44       radioGroup.add( bold );
45       radioGroup.add( italic );
46       radioGroup.add( boldItalic );
47
48       plainFont = new Font( "TimesRoman", Font.PLAIN, 14 );
49       boldFont = new Font( "TimesRoman", Font.BOLD, 14 );
50       italicFont = new Font( "TimesRoman", Font.ITALIC, 14 );
51       boldItalicFont =
52          new Font( "TimesRoman", Font.BOLD + Font.ITALIC, 14 );
53       t.setFont( plainFont );
54
55       setSize( 300, 100 );
56       show();
57    }
58
```

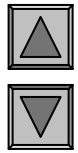
2. Register event

Create a **`ButtonGroup`**. Only one radio button in the group may be selected at a time.

`nGroup`

2.2 add

Method **`add`** adds radio buttons to the **`ButtonGroup`**



Outline

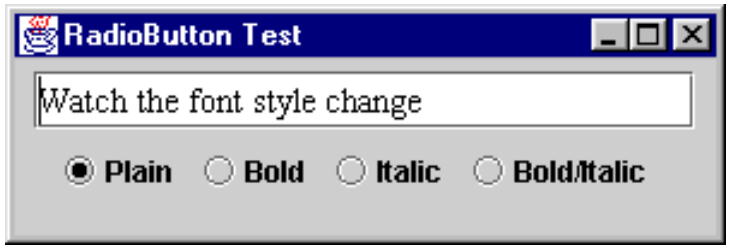
```
59    public static void main( String args[] )
60    {
61       RadioButtonTest app = new RadioButtonTest();
62
63       app.addWindowListener(
64          new WindowAdapter() {
65             public void windowClosing( WindowEvent e )
66             {
67                System.exit( 0 );
68             }
69          }
70       );
71    }
72
73    private class RadioButtonHandler implements ItemListener {
74       public void itemStateChanged( ItemEvent e )
75       {
76          if ( e.getSource() == plain )
77             t.setFont( plainFont );
78          else if ( e.getSource() == bold )
79             t.setFont( boldFont );
80          else if ( e.getSource() == italic )
81             t.setFont( italicFont );
82          else if ( e.getSource() == boldItalic )
83             t.setFont( boldItalicFont );
84
85          t.repaint();
86       }
87    }
88 }
```

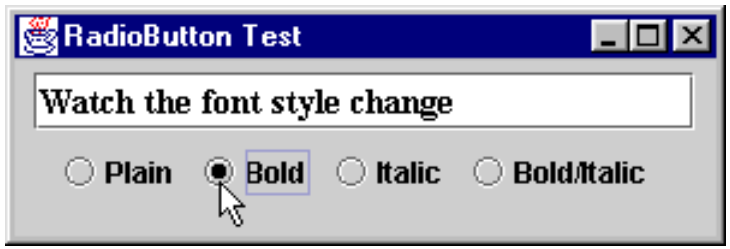
3. `main`

4. Inner class (event handler)

Outline

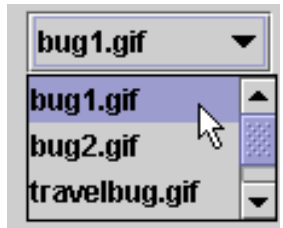
Program Output





2.8 JComboBox

- Combo box (drop down list)
  - List of items, user makes a selection
  - Class **JComboBox**
    - Generate **ItemEvents**

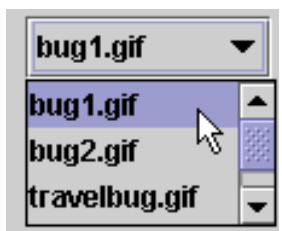


- **JComboBox**
  - Constructor

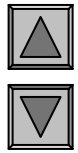
    JComboBox (arrayOfNames)
  - Numeric index keeps track of elements
    - First element added at index **0**
    - First item added is appears as currently selected item when combo box appears

2.8 JComboBox

- **JComboBox** methods
  - **getSelectedIndex**
    - Returns the index of the currently selected item
    - **myComboBox.getSelectedIndex()**
  - **setMaximumRowCount(n)**
    - Set max number of elements to display when user clicks combo box
    - Scrollbar automatically provided
    - **setMaximumRowCount(3)**



- Example
  - Use **JComboBox** to set the **Icon** for a **JLabel**



Outline

```
1  // Fig. 2.13: ComboBoxTest.java
2  // Using a JComboBox to select an image to display.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class ComboBoxTest extends JFrame {
8     private JComboBox images;
9     private JLabel label;
10    private String names[] =
11       { "bug1.gif", "bug2.gif",
12         "travelbug.gif", "buganim.gif" };
13    private Icon icons[] =
14       { new ImageIcon( names[ 0 ] ),
15         new ImageIcon( names[ 1 ] ),
16         new ImageIcon( names[ 2 ] ),
17         new ImageIcon( names[ 3 ] ) };
18
19    public ComboBoxTest()
20    {
21       super( "Testing JComboBox" );
22
23       Container c = getContentPane();
24       c.setLayout( new FlowLayout() );
25
26       images = new JComboBox( names );
27       images.setMaximumRowCount( 3 );
28
29       images.addItemListener(
30          new ItemListener() {
```

Initialize **JComboBox** with an array of **String**s.

Set the number of rows to be displayed at a time.

1. `import`

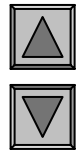
1.1 Initialization

1.2 Constructor

2. Initialize `JComboBox`

2.1 `setMaximumRowCount`

2.2 Register `ItemListener` (anonymous inner class)



Outline

2.3 `getSelectedIndex`

```
31             public void itemStateChanged( ItemEvent e )
32             {
33                label.setIcon(
34                   icons[ images.getSelectedIndex() ] );
35             }
36          }
37       );
38
39       c.add( images );
40
41       label = new JLabel( icons[ 0 ] );
42       c.add( label );
43
44       setSize( 350, 100 );
45       show();
46    }
47
48    public static void main( String args[] )
49    {
50       ComboBoxTest app = new ComboBoxTest();
51
52       app.addWindowListener(
53          new WindowAdapter() {
54             public void windowClosing( WindowEvent e )
55             {
56                System.exit( 0 );
57             }
58          }
59       );
60    }
61 }
```

Use method **`getSelectedIndex`** to determine which **`Icon`** to use.

2.9 JList

- List
  - Displays series of items, may select one or more
  - This section, discuss single-selection lists
- Class **JList**
  - Constructor **JList(arrayOfNames)**
    - Takes array of **Objects** (**String**s) to display in list
  - **setVisibleRowCount(n)**
    - Displays **n** items at a time
    - Does not provide automatic scrolling

2.9 JList

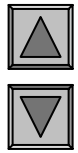
- **JScrollPane** object used for scrolling

```
40        c.add( new JScrollPane( colorList ) );
```

  - Takes component to which to add scrolling as argument
  - Add **JScrollPane** object to content pane
- **JList** methods
  - **setSelectionMode(selection_CONSTANT)**
  - **SINGLE_SELECTION**
    - One item selected at a time
  - **SINGLE_INTERVAL_SELECTION**
    - Multiple selection list, allows contiguous items to be selected
  - **MULTIPLE_INTERVAL_SELECTION**
    - Multiple-selection list, any items can be selected

2.9 JList

- **JList** methods
  - **getSelectedIndex()**
    - Returns index of selected item
- Event handlers
  - Implement interface **ListSelectionListener** (**javax.swing.event**)
  - Define method **valueChanged**
  - Register handler with **addListSelectionListener**
- **Example**
  - Use a **JList** to select the background color



Outline

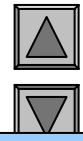
1. import

1.1 Declarations

1.2 Initialize `colorNames` and `colors`

1.3 Constructor

```
1  // Fig. 2.14: ListTest.java
2  // Selecting colors from a JList.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import javax.swing.event.*;
7
8  public class ListTest extends JFrame {
9     private JList colorList;
10    private Container c;
11
12    private String colorNames[] =
13       { "Black", "Blue", "Cyan", "Dark Gray", "Gray", "Green",
14         "Light Gray", "Magenta", "Orange", "Pink", "Red",
15         "White", "Yellow" };
16
17    private Color colors[] =
18       { Color.black, Color.blue, Color.cyan, Color.darkGray,
19         Color.gray, Color.green, Color.lightGray,
20         Color.magenta, Color.orange, Color.pink, Color.red,
21         Color.white, Color.yellow };
22
23    public ListTest()
24    {
25       super( "List Test" );
26
27       c = getContentPane();
28       c.setLayout( new FlowLayout() );
29
```



Outline

```
30       // create a list with the items in the colorNames array
31       colorList = new JList( colorNames );
32       colorList.setVisibleRowCount( 5 );
33
34       // do not allow multiple selections
35       colorList.setSelectionMode(
36          ListSelectionModel.SINGLE_SELECTION );
37
38       // add a JScrollPane containing the JList
39       // to the content pane
40       c.add( new JScrollPane( colorList ) );
41
42       // set up event handler
43       colorList.addListSelectionListener(
44          new ListSelectionListener() {
45             public void valueChanged( ListSelectionEvent e )
46             {
47                c.setBackground(
48                   colors[ colorList.getSelectedIndex() ] );
49             }
50          }
51       );
52
53       setSize( 350, 150 );
54       show();
55    }
56
57    public static void main( String args[] )
58    {
59       ListTest app = new ListTest();
```

Initialize **`JList`** with array of **`String`**s, and show 5 items at a time.

Make the **`JList`** a single-selection list.

Create a new **`JScrollPane`** object, initialize it with a **`JList`**, and attach it to the content pane.

Change the color according to the item selected (use **`getSelectedIndex`**).

2.1

`RowCount`

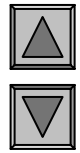
2.2

`Mode`

`ne`

3. Event handler

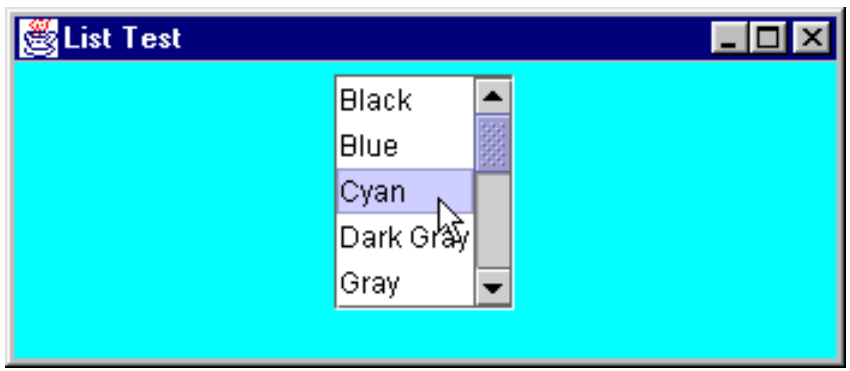
4 main

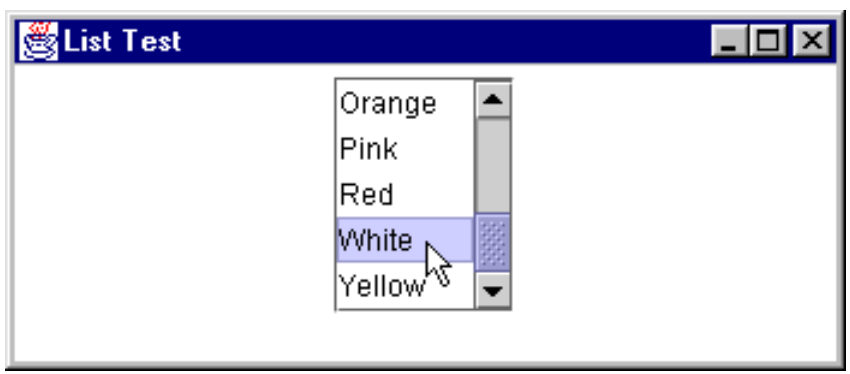


Outline

```
60
61       app.addWindowListener(
62          new WindowAdapter() {
63             public void windowClosing( WindowEvent e )
64             {
65                System.exit( 0 );
66             }
67          }
68       );
69    }
70 }
```

Program Output



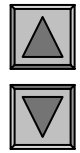


2.10 Multiple-Selection Lists

- Multiple selection lists
  - **`SINGLE_INTERVAL_SELECTION`**
    - Select a contiguous group of items by holding *Shift* key
  - **`MULTIPLE_INTERVAL_SELECTION`**
    - Select any amount of items
    - Hold *Ctrl* key and click each item to select
- **`JList`** methods
  - **`getSelectedValues()`**
    - Returns an array of **`Objects`** representing selected items
  - **`setListData( arrayOfObjects )`**
    - Sets items of **`JList`** to elements in **`arrayOfObjects`**

2.10 Multiple-Selection Lists

- **`JList`** methods
  - **`setFixedCellHeight( height )`**
    - Specifies height in pixels of each item in **`JList`**
  - **`setFixedCellWidth( width )`**
    - As above, set width of list
- Example
  - Have two multiple-selection lists
  - Copy button copies selected items in first list to other list



Outline

```
1  // Fig. 2.15: MultipleSelection.java
2  // Copying items from one List to another.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class MultipleSelection extends JFrame {
8     private JList colorList, copyList;
9     private JButton copy;
10    private String colorNames[] =
11       { "Black", "Blue", "Cyan", "Dark Gray", "Gray",
12         "Green", "Light Gray", "Magenta", "Orange", "Pink",
13         "Red", "White", "Yellow" };
14
15    public MultipleSelection()
16    {
17       super( "Multiple Selection Lists" );
18
19       Container c = getContentPane();
20       c.setLayout( new FlowLayout() );
21
22       colorList = new JList( colorNames );
23       colorList.setVisibleRowCount( 5 );
24       colorList.setFixedCellHeight( 15 );
25       colorList.setSelectionMode(
26          ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
27       c.add( new JScrollPane( colorList ) );
28
```

1. `import`

1.1 Initialize `colorNames`

1.2 Initialize `JList`

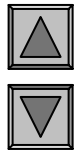
1.3 `setVisibleRowCount`

Initialize the **`JList`** with an array of **`String`**s

Specify the number of items to appear at a time.

Set the cell height

Specify that list is to be **`MULTIPLE_INTERVAL_SELECTION`**



Outline

```
29       // create copy button
30       copy = new JButton( "Copy >>>" );
31       copy.addActionListener(
32          new ActionListener() {
33             public void actionPerformed( ActionEvent e )
34             {
35                // place selected values in copyList
36                copyList.setListData(
37                   colorList.getSelectedValues() );
38             }
39          }
40       );
41       c.add( copy );
42
43       copyList = new JList( );
44       copyList.setVisibleRowCount( 5 );
45       copyList.setFixedCellWidth( 100 );
46       copyList.setFixedCellHeight( 15 );
47       copyList.setSelectionMode(
48          ListSelectionModel.SINGLE_INTERVAL_SELECTION );
49       c.add( new JScrollPane( copyList ) );
50
51       setSize( 300, 120 );
52       show();
53    }
54
```

Use the array returned by **`getSelectedValues`** to set the items of **`copyList`**.

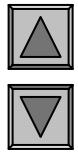
2. `JButton`

2.1 Event handler (anonymous inner class)

`ata`

2.2.1 `getSelectedValues`

2.3 Initialize `JList`

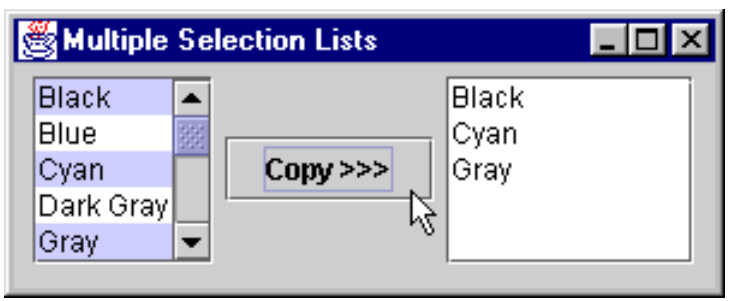


Outline

```
55    public static void main( String args[] )
56    {
57       MultipleSelection app = new MultipleSelection();
58
59       app.addWindowListener(
60          new WindowAdapter() {
61             public void windowClosing( WindowEvent e )
62             {
63                System.exit( 0 );
64             }
65          }
66       );
67    }
68 }
```

3. `main`

Program Output



2.11 Mouse Event Handling

- Mouse events
  - Can be trapped for any GUI component derived from **`java.awt.Component`**
  - Mouse event handling methods
    - Take a **`MouseEvent`** object
      - Contains info about event, including **`x`** and **`y`** coordinates
      - Methods **`getX`** and **`getY`**
  - Interfaces **`MouseListener`** and **`MouseMotionListener`**
    - **`addMouseListener`**
    - **`addMouseMotionListener`**
    - Must define all methods

2.11 Mouse Event Handling

- Interface **`MouseListener`**
  - **`public void mousePressed( MouseEvent e )`**
    - Mouse pressed on a component
  - **`public void mouseClicked( MouseEvent e )`**
    - Mouse pressed and released
  - **`public void mouseReleased( MouseEvent e )`**
    - Mouse released
  - **`public void mouseEntered( MouseEvent e )`**
    - Mouse enters bounds of component
  - **`public void mouseExited( MouseEvent e )`**
    - Mouse leaves bounds of component

2.11 Mouse Event Handling

- Interface **`MouseMotionListener`**
  - **`public void mouseDragged( MouseEvent e )`**
    - Mouse pressed and moved
  - **`public void mouseMoved( MouseEvent e )`**
    - Mouse moved when over component

```
17        getContentPane().add( statusBar, BorderLayout.SOUTH );
```

  - Adds component **`statusBar`** to the bottom portion of the content pane
  - More section 2.14.2

Outline

```
1  // Fig. 2.17: MouseTracker.java
2  // Demonstrating mouse events.
3
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class MouseTracker extends JFrame
9               implements MouseListener, MouseMotionListener {
10    private JLabel statusBar;
11
12    public MouseTracker()
13    {
14       super( "Demonstrating Mouse Events" );
15
16       statusBar = new JLabel();
17       getContentPane().add( statusBar, BorderLayout.SOUTH );
18
19       // application listens to its own mouse events
20       addMouseListener( this );
21       addMouseMotionListener( this );
22
23       setSize( 275, 100 );
24       show();
25    }
26
27    // MouseListener event handlers
28    public void mouseClicked( MouseEvent e )
29    {
```

Class implements interfaces **`MouseListener`** and **`MouseMotionListener`** to listen for mouse events. There are seven methods to define.

Puts the **`JLabel`** component at the bottom of the content pane. More later.

Application is its own event handler

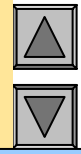
1. Class MouseTracker

er,

Listener)

1.2 Register event handlers (`this`)

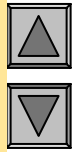
andler



Outline

```
30       statusBar.setText( "Clicked at [" + e.getX() +
31                          ", " + e.getY() + "]" );
32    }
33
34    public void mousePressed( MouseEvent e )
35    {
36       statusBar.setText( "Pressed at [" + e.getX() +
37                          ", " + e.getY() + "]" );
38    }
39
40    public void mouseReleased( MouseEvent e )
41    {
42       statusBar.setText( "Released at [" + e.getX() +
43                          ", " + e.getY() + "]" );
44    }
45
46    public void mouseEntered( MouseEvent e )
47    {
48       statusBar.setText( "Mouse in window" );
49    }
50
51    public void mouseExited( MouseEvent e )
52    {
53       statusBar.setText( "Mouse outside window" );
54    }
55
56    // MouseMotionListener event handlers
57    public void mouseDragged( MouseEvent e )
58    {
59       statusBar.setText( "Dragged at [" + e.getX() +
60                          ", " + e.getY() + "]" );
```

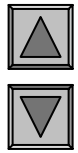
`getX` and **`getY`** return the coordinates of where the mouse event occurred.



Outline

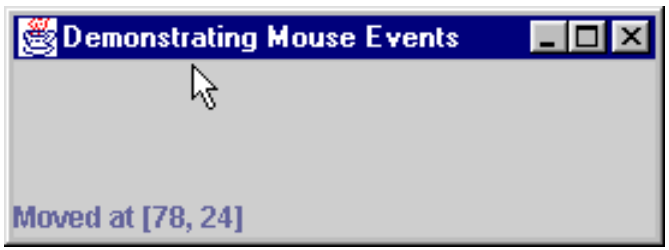
```
61    }
62
63    public void mouseMoved( MouseEvent e )
64    {
65       statusBar.setText( "Moved at [" + e.getX() +
66                          ", " + e.getY() + "]" );
67    }
68
69    public static void main( String args[] )
70    {
71       MouseTracker app = new MouseTracker();
72
73       app.addWindowListener(
74          new WindowAdapter() {
75             public void windowClosing( WindowEvent e )
76             {
77                System.exit( 0 );
78             }
79          }
80       );
81    }
82 }
```

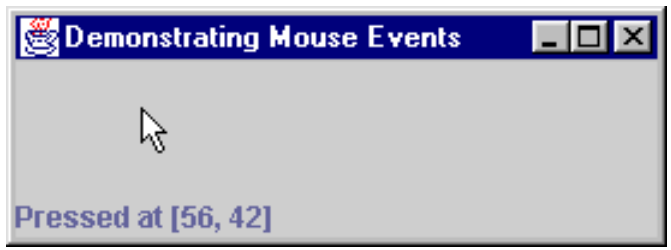
3. main

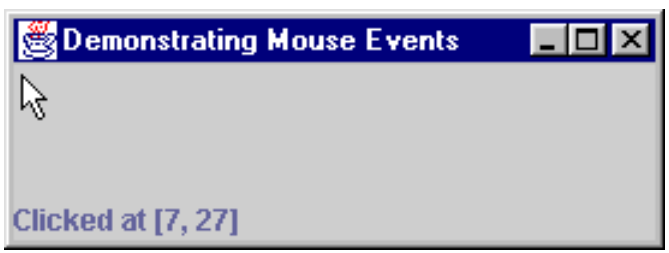


Outline

Program Output







2.12 Adapter Classes

- Time consuming to define all interface methods
  - **`MouseListener`** and **`MouseMotionListener`** have seven methods
    - What if we only want to use one?
    - Required to define all methods in interface
- Adapter class
  - Implements an interface
    - Default implementation (empty body) for all methods
  - Programmer extends adapter class
    - Overrides methods he wants to use
  - Has "is a" relationship with interface
    - **`MouseAdapter`** is a **`MouseListener`**

2.12 Adapter Classes

- Adapter classes

`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

2.12 Adapter Classes

```
18       addMouseMotionListener(
19          new MouseMotionAdapter() {
20             public void mouseDragged( MouseEvent e )
21             {
```

- Anonymous inner class
  - Extends **`MouseMotionAdapter`** (which implements **`MouseMotionListener`**)
  - Inner class gets default (empty body) implementation of **`mouseMoved`** and **`mouseDragged`**
  - Override methods want to use

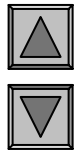
2.12 Adapter Classes

```
40       Painter app = new Painter();
42       app.addWindowListener(
43          new WindowAdapter() {
44             public void windowClosing( WindowEvent e )
45             {
46                System.exit( 0 );
47             }
48          }
49       );
```

- Used in applications extending **`JFrame`**
- Interface **`WindowListener`** specifies seven methods
  - **`WindowAdapter`** define these for us
- Only override the method we want
  - **`windowClosing`**
  - Enables use of close button

2.12 Adapter Classes

- Example program
  - Simple paint program
  - Draw oval whenever user drags mouse
  - Only want to define method **`mouseDragged`**
    - Use **`MouseMotionAdapter`**



Outline

```
1  // Fig. 2.19: Painter.java
2  // Using class MouseMotionAdapter.
3  import javax.swing.*;
4  import java.awt.event.*;
5  import java.awt.*;
6
7  public class Painter extends JFrame {
8     private int xValue = -10, yValue = -10;
9
10    public Painter()
11    {
12       super( "A simple paint program" );
13
14       getContentPane().add(
15          new Label( "Drag the mouse to draw" ),
16          BorderLayout.SOUTH );
17
18       addMouseMotionListener(
19          new MouseMotionAdapter() {
20             public void mouseDragged( MouseEvent e )
21             {
22                xValue = e.getX();
23                yValue = e.getY();
24                repaint();
25             }
26          }
27       );
28
29       setSize( 300, 150 );
30       show();
31    }
```

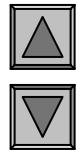
1. **import**

1.1 **addMouseMotion Listener**

1.2. **...apter**

Use adapter class so we do not have to define all the methods of interface **MouseMotionListner**.

Update **xValue** and **yValue**, then call **repaint**.



Outline

```
32
33    public void paint( Graphics g )
34    {
35       g.fillOval( xValue, yValue, 4, 4 );
36    }
37
38    public static void main( String args[] )
39    {
40       Painter app = new Painter();
41
42       app.addWindowListener(
43          new WindowAdapter() {
44             public void windowClosing( WindowEvent e )
45             {
46                System.exit( 0 );
47             }
48          }
49       );
50    }
51 }
```

Draw an oval based at location **xValue**, **yValue**.

3. main

3.1 addWindowListener

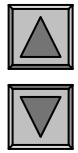
3.2 WindowAdapter

2.12 Adapter Classes

- Class **`MouseEvent`**
  - Inherits from **`InputEvent`**
  - Can distinguish between buttons on multi-button mouse
    - Combination of a mouse click and a keystroke
  - Java assumes every mouse has a left mouse button
    - Alt + click = center mouse button
    - Meta + click = right mouse button
  - Method **`getClickCount`**
    - Returns number of mouse clicks (separate for each button)
  - Methods **`isAltDown`** and **`isMetaDown`**
    - Returns **`true`** if **`Alt`** or **`Meta`** key down when mouse clicked

2.12 Adapter Classes

- Class **`JFrame`**
  - Method **`setTitle( "String" )`**
    - Sets title bar of window



Outline

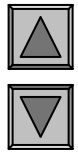
1. `import`

1.1 Constructor

1.2 Register event handler

2. `paint`

```
1  // Fig. 2.20: MouseDetails.java
2  // Demonstrating mouse clicks and
3  // distinguishing between mouse buttons.
4  import javax.swing.*;
5  import java.awt.*;
6  import java.awt.event.*;
7
8  public class MouseDetails extends JFrame {
9     private String s = "";
10    private int xPos, yPos;
11
12    public MouseDetails()
13    {
14       super( "Mouse clicks and buttons" );
15
16       addMouseListener( new MouseClickHandler() );
17
18       setSize( 350, 150 );
19       show();
20    }
21
22    public void paint( Graphics g )
23    {
24       g.drawString( "Clicked @ [" + xPos + ", " + yPos + "]",
25                     xPos, yPos );
26    }
27
```



Outline

```
28    public static void main( String args[] )
29    {
30       MouseDetails app = new MouseDetails();
31
32       app.addWindowListener(
33          new WindowAdapter() {
34             public void windowClosing( WindowEvent e )
35             {
36                System.exit( 0 );
37             }
38          }
39       );
40    }
41
42    // inner class to handle mouse events
43    private class MouseClickHandler extends MouseAdapter {
44       public void mouseClicked( MouseEvent e )
45       {
46          xPos = e.getX();
47          yPos = e.getY();
48
49          String s =
50             "Clicked " + e.getClickCount() + " time(s)";
51
52          if ( e.isMetaDown() )      // Right mouse button
53             s += " with right mouse button";
54          else if ( e.isAltDown() )  // Middle mouse button
55             s += " with center mouse button";
56          else                       // Left mouse button
57             s += " with left mouse button";
58
```

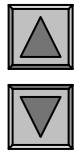
Use a named inner class as the event handler. Can still inherit from MouseAdapter (**`extends MouseAdapter`**).

Use **`getClickCount`**, **`isAltDown`**, and **`isMetaDown`** to determine the **`String`** to use.

3. `main`

4.1 `getClickCount`

4.2 `isMetaDown`



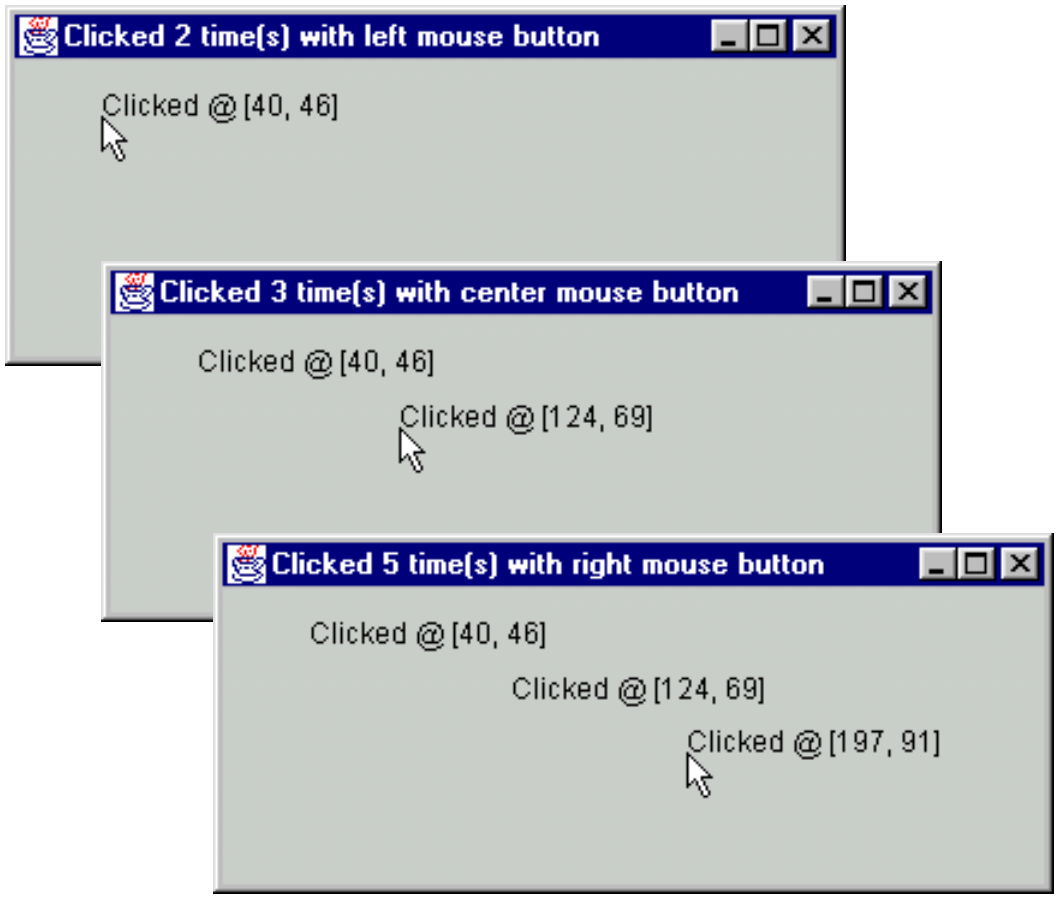
Outline

```
59          setTitle( s );  // set the title bar of the window
60          repaint();
61       }
62    }
63 }
```

Set the title bar.

4.4 `setTitle`

Program Output



2.13 Keyboard Event Handling

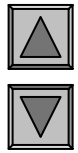
- Interface **`KeyListener`**
  - Handles key events (keys pressed on keyboard)
  - Must define methods
    - **`keyPressed`** - called when any key pressed
    - **`keyTyped`** - called when non-action key pressed
      - Action keys: arrow keys, home, end, page up, page down, function keys, num lock, print screen, scroll lock, caps lock, pause
    - **`keyReleased`** - called for any key after it is released
    - Each get a **`KeyEvent`** as an argument
      - Subclass of **`InputEvent`**

2.13 Keyboard Event Handling

- **`KeyEvent`** methods
  - **`getKeyCode`**
    - Every key represented with a virtual key code (constant)
    - Complete list in on-line documentation (**`java.awt.event`**)
  - **`getKeyText`**
    - Takes key code constant, returns name of key
  - **`getKeyChar`**
    - Gets Unicode character of key pressed
  - **`isActionKey`**
    - Returns **`true`** if key that generated event is an action key

2.13 Keyboard Event Handling

- **`KeyEvent`** methods
  - **`getModifiers`** (from class **`InputEvent`**)
    - Returns which modifiers were pressed
  - **`getKeyModifierText ( e.getModifiers )`**
    - Returns string containing names of modifier keys
- Upcoming example
  - Create a **`JTextArea`**
  - Modify text depending on what keys are pressed



Outline

```
1  // Fig. 2.22: KeyDemo.java
2  // Demonstrating keystroke events.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class KeyDemo extends JFrame implements KeyListener {
8     private String line1 = "", line2 = "";
9     private String line3 = "";
10    private JTextArea textArea;
11
12    public KeyDemo()
13    {
14       super( "Demonstrating Keystroke Events" );
15
16       textArea = new JTextArea( 10, 15 );
17       textArea.setText( "Press any key on the keyboard..." );
18       textArea.setEnabled( false );
19
20       // allow frame to process Key events
21       addKeyListener( this );
22
23       getContentPane().add( textArea );
24
25       setSize( 350, 100 );
26       show();
27    }
28
```

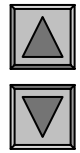
Class implements interface **`KeyListener`**, so it must define the three required methods.

Register the event handler.

1. import

1.1 Class `KeyDemo` (implements

1.2 `addKeyListener`



Outline

```
29    public void keyPressed( KeyEvent e )
30    {
31       line1 = "Key pressed: " +
32               e.getKeyText( e.getKeyCode() );
33       setLines2and3( e );
34    }
35
36    public void keyReleased( KeyEvent
37    {
38       line1 = "Key released: " +
39               e.getKeyText( e.getKeyCode() );
40       setLines2and3( e );
41    }
42
43    public void keyTyped( KeyEvent e )
44    {
45       line1 = "Key typed: " + e.getKeyChar();
46       setLines2and3( e );
47    }
48
49    private void setLines2and3( KeyEvent e )
50    {
51       line2 = "This key is " +
52               ( e.isActionKey() ? "" : "not " ) +
53               "an action key";
54
55       String temp =
56          e.getKeyModifiersText( e.getModifiers() );
57
58       line3 = "Modifier keys pressed: " +
59               ( temp.equals( "" ) ? "none" : temp );
60
```

`getKeyCode` returns the virtual key code. **`getKeyText`** converts the key code to a **`String`** containing the name.

Test if the key is an action key

`getModifiers` returns the modifier keys, and **`getKeyModifersText`** turns them into a **`String`**.

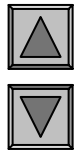
andling

2.1 `getKeyText`

2.2 `getKeyCode`

2.3 `isActionKey`

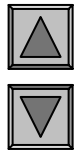
2.4 Determine modifier



Outline

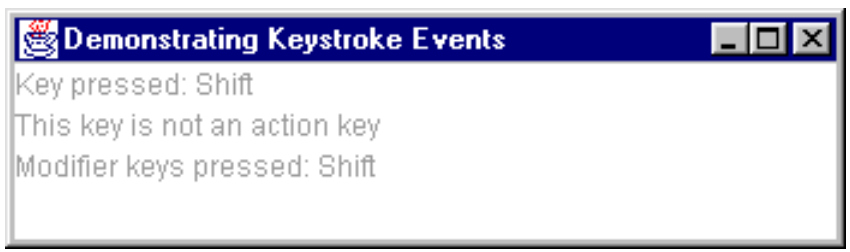
```
61          textArea.setText(
62             line1 + "\n" + line2 + "\n" + line3 + "\n" );
63       }
64
65       public static void main( String args[] )
66       {
67          KeyDemo app = new KeyDemo();
68
69          app.addWindowListener(
70             new WindowAdapter() {
71                public void windowClosing( WindowEvent e )
72                {
73                   System.exit( 0 );
74                }
75             }
76          );
77       }
78    }
```

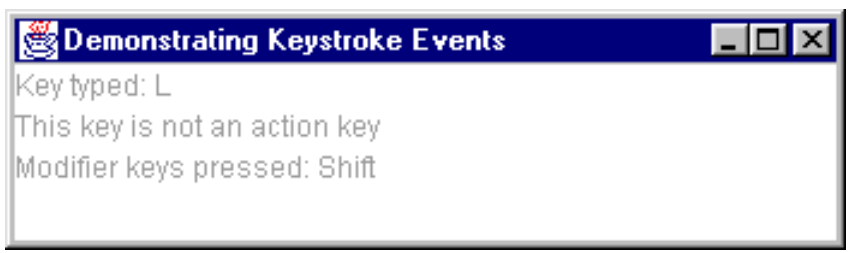
3. main

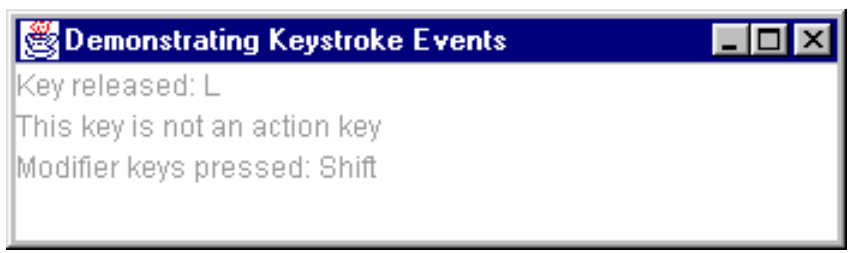


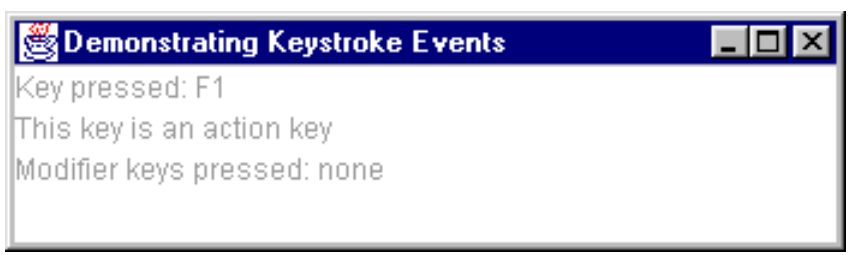
Outline

Program Output







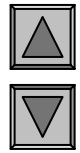


2.14 Layout Managers

- Layout managers
  - Arrange GUI components on a container
  - Provide basic layout capabilities
    - Easier to use than determining exact size and position of every component
    - Programmer concentrates on "look and feel" rather than details

2.14.1 `FlowLayout`

- Most basic layout manager
  - Components placed left to right in order added
  - When edge of container reached, continues on next line
  - Components can be left-aligned, centered (default), or right-aligned
- **`FlowLayout`** methods
  - **`setAlignment( position_CONSTANT )`**
    - **`FlowLayout.LEFT`, `FlowLayout.CENTER`, `FlowLayout.RIGHT`**
  - **`layoutContainer( container )`**
    - Update **`Container`** specified with layout
      - I.e., content pane



Outline

```
1  // Fig. 12.24: FlowLayoutDemo.java
2  // Demonstrating FlowLayout alignments.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class FlowLayoutDemo extends JFrame {
8     private JButton left, center, right;
9     private Container c;
10    private FlowLayout layout;
11
12    public FlowLayoutDemo()
13    {
14       super( "FlowLayout Demo" );
15
16       layout = new FlowLayout();
17
18       c = getContentPane();
19       c.setLayout( layout );
20
21       left = new JButton( "Left" );
22       left.addActionListener(
23          new ActionListener() {
24             public void actionPerformed( ActionEvent e )
25             {
26                layout.setAlignment( FlowLayout.LEFT );
27
28                // re-align attached components
29                layout.layoutContainer( c );
30             }
```

`setAlignment` changes the alignment of the layout.

Use method **`layoutContainer`** to update changes

1. `import`

1.1 Declarations

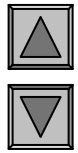
1.2 Initialize `FlowLayout`

reate button

1.4 Event handler

1.4.1 setAlignment

1.4.2 `layoutContainer`



Outline

```
31          }
32       );
33       c.add( left );
34
35       center = new JButton( "Center" );
36       center.addActionListener(
37          new ActionListener() {
38             public void actionPerformed( ActionEvent e )
39             {
40                layout.setAlignment( FlowLayout.CENTER );
41
42                // re-align attached components
43                layout.layoutContainer( c );
44             }
45          }
46       );
47       c.add( center );
48
49       right = new JButton( "Right" );
50       right.addActionListener(
51          new ActionListener() {
52             public void actionPerformed( ActionEvent e )
53             {
54                layout.setAlignment( FlowLayout.RIGHT );
55
56                // re-align attached components
57                layout.layoutContainer( c );
58             }
59          }
60       );
```

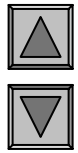
1.5 add JButton

2. JButton

2.1 Event handler

3. JButton

3.1 Event handler

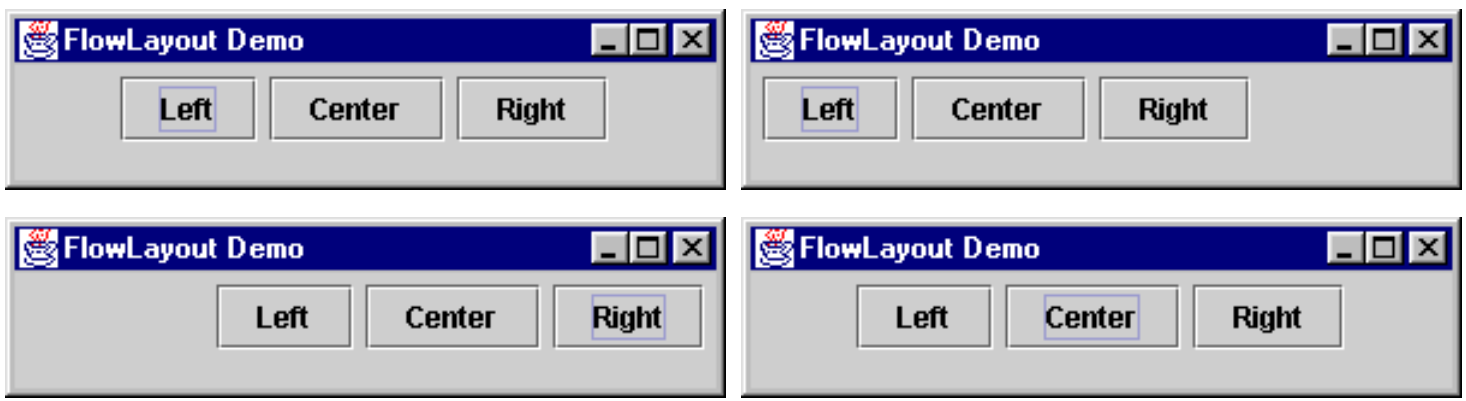


Outline

```
61       c.add( right );
62
63       setSize( 300, 75 );
64       show();
65    }
66
67    public static void main( String args[] )
68    {
69       FlowLayoutDemo app = new FlowLayoutDemo();
70
71       app.addWindowListener(
72          new WindowAdapter() {
73             public void windowClosing( WindowEvent e )
74             {
75                System.exit( 0 );
76             }
77          }
78       );
79    }
80 }
```

4. `main`

Program Output



2.14.2 BorderLayout

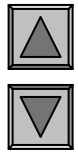
- **BorderLayout**
  - Default manager for content pane
  - Arrange components into 5 regions
    - North, south, east, west, center
  - Up to 5 components can be added directly
    - One for each region
  - Components placed in
    - North/South - Region is as tall as component
    - East/West - Region is as wide as component
    - Center - Region expands to take all remaining space

2.14.2 BorderLayout

- Methods
  - Constructor: **`BorderLayout( hGap, vGap );`**
    - **`hGap`** - horizontal gap space between regions
    - **`vGap`** - vertical gap space between regions
    - Default is **`0`** for both
  - Adding components
    - **`myContainer.add( component, position )`**
    - **`component`** - component to add
    - **`position`** - **`BorderLayout.NORTH`**
      - **`SOUTH, EAST, WEST, CENTER`** similar

2.14.2 BorderLayout

- Methods
  - **`setVisible( boolean )`** (in class **`JButton`**)
    - If **`false`**, hides component
  - **`layoutContainer( container )`** - updates container, as before



Outline

1. import

1.1 Declarations

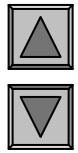
1.2 Initialize layout

1.3 Create JButtons

handler

```
1  // Fig. 12.25: BorderLayoutDemo.java
2  // Demonstrating BorderLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class BorderLayoutDemo extends JFrame
8                              implements ActionListener {
9     private JButton b[];
10    private String names[] =
11       { "Hide North", "Hide South", "Hide East",
12         "Hide West", "Hide Center" };
13    private BorderLayout layout;
14
15    public BorderLayoutDemo()
16    {
17       super( "BorderLayout Demo" );
18
19       layout = new BorderLayout( 5, 5 );
20
21       Container c = getContentPane();
22       c.setLayout( layout );
23
24       // instantiate button objects
25       b = new JButton[ names.length ];
26
27       for ( int i = 0; i < names.length; i++ ) {
28          b[ i ] = new JButton( names[ i ] );
29          b[ i ].addActionListener( this );
30       }
```

Set horizontal and vertical spacing in constructor.

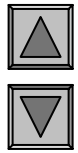


Outline


```
31
32       // order not important
33       c.add( b[ 0 ], BorderLayout.NORTH );  // North position
34       c.add( b[ 1 ], BorderLayout.SOUTH );  // South position
35       c.add( b[ 2 ], BorderLayout.EAST );   // East position
36       c.add( b[ 3 ], BorderLayout.WEST );   // West position
37       c.add( b[ 4 ], BorderLayout.CENTER ); // Center position
38
39       setSize( 300, 200 );
40       show();
41    }
42
43    public void actionPerformed( ActionEvent e )
44    {
45       for ( int i = 0; i < b.length; i++ )
46          if ( e.getSource() == b[ i ] )
47             b[ i ].setVisible( false );
48          else
49             b[ i ].setVisible( true );
50
51       // re-layout the content pane
52       layout.layoutContainer( getContentPane() );
53    }
54
55    public static void main( String args[] )
56    {
57       BorderLayoutDemo app = new BorderLayoutDemo();
58
59       app.addWindowListener(
60          new WindowAdapter() {
```

Hide the button that generated the event.

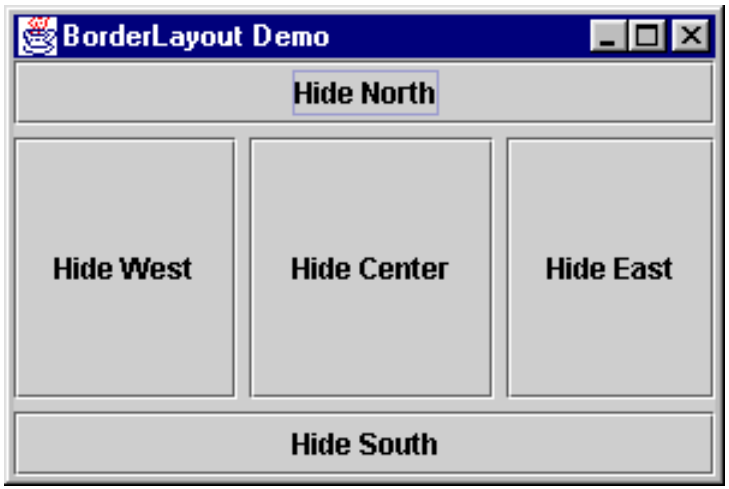
Recalculates layout of content pane.

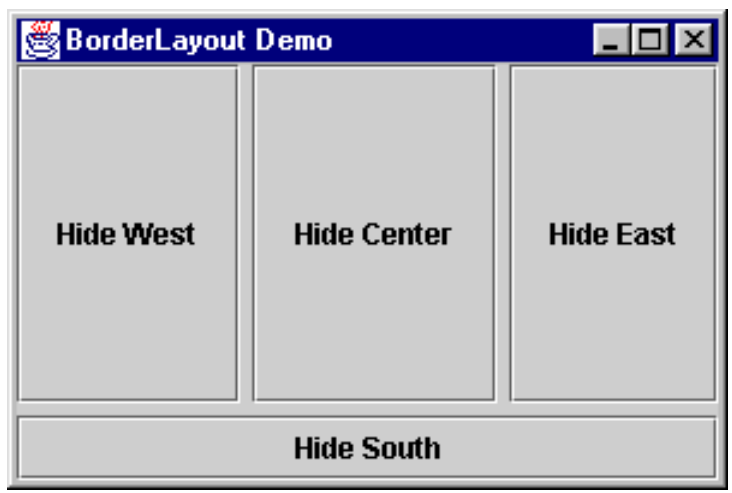


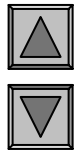
Outline

```
61             public void windowClosing( WindowEvent e )
62             {
63                System.exit( 0 );
64             }
65          }
66       );
67    }
68 }
```

Program Output

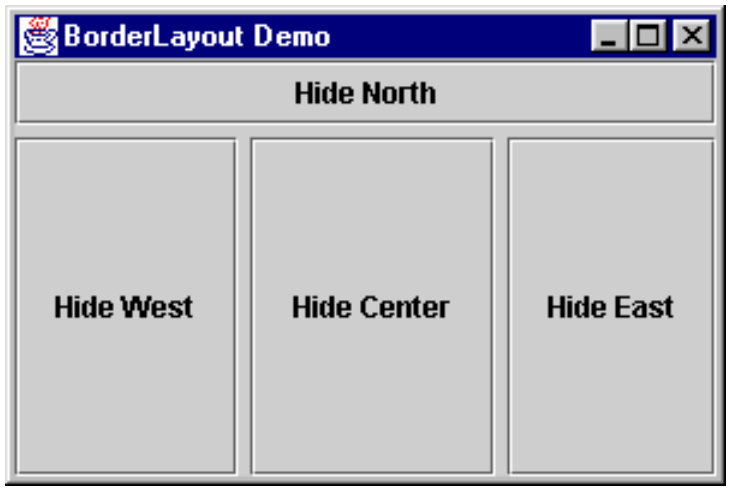


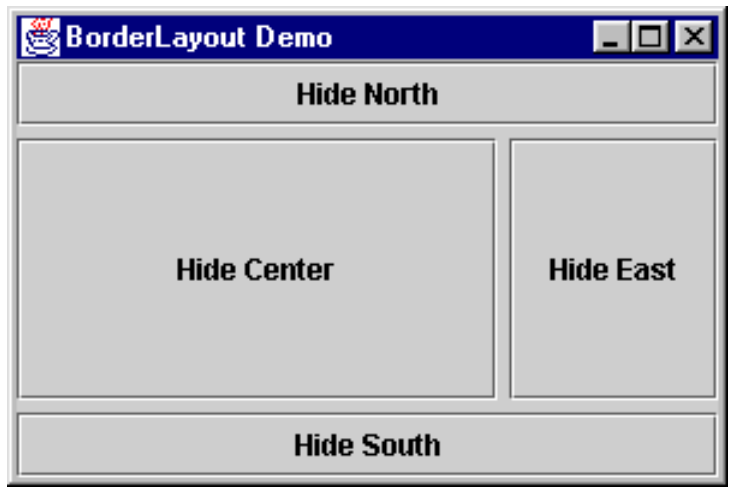


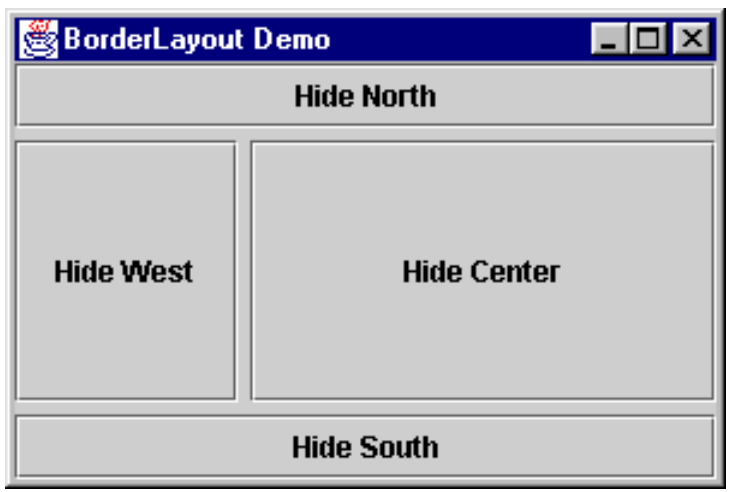


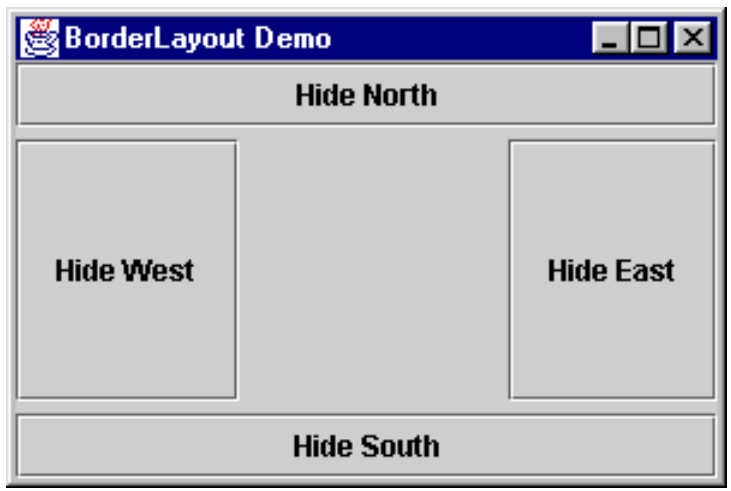
Outline

Program Output









2.11.3 GridLayout

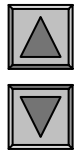
- **GridLayout**
  - Divides container into a grid
  - Components placed in rows and columns
  - All components have same width and height
    - Added starting from top left, then from left to right
    - When row full, continues on next row, left to right
- Constructors
  - **GridLayout(rows, columns, hGap, vGap)**
    - Specify number of rows and columns, and horizontal and vertical gaps between elements (in pixels)
  - **GridLayout(rows, columns)**
    - Default **0** for **hGap** and **vGap**

2.11.3 GridLayout

- Updating containers
  - **Container** method **validate**
    - Re-layouts a container for which the layout has changed
  - Example:

```
Container c = getContentPane;
c.setLayout( myLayout );
if ( x = 3 ){
  c.setLayout( myLayout2 );
  c.validate();
}
```

    - Changes layout and updates **c** if condition met



Outline

1. `import`

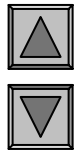
1.1 Declarations

1.2 Initialize `layout`

1.3 Register event

```
1  // Fig. 2.26: GridLayoutDemo.java
2  // Demonstrating GridLayout.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class GridLayoutDemo extends JFrame
8                              implements ActionListener {
9     private JButton b[];
10    private String names[] =
11       { "one", "two", "three", "four", "five", "six" };
12    private boolean toggle = true;
13    private Container c;
14    private GridLayout grid1, grid2;
15
16    public GridLayoutDemo()
17    {
18       super( "GridLayout Demo" );
19
20       grid1 = new GridLayout( 2, 3, 5, 5 );
21       grid2 = new GridLayout( 3, 2 );
22
23       c = getContentPane();
24       c.setLayout( grid1 );
25
26       // create and add buttons
27       b = new JButton[ names.length ];
28
29       for (int i = 0; i < names.length; i++ ) {
30          b[ i ] = new JButton( names[ i ] );
31          b[ i ].addActionListener( this );
```

Create two **`GridLayout`**s, a 2 by 3 and a 3 by 2 (rows, columns).



Outline

1.4 add

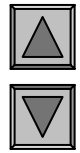
2. actionPerformed

3. main

```
32          c.add( b[ i ] );
33       }
34
35       setSize( 300, 150 );
36       show();
37    }
38
39    public void actionPerformed( ActionEvent e )
40    {
41       if ( toggle )
42          c.setLayout( grid2 );
43       else
44          c.setLayout( grid1 );
45
46       toggle = !toggle;
47       c.validate();
48    }
49
50    public static void main( String args[] )
51    {
52       GridLayoutDemo app = new GridLayoutDemo();
53
54       app.addWindowListener(
55          new WindowAdapter() {
56             public void windowClosing( WindowEvent e )
57             {
58                System.exit( 0 );
59             }
60          }
61       );
62    }
63 }
```

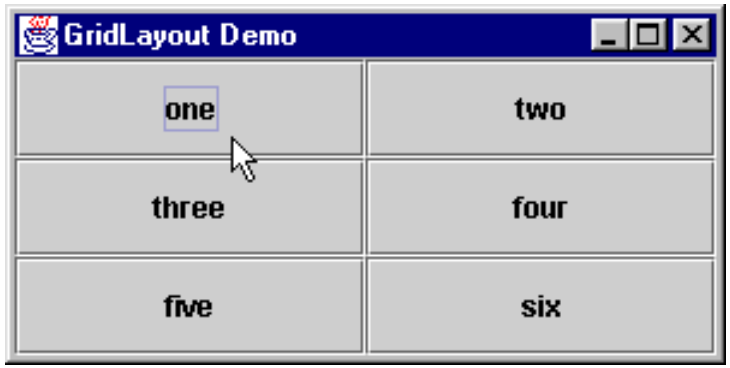
Add buttons to layout. Added from left to right in order.

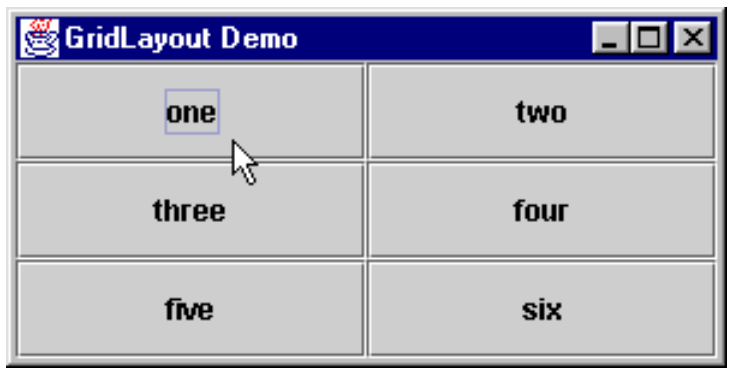
Toggle layouts and update content pane with **validate**.



Outline

Program Output



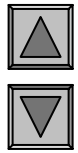


2.12 Panels

- Complex GUIs
  - Each component needs to be placed in an exact location
  - Can use multiple panels
    - Each panel's components arranged in a specific layout
- Panels
  - Class **`JPanel`** inherits from **`JComponent`**, which inherits from **`java.awt.Container`**
    - Every **`JPanel`** is a **`Container`**
  - **`JPanels`** can have components (and other **`JPanels`**) added to them
    - **`JPanel`** sized to components it contains
    - Grows to accomodate components as they are added

2.12 Panels

- Usage
  - Create panels, and set the layout for each
  - Add components to the panels as needed
  - Add the panels to the content pane (default **`BorderLayout`**)



Outline

```
1  // Fig. 2.27: PanelDemo.java
2  // Using a JPanel to help lay out components.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class PanelDemo extends JFrame {
8     private JPanel buttonPanel;
9     private JButton buttons[];
10
11    public PanelDemo()
12    {
13       super( "Panel Demo" );
14
15       Container c = getContentPane();
16       buttonPanel = new JPanel();
17       buttons = new JButton[ 5 ];
18
19       buttonPanel.setLayout(
20          new GridLayout( 1, buttons.length ) );
21
22       for ( int i = 0; i < buttons.length; i++ ) {
23          buttons[ i ] = new JButton( "Button " + (i + 1) );
24          buttonPanel.add( buttons[ i ] );
25       }
26
27       c.add( buttonPanel, BorderLayout.SOUTH );
28
29       setSize( 425, 150 );
30       show();
31    }
```

Create a new panel.

Add components to panel.

Add panel to the content pane (**BorderLayout.SOUTH**).

1. **import**

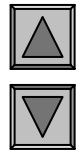
1.1 Declarations

1.2 Initialize **buttonPanel**

GridLayout

1.4 **ButtonPanel.add**

1.5 **c.add**



Outline

```
32
33    public static void main( String args[] )
34    {
35       PanelDemo app = new PanelDemo();
36
37       app.addWindowListener(
38          new WindowAdapter() {
39             public void windowClosing( WindowEvent e )
40             {
41                System.exit( 0 );
42             }
43          }
44       );
45    }
46 }
```

2. `main`

Program Output

JPanel sized to its components. Grows as needed.