```
1  // Fig. 13.1: TextAreaDemo.java
2  // Copying selected text from one text area to another.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class TextAreaDemo extends JFrame {
8     private JTextArea t1, t2;
9     private JButton copy;
10
11    public TextAreaDemo()
12    {
13       super( "TextArea Demo" );
14
15       Box b = Box.createHorizontalBox();
16
17       String s = "This is a demo string to\n" +
18                  "illustrate copying text\n" +
19                  "from one TextArea to \n" +
20                  "another TextArea using an\n"+
21                  "external event\n";
22
23       t1 = new JTextArea( s, 10, 15 );
24       b.add( new JScrollPane( t1 ) );
25
26       copy = new JButton( "Copy >>>" );
27       copy.addActionListener(
28          new ActionListener() {
29             public void actionPerformed( ActionEvent e )
30             {
31                t2.setText( t1.getSelectedText() );
32             }
33          }
34       );
35       b.add( copy );
36
37       t2 = new JTextArea( 10, 15 );
38       t2.setEditable( false );
39       b.add( new JScrollPane( t2 ) );
40
41       Container c = getContentPane();
42       c.add( b );    // Box placed in BorderLayout.CENTER
43       setSize( 425, 200 );
44       show();
45    }
46
47    public static void main( String args[] )
48    {
49       TextAreaDemo app = new TextAreaDemo();
50
51       app.addWindowListener(
52          new WindowAdapter() {
53             public void windowClosing( WindowEvent e )
54             {
55                System.exit( 0 );
```

```
56                  }
57               }
58            );
59         }
60      }
```

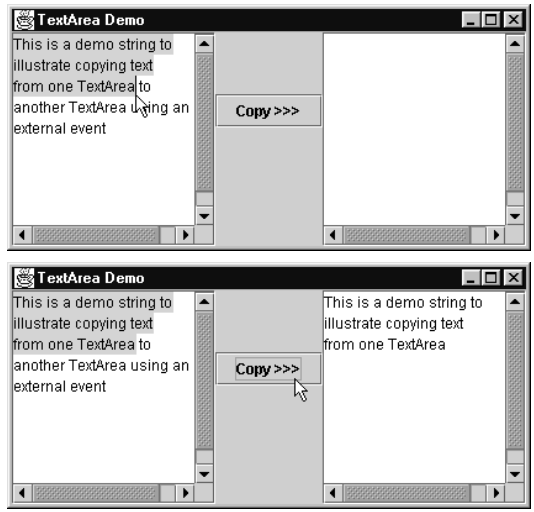


Fig. 13.1 Copying selected text from one text area to another.

```
1  // Fig. 13.2: CustomPanel.java
2  // A customized JPanel class.
3  import java.awt.*;
4  import javax.swing.*;
5
6  public class CustomPanel extends JPanel {
7     public final static int CIRCLE = 1, SQUARE = 2;
8     private int shape;
9
10    public void paintComponent( Graphics g )
11    {
12       super.paintComponent( g );
13
14       if ( shape == CIRCLE )
15          g.fillOval( 50, 10, 60, 60 );
16       else if ( shape == SQUARE )
17          g.fillRect( 50, 10, 60, 60 );
18    }
19
20    public void draw( int s )
21    {
22       shape = s;
23       repaint();
24    }
25 }
```

Fig. 13.2 Extending class **`JPanel`** (part 1 of 3).

```
26 // Fig. 13.2: CustomPanelTest.java
27 // Using a customized Panel object.
28 import java.awt.*;
29 import java.awt.event.*;
30 import javax.swing.*;
31
32 public class CustomPanelTest extends JFrame {
33    private JPanel buttonPanel;
34    private CustomPanel myPanel;
35    private JButton circle, square;
36
37    public CustomPanelTest()
38    {
39       super( "CustomPanel Test" );
40
41       myPanel = new CustomPanel();   // instantiate canvas
42       myPanel.setBackground( Color.green );
43
44       square = new JButton( "Square" );
45       square.addActionListener(
46          new ActionListener() {
47             public void actionPerformed( ActionEvent e )
48             {
49                myPanel.draw( CustomPanel.SQUARE );
```

Fig. 13.2 Extending class **`JPanel`** (part 2 of 3).

```
50                }
51             }
52          );
53
54          circle = new JButton( "Circle" );
55          circle.addActionListener(
56             new ActionListener() {
57                public void actionPerformed( ActionEvent e )
58                {
59                   myPanel.draw( CustomPanel.CIRCLE );
60                }
61             }
62          );
63
64          buttonPanel = new JPanel();
65          buttonPanel.setLayout( new GridLayout( 1, 2 ) );
66          buttonPanel.add( circle );
67          buttonPanel.add( square );
68
69          Container c = getContentPane();
70          c.add( myPanel, BorderLayout.CENTER );
71          c.add( buttonPanel, BorderLayout.SOUTH );
72
73          setSize( 300, 150 );
74          show();
75       }
76
77       public static void main( String args[] )
78       {
79          CustomPanelTest app = new CustomPanelTest();
80
81          app.addWindowListener(
82             new WindowAdapter() {
83                public void windowClosing( WindowEvent e )
84                {
85                   System.exit( 0 );
86                }
87             }
88          );
89       }
90    }
```

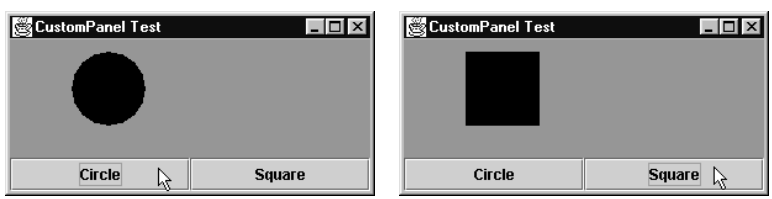


Fig. 13.2 Extending class **`JPanel`** (part 3 of 3).

```
1  // Fig. 13.3: SelfContainedPanelTest.java
2  // Creating a self-contained subclass of JPanel
3  // that processes its own mouse events.
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7  import com.deitel.jhtp3.ch13.SelfContainedPanel;
8
9  public class SelfContainedPanelTest extends JFrame {
10    private SelfContainedPanel myPanel;
11
12    public SelfContainedPanelTest()
13    {
14       myPanel = new SelfContainedPanel();
15       myPanel.setBackground( Color.yellow );
16
17       Container c = getContentPane();
18       c.setLayout( new FlowLayout() );
19       c.add( myPanel );
20
21       addMouseMotionListener(
22          new MouseMotionListener() {
23             public void mouseDragged( MouseEvent e )
24             {
25                setTitle( "Dragging: x=" + e.getX() +
26                          "; y=" + e.getY() );
27             }
28
29             public void mouseMoved( MouseEvent e )
30             {
31                setTitle( "Moving: x=" + e.getX() +
32                          "; y=" + e.getY() );
33             }
34          }
35       );
36
37       setSize( 300, 200 );
38       show();
39    }
40
41    public static void main( String args[] )
42    {
43       SelfContainedPanelTest app =
44          new SelfContainedPanelTest();
45
```

```
46          app.addWindowListener(
47             new WindowAdapter() {
48                public void windowClosing( WindowEvent e )
49                {
50                   System.exit( 0 );
51                }
52             }
53          );
54       }
55    }
```

Fig. 13.3 Capturing mouse events with a **`JPanel`**.

```
56    // Fig. 13.3: SelfConainedPanel.java
57    // A self-contained JPanel class that
58    // handles its own mouse events.
59    package com.deitel.jhtp3.ch13;
60
61    import java.awt.*;
62    import java.awt.event.*;
63    import javax.swing.*;
64
65    public class SelfContainedPanel extends JPanel {
66       private int x1, y1, x2, y2;
67
68       public SelfContainedPanel()
69       {
70          addMouseListener(
71             new MouseAdapter() {
72                public void mousePressed( MouseEvent e )
73                {
74                   x1 = e.getX();
75                   y1 = e.getY();
76                }
77
78                public void mouseReleased( MouseEvent e )
79                {
80                   x2 = e.getX();
81                   y2 = e.getY();
82                   repaint();
83                }
84             }
85          );
86
87          addMouseMotionListener(
88             new MouseMotionAdapter() {
89                public void mouseDragged( MouseEvent e )
90                {
91                   x2 = e.getX();
92                   y2 = e.getY();
93                   repaint();
94                }
95             }
96          );
97       }
```

```
98
99       public Dimension getPreferredSize()
100      {
101         return new Dimension( 150, 100 );
102      }
103
104      public void paintComponent( Graphics g )
105      {
106         super.paintComponent( g );
107
108         g.drawOval( Math.min( x1, x2 ), Math.min( y1, y2 ),
109                     Math.abs( x1 - x2 ), Math.abs( y1 - y2 ) );
110      }
111   }
```

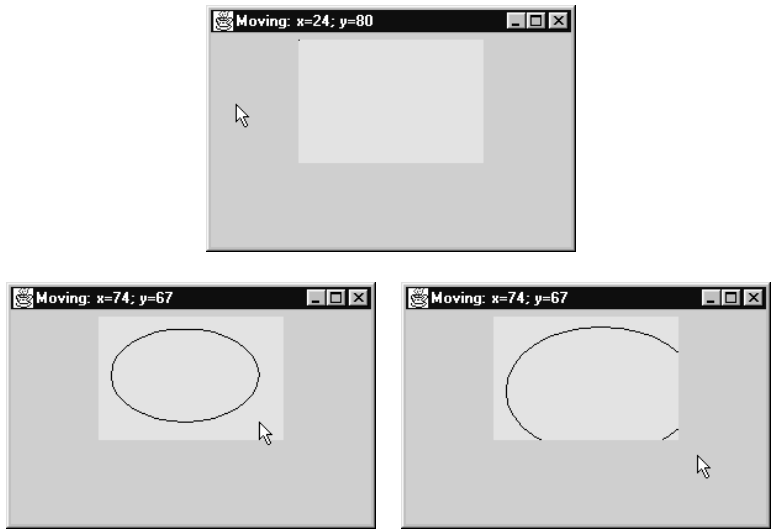


Fig. 13.3 Capturing mouse events with a **`JPanel`**

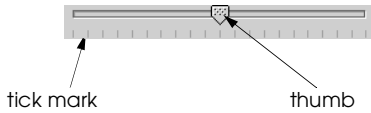


Fig. 13.4 A horizontal **`JSlider`** component.

```
1  // Fig. 13.5: SliderDemo.java
2  // Using JSliders to size an oval.
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6  import javax.swing.event.*;
7
8  public class SliderDemo extends JFrame {
9     private JSlider diameter;
10    private OvalPanel myPanel;
11
12    public SliderDemo()
13    {
14       super( "Slider Demo" );
15
16       myPanel = new OvalPanel();
17       myPanel.setBackground( Color.yellow );
18
19       diameter = new JSlider( SwingConstants.HORIZONTAL,
20                               0, 200, 10 );
21       diameter.setMajorTickSpacing( 10 );
22       diameter.setPaintTicks( true );
23       diameter.addChangeListener(
24          new ChangeListener() {
25             public void stateChanged( ChangeEvent e )
26             {
27                myPanel.setDiameter( diameter.getValue() );
28             }
29          }
30       );
31
32       Container c = getContentPane();
33       c.add( diameter, BorderLayout.SOUTH );
34       c.add( myPanel, BorderLayout.CENTER );
35
36       setSize( 220, 270 );
37       show();
38    }
39
40    public static void main( String args[] )
41    {
42       SliderDemo app = new SliderDemo();
43
44       app.addWindowListener(
45          new WindowAdapter() {
46             public void windowClosing( WindowEvent e )
47             {
48                System.exit( 0 );
49             }
50          }
51       );
52    }
53 }
```

Fig. 13.5 Using a **`JSlider`** to determine the diameter of a circle.

```
54 // Fig. 13.5: OvalPanel.java
55 // A customized JPanel class.
56 import java.awt.*;
57 import javax.swing.*;
58
59 public class OvalPanel extends JPanel {
60    private int diameter = 10;
61
62    public void paintComponent( Graphics g )
63    {
64       super.paintComponent( g );
65       g.fillOval( 10, 10, diameter, diameter );
66    }
67
68    public void setDiameter( int d )
69    {
70       diameter = ( d >= 0 ? d : 10 );  // default diameter 10
71       repaint();
72    }
73
74    // the following methods are used by layout managers
75    public Dimension getPreferredSize()
76    {
77       return new Dimension( 200, 200 );
78    }
79
80    public Dimension getMinimumSize()
81    {
82       return getPreferredSize();
83    }
84 }
```

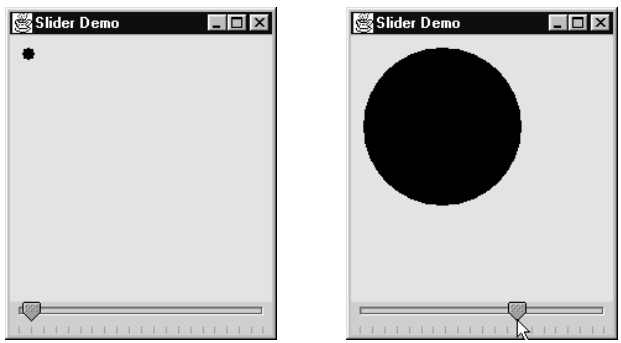


Fig. 13.5 Using a `JSlider` to determine the diameter of a circle.

```
1  // Fig. 13.6: DrawShapes.java
2  // Draw random lines, rectangles and ovals
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class DrawShapes extends JApplet {
8     private JButton choices[];
9     private String names[] = { "Line", "Rectangle", "Oval" };
10    private JPanel buttonPanel;
11    private DrawPanel drawingArea;
12    private int width = 300, height = 200;
13
14    public void init()
15    {
16       drawingArea = new DrawPanel( width, height );
17
18       choices = new JButton[ names.length ];
19       buttonPanel = new JPanel();
20       buttonPanel.setLayout(
21          new GridLayout( 1, choices.length ) );
22       ButtonHandler handler = new ButtonHandler();
23
24       for ( int i = 0; i < choices.length; i++ ) {
25          choices[ i ] = new JButton( names[ i ] );
26          buttonPanel.add( choices[ i ] );
27          choices[ i ].addActionListener( handler );
28       }
```

Fig. 13.6 Creating a GUI-based application from an applet (part 1 of 4).

```
29
30       Container c = getContentPane();
31       c.add( buttonPanel, BorderLayout.NORTH );
32       c.add( drawingArea, BorderLayout.CENTER );
33    }
34
35    public void setWidth( int w )
36       { width = ( w >= 0 ? w : 300 ); }
37
38    public void setHeight( int h )
39       { height = ( h >= 0 ? h : 200 ); }
40
41    public static void main( String args[] )
42    {
43       int width, height;
44
45       if ( args.length != 2 ) {  // no command-line arguments
46          width = 300;
47          height = 200;
48       }
49       else {
50          width = Integer.parseInt( args[ 0 ] );
```

```
51             height = Integer.parseInt( args[ 1 ] );
52          }
53
54          // create window in which applet will execute
55          JFrame applicationWindow =
56             new JFrame( "An applet running as an application" );
57
58          applicationWindow.addWindowListener(
59             new WindowAdapter() {
60                public void windowClosing( WindowEvent e )
61                {
62                   System.exit( 0 );
63                }
64             }
65          );
66
67          // create one applet instance
68          DrawShapes appletObject = new DrawShapes();
69          appletObject.setWidth( width );
70          appletObject.setHeight( height );
71
72          // call applet's init and start methods
73          appletObject.init();
74          appletObject.start();
75
76          // attach applet to center of window
77          applicationWindow.getContentPane().add( appletObject );
78
79          // set the window's size
80          applicationWindow.setSize( width, height );
81
```

Fig. 13.6 Creating a GUI-based application from an applet (part 2 of 4).

```
82          // showing the window causes all GUI components
83          // attached to the window to be painted
84          applicationWindow.show();
85       }
86
87       private class ButtonHandler implements ActionListener {
88          public void actionPerformed( ActionEvent e )
89          {
90             for ( int i = 0; i < choices.length; i++ )
91                if ( e.getSource() == choices[ i ] ) {
92                   drawingArea.setCurrentChoice( i );
93                   break;
94                }
95          }
96       }
97    }
98
99    // subclass of JPanel to allow drawing in a separate area
100   class DrawPanel extends JPanel {
101      private int currentChoice = -1;  // don't draw first time
102      private int width = 100, height = 100;
```

```
103
104     public DrawPanel( int w, int h )
105     {
106        width = ( w >= 0 ? w : 100 );
107        height = ( h >= 0 ? h : 100 );
108     }
109
110     public void paintComponent( Graphics g )
111     {
112        super.paintComponent( g );
113
114        switch( currentChoice ) {
115           case 0:
116              g.drawLine( randomX(), randomY(),
117                          randomX(), randomY() );
118              break;
119           case 1:
120              g.drawRect( randomX(), randomY(),
121                          randomX(), randomY() );
122              break;
123           case 2:
124              g.drawOval( randomX(), randomY(),
125                          randomX(), randomY() );
126              break;
127        }
128     }
129
130     public void setCurrentChoice( int c )
131     {
132        currentChoice = c;
133        repaint();
134     }
```

Fig. 13.6 Creating a GUI-based application from an applet (part 3 of 4).

```
135
136     private int randomX()
137        { return (int) ( Math.random() * width ); }
138
139     private int randomY()
140        { return (int) ( Math.random() * height ); }
141  }
```

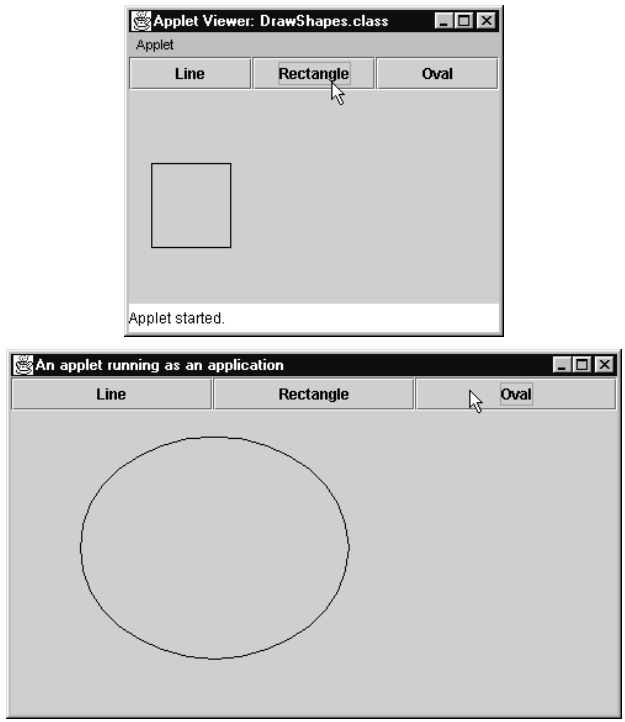


Fig. 13.6 Creating a GUI-based application from an applet (part 4 of 4).

```
1  // Fig. 13.7: MenuTest.java
2  // Demonstrating menus
3  import javax.swing.*;
4  import java.awt.event.*;
5  import java.awt.*;
6
7  public class MenuTest extends JFrame {
8     private Color colorValues[] =
9        { Color.black, Color.blue, Color.red, Color.green };
10    private JRadioButtonMenuItem colorItems[], fonts[];
11    private JCheckBoxMenuItem styleItems[];
12    private JLabel display;
```

Fig. 13.7 Using **JMenu**s and mnemonics (part 1 of 5).

```
13    private ButtonGroup fontGroup, colorGroup;
14    private int style;
15
16    public MenuTest()
17    {
18       super( "Using JMenus" );
19
20       JMenuBar bar = new JMenuBar();  // create menubar
21       setJMenuBar( bar );  // set the menubar for the JFrame
22
23       // create File menu and Exit menu item
24       JMenu fileMenu = new JMenu( "File" );
25       fileMenu.setMnemonic( 'F' );
26       JMenuItem aboutItem = new JMenuItem( "About..." );
27       aboutItem.setMnemonic( 'A' );
28       aboutItem.addActionListener(
29          new ActionListener() {
30             public void actionPerformed( ActionEvent e )
31             {
32                JOptionPane.showMessageDialog( MenuTest.this,
33                   "This is an example\nof using menus",
34                   "About", JOptionPane.PLAIN_MESSAGE );
35             }
36          }
37       );
38       fileMenu.add( aboutItem );
39
40       JMenuItem exitItem = new JMenuItem( "Exit" );
41       exitItem.setMnemonic( 'x' );
42       exitItem.addActionListener(
43          new ActionListener() {
44             public void actionPerformed( ActionEvent e )
45             {
46                System.exit( 0 );
47             }
48          }
49       );
50       fileMenu.add( exitItem );
```

```
51          bar.add( fileMenu );     // add File menu
52
53          // create the Format menu, its submenus and menu items
54          JMenu formatMenu = new JMenu( "Format" );
55          formatMenu.setMnemonic( 'r' );
56
57          // create Color submenu
58          String colors[] =
59             { "Black", "Blue", "Red", "Green" };
60          JMenu colorMenu = new JMenu( "Color" );
61          colorMenu.setMnemonic( 'C' );
62          colorItems = new JRadioButtonMenuItem[ colors.length ];
63          colorGroup = new ButtonGroup();
64          ItemHandler itemHandler = new ItemHandler();
65
```

Fig. 13.7 Using **JMenu**s and mnemonics (part 2 of 5).

```
66          for ( int i = 0; i < colors.length; i++ ) {
67             colorItems[ i ] =
68                new JRadioButtonMenuItem( colors[ i ] );
69             colorMenu.add( colorItems[ i ] );
70             colorGroup.add( colorItems[ i ] );
71             colorItems[ i ].addActionListener( itemHandler );
72          }
73
74          colorItems[ 0 ].setSelected( true );
75          formatMenu.add( colorMenu );
76          formatMenu.addSeparator();
77
78          // create Font submenu
79          String fontNames[] =
80             { "TimesRoman", "Courier", "Helvetica" };
81          JMenu fontMenu = new JMenu( "Font" );
82          fontMenu.setMnemonic( 'n' );
83          fonts = new JRadioButtonMenuItem[ fontNames.length ];
84          fontGroup = new ButtonGroup();
85
86          for ( int i = 0; i < fonts.length; i++ ) {
87             fonts[ i ] =
88                new JRadioButtonMenuItem( fontNames[ i ] );
89             fontMenu.add( fonts[ i ] );
90             fontGroup.add( fonts[ i ] );
91             fonts[ i ].addActionListener( itemHandler );
92          }
93
94          fonts[ 0 ].setSelected( true );
95          fontMenu.addSeparator();
96
97          String styleNames[] = { "Bold", "Italic" };
98          styleItems = new JCheckBoxMenuItem[ styleNames.length ];
99          StyleHandler styleHandler = new StyleHandler();
100
101         for ( int i = 0; i < styleNames.length; i++ ) {
102            styleItems[ i ] =
```

```
            new JCheckBoxMenuItem( styleNames[ i ] );
         fontMenu.add( styleItems[ i ] );
         styleItems[ i ].addItemListener( styleHandler );
      }

      formatMenu.add( fontMenu );
      bar.add( formatMenu );  // add Format menu

      display = new JLabel(
         "Sample Text", SwingConstants.CENTER );
      display.setForeground( colorValues[ 0 ] );
      display.setFont(
         new Font( "TimesRoman", Font.PLAIN, 72 ) );

      getContentPane().setBackground( Color.cyan );
      getContentPane().add( display, BorderLayout.CENTER );
```

Fig. 13.7 Using **JMenu**s and mnemonics (part 3 of 5).

```
      setSize( 500, 200 );
      show();
   }

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

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

   class ItemHandler implements ActionListener {
      public void actionPerformed( ActionEvent e )
      {
         for ( int i = 0; i < colorItems.length; i++ )
            if ( colorItems[ i ].isSelected() ) {
               display.setForeground( colorValues[ i ] );
               break;
            }

         for ( int i = 0; i < fonts.length; i++ )
            if ( e.getSource() == fonts[ i ] ) {
               display.setFont( new Font(
                  fonts[ i ].getText(), style, 72 ) );
               break;
            }

         repaint();
```

```
155         }
156      }
157
158      class StyleHandler implements ItemListener {
159         public void itemStateChanged( ItemEvent e )
160         {
161            style = 0;
162
163            if ( styleItems[ 0 ].isSelected() )
164               style += Font.BOLD;
165
166            if ( styleItems[ 1 ].isSelected() )
167               style += Font.ITALIC;
168
169            display.setFont( new Font(
170               display.getFont().getName(), style, 72 ) );
171
```

Fig. 13.7 Using **JMenu**s and mnemonics (part 4 of 5).

```
172            repaint();
173         }
174      }
175   }
```

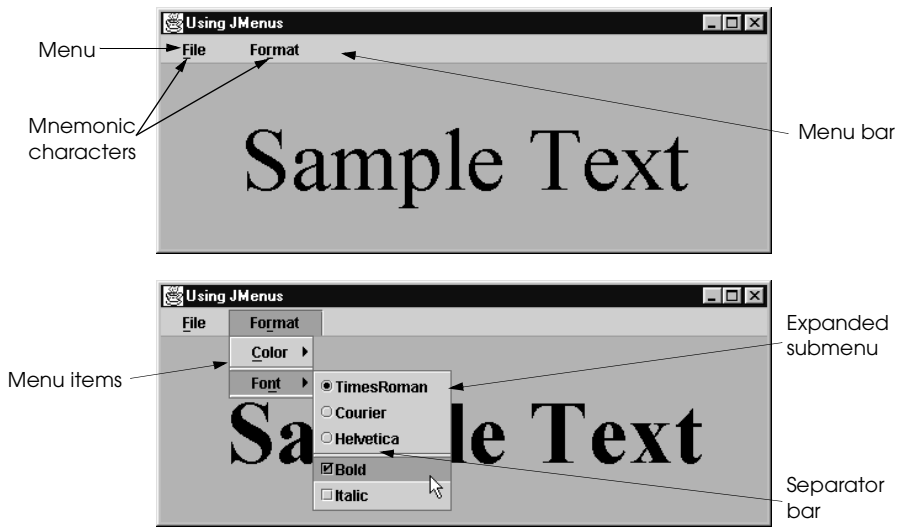


Fig. 13.7 Using **JMenu**s and mnemonics (part 5 of 5).

```
1  // Fig. 13.8: PopupTest.java
2  // Demonstrating JPopupMenus
3  import javax.swing.*;
4  import java.awt.event.*;
5  import java.awt.*;
6
```

Fig. 13.8 Using a **PopupMenu** object (part 1 of 3).

```
7  public class PopupTest extends JFrame {
8     private JRadioButtonMenuItem items[];
9     private Color colorValues[] =
10       { Color.blue, Color.yellow, Color.red };
11
12    public PopupTest()
13    {
14       super( "Using JPopupMenus" );
15
16       final JPopupMenu popupMenu = new JPopupMenu();
17       ItemHandler handler = new ItemHandler();
18       String colors[] = { "Blue", "Yellow", "Red" };
19       ButtonGroup colorGroup = new ButtonGroup();
20       items = new JRadioButtonMenuItem[ 3 ];
21
22       // construct each menu item and add to popup menu; also
23       // enable event handling for each menu item
24       for ( int i = 0; i < items.length; i++ ) {
25          items[ i ] = new JRadioButtonMenuItem( colors[ i ] );
26          popupMenu.add( items[ i ] );
27          colorGroup.add( items[ i ] );
28          items[ i ].addActionListener( handler );
29       }
30
31       getContentPane().setBackground( Color.white );
32
33       // define a MouseListener for the window that displays
34       // a JPopupMenu when the popup trigger event occurs
35       addMouseListener(
36          new MouseAdapter() {
37             public void mousePressed( MouseEvent e )
38                { checkForTriggerEvent( e ); }
39
40             public void mouseReleased( MouseEvent e )
41                { checkForTriggerEvent( e ); }
42
43             private void checkForTriggerEvent( MouseEvent e )
44             {
45                if ( e.isPopupTrigger() )
46                   popupMenu.show( e.getComponent(),
47                                   e.getX(), e.getY() );
48             }
49          }
50       );
```

```
51
52          setSize( 300, 200 );
53          show();
54       }
55
56       public static void main( String args[] )
57       {
58          PopupTest app = new PopupTest();
59
```

Fig. 13.8 Using a **PopupMenu** object (part 2 of 3).

```
60          app.addWindowListener(
61             new WindowAdapter() {
62                public void windowClosing( WindowEvent e )
63                {
64                   System.exit( 0 );
65                }
66             }
67          );
68       }
69
70       private class ItemHandler implements ActionListener {
71          public void actionPerformed( ActionEvent e )
72          {
73             // determine which menu item was selected
74             for ( int i = 0; i < items.length; i++ )
75                if ( e.getSource() == items[ i ] ) {
76                   getContentPane().setBackground(
77                      colorValues[ i ] );
78                   repaint();
79                   return;
80                }
81          }
82       }
83    }
```

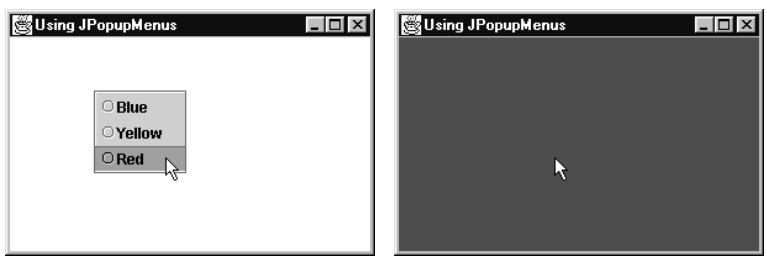


Fig. 13.8 Using a **PopupMenu** object (part 3 of 3).

```
1  // Fig. 13.9: LookAndFeelDemo.java
2  // Changing the look and feel.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class LookAndFeelDemo extends JFrame {
8     private String strings[] = { "Metal", "Motif", "Windows" };
9     private UIManager.LookAndFeelInfo looks[];
10    private JRadioButton radio[];
11    private ButtonGroup group;
12    private JButton button;
13    private JLabel label;
14    private JComboBox comboBox;
15
16    public LookAndFeelDemo()
17    {
18       super( "Look and Feel Demo" );
19
20       Container c = getContentPane();
21
22       JPanel northPanel = new JPanel();
23       northPanel.setLayout( new GridLayout( 3, 1, 0, 5 ) );
24       label = new JLabel( "This is a Metal look-and-feel",
25                           SwingConstants.CENTER );
26       northPanel.add( label );
27       button = new JButton( "JButton" );
28       northPanel.add( button );
29       comboBox = new JComboBox( strings );
30       northPanel.add( comboBox );
31
32       c.add( northPanel, BorderLayout.NORTH );
33
34       JPanel southPanel = new JPanel();
35       radio = new JRadioButton[ strings.length ];
36       group = new ButtonGroup();
37       ItemHandler handler = new ItemHandler();
38       southPanel.setLayout(
39          new GridLayout( 1, radio.length ) );
40
41       for ( int i = 0; i < radio.length; i++ ) {
42          radio[ i ] = new JRadioButton( strings[ i ] );
43          radio[ i ].addItemListener( handler );
44          group.add( radio[ i ] );
45          southPanel.add( radio[ i ] );
46       }
47
```

Fig. 13.9 Changing the look-and-feel of a Swing-based GUI (part 1 of 3).

```
48          c.add( southPanel, BorderLayout.SOUTH );
49
50          // get the installed look-and-feel information
51          looks = UIManager.getInstalledLookAndFeels();
52
53          setSize( 300, 200 );
54          show();
55
56          radio[ 0 ].setSelected( true );
57       }
58
59       private void changeTheLookAndFeel( int value )
60       {
61          try {
62             UIManager.setLookAndFeel(
63                looks[ value ].getClassName() );
64             SwingUtilities.updateComponentTreeUI( this );
65          }
66          catch ( Exception e ) {
67             e.printStackTrace();
68          }
69       }
70
71       public static void main( String args[] )
72       {
73          LookAndFeelDemo dx = new LookAndFeelDemo();
74
75          dx.addWindowListener(
76             new WindowAdapter() {
77                public void windowClosing( WindowEvent e )
78                {
79                   System.exit( 0 );
80                }
81             }
82          );
83       }
84
85       private class ItemHandler implements ItemListener {
86          public void itemStateChanged( ItemEvent e )
87          {
88             for ( int i = 0; i < radio.length; i++ )
89                if ( radio[ i ].isSelected() ) {
90                   label.setText( "This is a " +
91                      strings[ i ] + " look-and-feel" );
92                   comboBox.setSelectedIndex( i );
93                   changeTheLookAndFeel( i );
94                }
95          }
96       }
97    }
```

Fig. 13.9 Changing the look-and-feel of a Swing-based GUI (part 2 of 3).

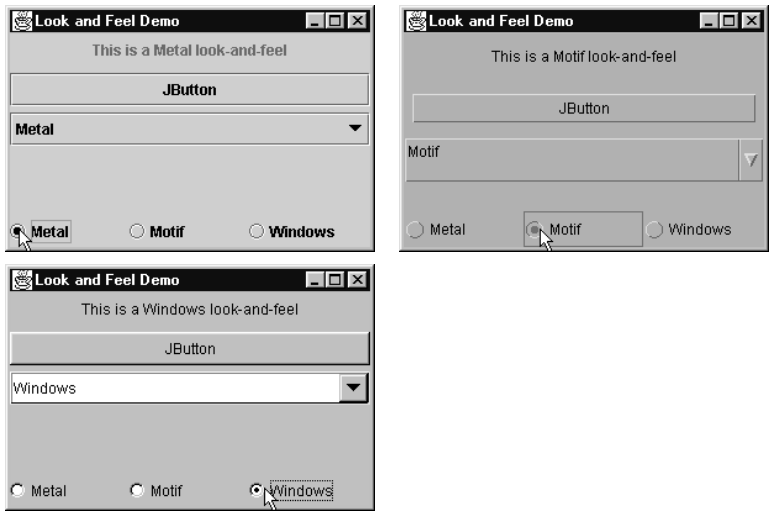


Fig. 13.9 Changing the look-and-feel of a Swing-based GUI (part 3 of 3).

```
1  // Fig. 13.10: DesktopTest.java
2  // Demonstrating JDesktopPane.
3  import javax.swing.*;
4  import java.awt.event.*;
5  import java.awt.*;
6
7  public class DesktopTest extends JFrame {
8     public DesktopTest()
9     {
10       super( "Using a JDesktopPane" );
11
12       JMenuBar bar = new JMenuBar();
13       JMenu addMenu = new JMenu( "Add" );
14       JMenuItem newFrame = new JMenuItem( "Internal Frame" );
15       addMenu.add( newFrame );
16       bar.add( addMenu );
17       setJMenuBar( bar );
```

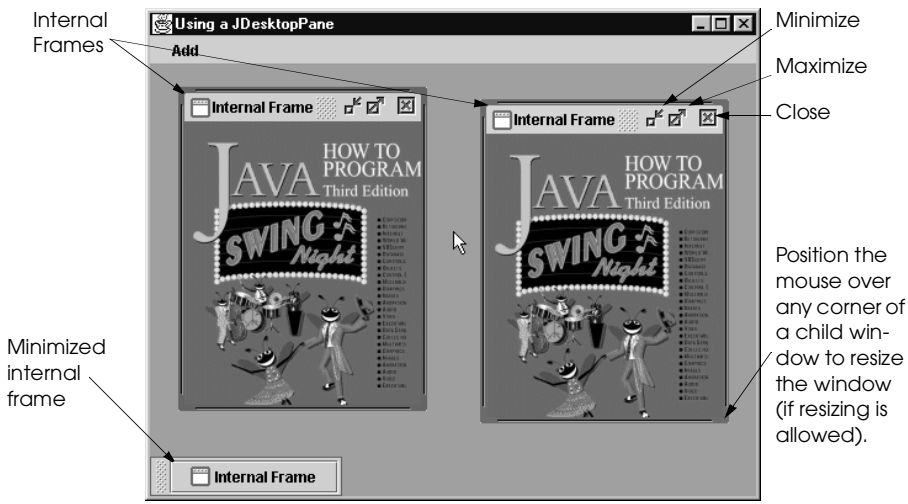
Fig. 13.10 Using **`JDesktopPane`** and **`JInternalFrame`** (part 1 of 3).

```
18
19       final JDesktopPane theDesktop = new JDesktopPane();
20       getContentPane().add( theDesktop );
21
22       newFrame.addActionListener(
23          new ActionListener() {
24             public void actionPerformed( ActionEvent e ) {
25                JInternalFrame frame =
26                   new JInternalFrame(
27                      "Internal Frame",
28                      true, true, true, true );
29
30                Container c = frame.getContentPane();
31                MyJPanel panel = new MyJPanel();
32
33                c.add( panel, BorderLayout.CENTER );
34                frame.setSize(
35                   panel.getImageWidthHeight().width,
36                   panel.getImageWidthHeight().height );
37                frame.setOpaque( true );
38                theDesktop.add( frame );
39             }
40          }
41       );
42
43       setSize( 500, 400 );
44       show();
45    }
46
47    public static void main( String args[] )
48    {
49       DesktopTest app = new DesktopTest();
```

```
50
51          app.addWindowListener(
52             new WindowAdapter() {
53                public void windowClosing( WindowEvent e )
54                {
55                   System.exit( 0 );
56                }
57             }
58          );
59       }
60    }
61
62    class MyJPanel extends JPanel {
63       private ImageIcon imgIcon;
64
65       public MyJPanel()
66       {
67          imgIcon = new ImageIcon( "jhtp3.gif" );
68       }
69
```

Fig. 13.10 Using **`JDesktopPane`** and **`JInternalFrame`** (part 2 of 3).

```
70       public void paintComponent( Graphics g )
71       {
72          imgIcon.paintIcon( this, g, 0, 0 );
73       }
74
75       public Dimension getImageWidthHeight()
76       {
77          return new Dimension( imgIcon.getIconWidth(),
78                                imgIcon.getIconHeight() );
79       }
80    }
```



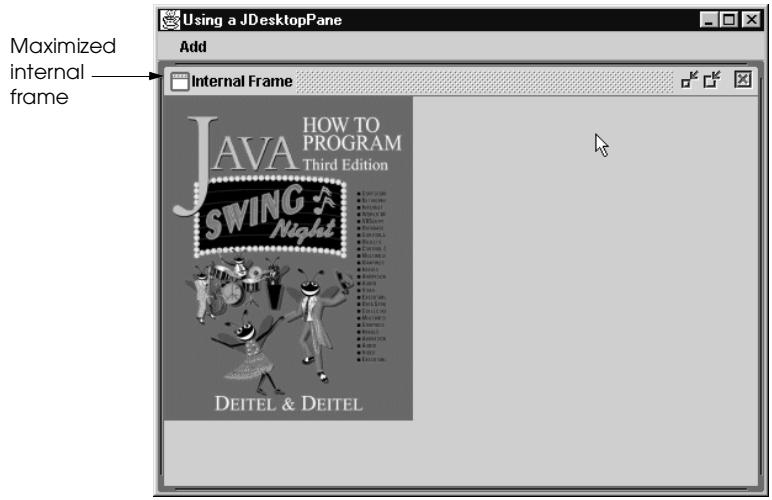


Fig. 13.10 Using **`JDesktopPane`** and **`JInternalFrame`** (part 3 of 3).

Layout manager	Description
`BoxLayout`	A layout manager that allows GUI components to be arranged left-to-right or top-to-bottom in a container. Class ***`Box`*** defines a container with `BoxLayout` as its default layout manager and provides static methods to create a `Box` with a horizontal or vertical `BoxLayout`.
`CardLayout`	A layout manager that stacks components like a deck of cards. If a component in the deck is a container, it can use any layout manager. Only the component at the "top" of the deck is visible.
`GridBagLayout`	A layout manager similar to `GridLayout`. Unlike `GridLayout` each component size can vary and components can be added in any order.

Fig. 13.11 Additional layout managers.

```
1  // Fig. 13.12: BoxLayoutDemo.java
2  // Demonstrating BoxLayout.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class BoxLayoutDemo extends JFrame {
8     public BoxLayoutDemo()
9     {
10       super( "Demostrating BoxLayout" );
11       final int SIZE = 3;
12
```

Fig. 13.12 Demonstrating the **`BoxLayout`** layout manager (part 1 of 3).

```
13       Container c = getContentPane();
14       c.setLayout( new BorderLayout( 30, 30 ) );
15
16       Box boxes[] = new Box[ 4 ];
17
18       boxes[ 0 ] = Box.createHorizontalBox();
19       boxes[ 1 ] = Box.createVerticalBox();
20       boxes[ 2 ] = Box.createHorizontalBox();
21       boxes[ 3 ] = Box.createVerticalBox();
22
23       // add buttons to boxes[ 0 ]
24       for ( int i = 0; i < SIZE; i++ )
25          boxes[ 0 ].add( new JButton( "boxes[0]: " + i ) );
26
27       // create strut and add buttons to boxes[ 1 ]
28       for ( int i = 0; i < SIZE; i++ ) {
29          boxes[ 1 ].add( Box.createVerticalStrut( 25 ) );
30          boxes[ 1 ].add( new JButton( "boxes[1]: " + i ) );
31       }
32
33       // create horizontal glue and add buttons to boxes[ 2 ]
34       for ( int i = 0; i < SIZE; i++ ) {
35          boxes[ 2 ].add( Box.createHorizontalGlue() );
36          boxes[ 2 ].add( new JButton( "boxes[2]: " + i ) );
37       }
38
39       // create rigid area and add buttons to boxes[ 3 ]
40       for ( int i = 0; i < SIZE; i++ ) {
41          boxes[ 3 ].add(
42             Box.createRigidArea( new Dimension( 12, 8 ) ) );
43          boxes[ 3 ].add( new JButton( "boxes[3]: " + i ) );
44       }
45
46       // create horizontal glue and add buttons to panel
47       JPanel panel = new JPanel();
48       panel.setLayout(
49          new BoxLayout( panel, BoxLayout.Y_AXIS ) );
50
51       for ( int i = 0; i < SIZE; i++ ) {
52          panel.add( Box.createGlue() );
```

```
53             panel.add( new JButton( "panel: " + i ) );
54          }
55
56          // place panels on frame
57          c.add( boxes[ 0 ], BorderLayout.NORTH );
58          c.add( boxes[ 1 ], BorderLayout.EAST );
59          c.add( boxes[ 2 ], BorderLayout.SOUTH );
60          c.add( boxes[ 3 ], BorderLayout.WEST );
61          c.add( panel, BorderLayout.CENTER );
62
63          setSize( 350, 300 );
64          show();
65       }
```

Fig. 13.12 Demonstrating the **`BoxLayout`** layout manager (part 2 of 3).

```
66
67       public static void main( String args[] )
68       {
69          BoxLayoutDemo app = new BoxLayoutDemo();
70
71          app.addWindowListener(
72             new WindowAdapter() {
73                public void windowClosing( WindowEvent e )
74                {
75                   System.exit( 0 );
76                }
77             }
78          );
79       }
80    }
```

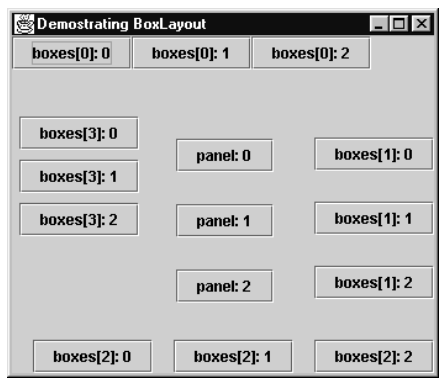
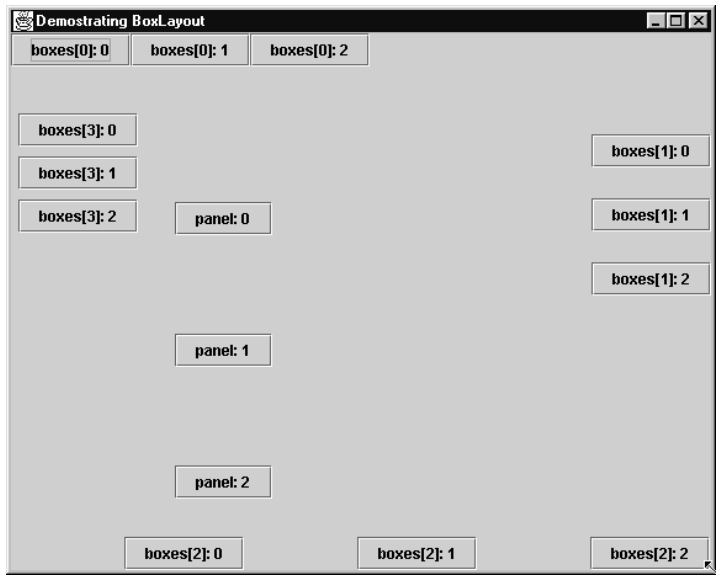


Fig. 13.12 Demonstrating the **`BoxLayout`** layout manager (part 3 of 3).



```
1  // Fig. 13.13: CardDeck.java
2  // Demonstrating CardLayout.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class CardDeck extends JFrame
8                         implements ActionListener {
9     private CardLayout cardManager;
10    private JPanel deck;
11    private JButton controls[];
12    private String names[] = { "First card", "Next card",
13                               "Previous card", "Last card" };
14
15    public CardDeck()
16    {
17       super( "CardLayout " );
18
19       Container c = getContentPane();
20
21       // create the JPanel with CardLayout
22       deck = new JPanel();
23       cardManager = new CardLayout();
24       deck.setLayout( cardManager );
25
26       // set up card1 and add it to JPanel deck
27       JLabel label1 =
28          new JLabel( "card one", SwingConstants.CENTER );
29       JPanel card1 = new JPanel();
30       card1.add( label1 );
31       deck.add( card1, label1.getText() ); // add card to deck
32
33       // set up card2 and add it to JPanel deck
34       JLabel label2 =
35          new JLabel( "card two", SwingConstants.CENTER );
36       JPanel card2 = new JPanel();
```

Fig. 13.13 Demonstrating the **`CardLayout`** layout manager (part 1 of 3).

```
37       card2.setBackground( Color.yellow );
38       card2.add( label2 );
39       deck.add( card2, label2.getText() ); // add card to deck
40
41       // set up card3 and add it to JPanel deck
42       JLabel label3 = new JLabel( "card three" );
43       JPanel card3 = new JPanel();
44       card3.setLayout( new BorderLayout() );
45       card3.add( new JButton( "North" ), BorderLayout.NORTH );
46       card3.add( new JButton( "West" ), BorderLayout.WEST );
47       card3.add( new JButton( "East" ), BorderLayout.EAST );
48       card3.add( new JButton( "South" ), BorderLayout.SOUTH );
49       card3.add( label3, BorderLayout.CENTER );
```

```
50          deck.add( card3, label3.getText() ); // add card to deck
51
52          // create and layout buttons that will control deck
53          JPanel buttons = new JPanel();
54          buttons.setLayout( new GridLayout( 2, 2 ) );
55          controls = new JButton[ names.length ];
56
57          for ( int i = 0; i < controls.length; i++ ) {
58             controls[ i ] = new JButton( names[ i ] );
59             controls[ i ].addActionListener( this );
60             buttons.add( controls[ i ] );
61          }
62
63          // add JPanel deck and JPanel buttons to the applet
64          c.add( buttons, BorderLayout.WEST );
65          c.add( deck, BorderLayout.EAST );
66
67          setSize( 450, 200 );
68          show();
69       }
70
71       public void actionPerformed( ActionEvent e )
72       {
73          if ( e.getSource() == controls[ 0 ] )
74             cardManager.first( deck ); // show first card
75          else if ( e.getSource() == controls[ 1 ] )
76             cardManager.next( deck );  // show next card
77          else if ( e.getSource() == controls[ 2 ] )
78             cardManager.previous( deck );  // show previous card
79          else if ( e.getSource() == controls[ 3 ] )
80            cardManager.last( deck );  // show last card
81       }
82
83       public static void main( String args[] )
84       {
85          CardDeck cardDeckDemo = new CardDeck();
86
87          cardDeckDemo.addWindowListener(
88             new WindowAdapter() {
```

Fig. 13.13 Demonstrating the **`CardLayout`** layout manager (part 2 of 3).

```
89                public void windowClosing( WindowEvent e )
90                {
91                   System.exit( 0 );
92                }
93             }
94          );
95       }
96    }
```

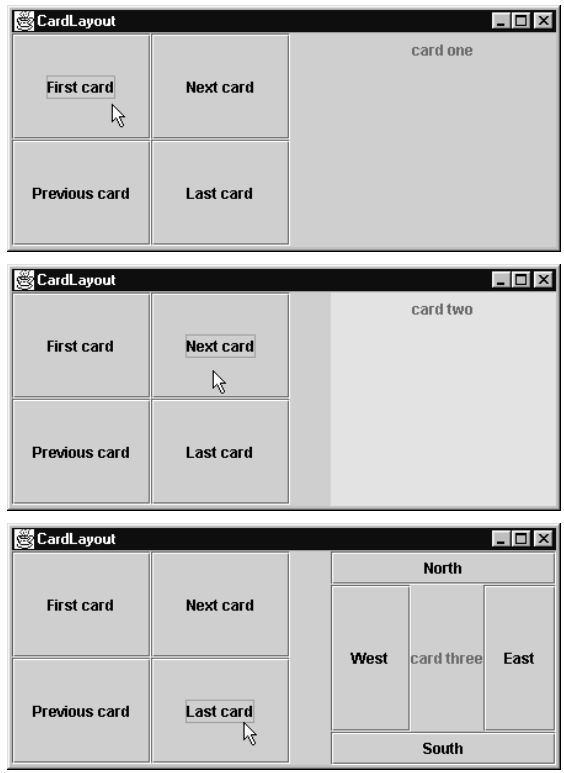


Fig. 13.13 Demonstrating the **`CardLayout`** layout manager (part 3 of 3).

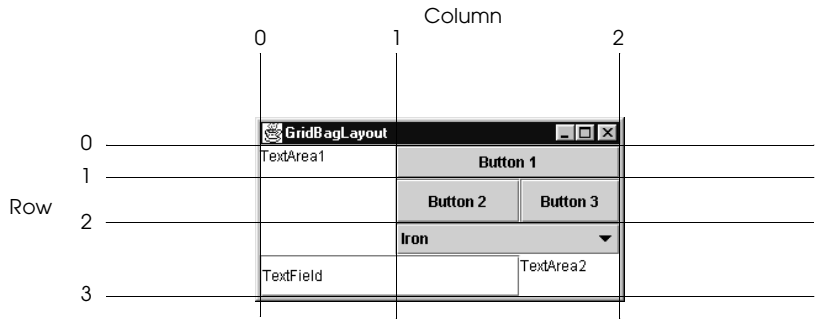


Fig. 13.14 Designing a GUI that will use **`GridBagLayout`**.

GridBagConstraints instance variable	Description
`gridx`	The column in which the component will be placed.
`gridy`	The row in which the component will be placed.
`gridwidth`	The number of columns the component occupies.
`gridheight`	The number of rows the component occupies.
`weightx`	The portion of extra space to allocate horizontally. The components can become wider when extra space is available.
`weighty`	The portion of extra space to allocate vertically. The components can become taller when extra space is available.

Fig. 13.15 `GridBagConstraints` instance variables.

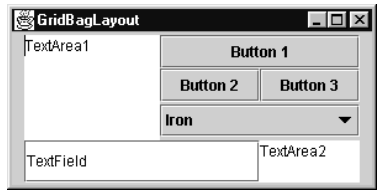


Fig. 13.16 **`GridBagLayout`** with the weights set to zero.

```
1  // Fig. 13.17: GridBagDemo.java
2  // Demonstrating GridBagLayout.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class GridBagDemo extends JFrame {
8     private Container container;
9     private GridBagLayout gbLayout;
10    private GridBagConstraints gbConstraints;
11
12    public GridBagDemo()
13    {
14       super( "GridBagLayout" );
15
16       container = getContentPane();
17       gbLayout = new GridBagLayout();
18       container.setLayout( gbLayout );
19
20       // instantiate gridbag constraints
21       gbConstraints = new GridBagConstraints();
22
23       JTextArea ta = new JTextArea( "TextArea1", 5, 10 );
24       JTextArea tx = new JTextArea( "TextArea2", 2, 2 );
25       String names[] = { "Iron", "Steel", "Brass" };
26       JComboBox cb = new JComboBox( names );
27       JTextField tf = new JTextField( "TextField" );
28       JButton b1 = new JButton( "Button 1" );
29       JButton b2 = new JButton( "Button 2" );
30       JButton b3 = new JButton( "Button 3" );
31
32       // text area
33       // weightx and weighty are both 0: the default
34       // anchor for all components is CENTER: the default
35       gbConstraints.fill = GridBagConstraints.BOTH;
36       addComponent( ta, 0, 0, 1, 3 );
37
38       // button b1
39       // weightx and weighty are both 0: the default
40       gbConstraints.fill = GridBagConstraints.HORIZONTAL;
41       addComponent( b1, 0, 1, 2, 1 );
42
43       // combo box
44       // weightx and weighty are both 0: the default
45       // fill is HORIZONTAL
46       addComponent( cb, 2, 1, 2, 1 );
47
48       // button b2
49       gbConstraints.weightx = 1000;  // can grow wider
50       gbConstraints.weighty = 1;     // can grow taller
51       gbConstraints.fill = GridBagConstraints.BOTH;
52       addComponent( b2, 1, 1, 1, 1 );
```

Fig. 13.17 Demonstrating the **`GridBagLayout`** layout manager (part 1 of 3).

```
53
54          // button b3
55          // fill is BOTH
56          gbConstraints.weightx = 0;
57          gbConstraints.weighty = 0;
58          addComponent( b3, 1, 2, 1, 1 );
59
60          // textfield
61          // weightx and weighty are both 0: fill is BOTH
62          addComponent( tf, 3, 0, 2, 1 );
63
64          // textarea
65          // weightx and weighty are both 0: fill is BOTH
66          addComponent( tx, 3, 2, 1, 1 );
67
68          setSize( 300, 150 );
69          show();
70       }
71
72       // addComponent is programmer defined
73       private void addComponent( Component c,
74          int row, int column, int width, int height )
75       {
76          // set gridx and gridy
77          gbConstraints.gridx = column;
78          gbConstraints.gridy = row;
79
80          // set gridwidth and gridheight
81          gbConstraints.gridwidth = width;
82          gbConstraints.gridheight = height;
83
84          // set constraints
85          gbLayout.setConstraints( c, gbConstraints );
86          container.add( c );      // add component
87       }
88
89       public static void main( String args[] )
90       {
91          GridBagDemo app = new GridBagDemo();
92
93          app.addWindowListener(
94             new WindowAdapter() {
95                public void windowClosing( WindowEvent e )
96                {
97                   System.exit( 0 );
98                }
99             }
100         );
101      }
102   }
```

Fig. 13.17 Demonstrating the **`GridBagLayout`** layout manager (part 2 of 3).

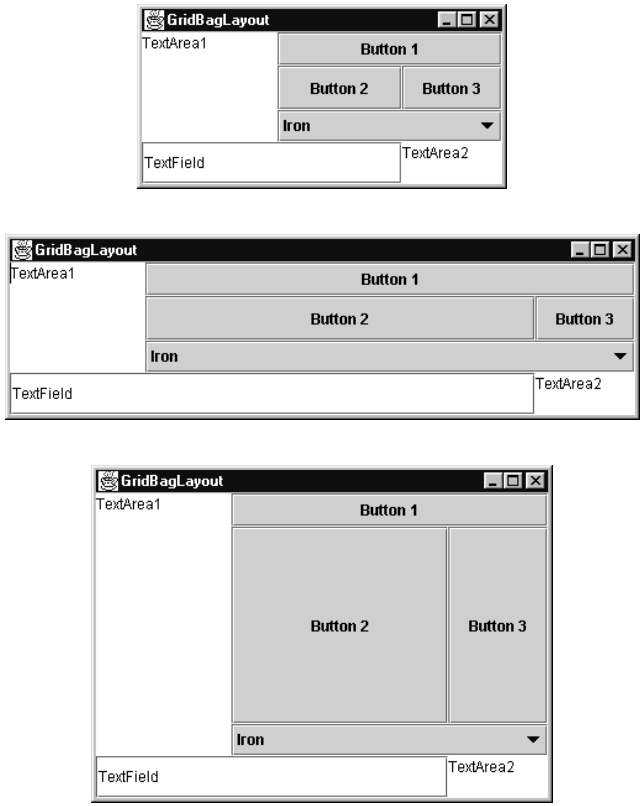


Fig. 13.17 Demonstrating the **`GridBagLayout`** layout manager (part 3 of 3).

```
1  // Fig. 13.18: GridBagDemo2.java
2  // Demonstrating GridBagLayout constants.
3  import javax.swing.*;
4  import java.awt.*;
5  import java.awt.event.*;
6
7  public class GridBagDemo2 extends JFrame {
8     private GridBagLayout gbLayout;
9     private GridBagConstraints gbConstraints;
10    private Container container;
11
12    public GridBagDemo2()
13    {
14       super( "GridBagLayout" );
15
16       container = getContentPane();
17       gbLayout = new GridBagLayout();
18       container.setLayout( gbLayout );
19
20       // instantiate gridbag constraints
21       gbConstraints = new GridBagConstraints();
22
23       // create GUI components
24       String metals[] = { "Copper", "Aluminum", "Silver" };
25       JComboBox comboBox = new JComboBox( metals );
26
27       JTextField textField = new JTextField( "TextField" );
28
29       String fonts[] = { "Serif", "Monospaced" };
30       JList list = new JList( fonts );
31
32       String names[] =
33          { "zero", "one", "two", "three", "four" };
34       JButton buttons[] = new JButton[ names.length ];
35
36       for ( int i = 0; i < buttons.length; i++ )
37          buttons[ i ] = new JButton( names[ i ] );
```

Fig. 13.18 Demonstrating the **`GridBagConstraints`** constants **`RELATIVE`** and **`REMAINDER`** (part 1 of 3).

```
38
39       // define GUI component constraints
40       // textField
41       gbConstraints.weightx = 1;
42       gbConstraints.weighty = 1;
43       gbConstraints.fill = GridBagConstraints.BOTH;
44       gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
45       addComponent( textField );
46
47       // buttons[0] -- weightx and weighty are 1: fill is BOTH
```

```
48          gbConstraints.gridwidth = 1;
49          addComponent( buttons[ 0 ] );
50
51          // buttons[1] -- weightx and weighty are 1: fill is BOTH
52          gbConstraints.gridwidth = GridBagConstraints.RELATIVE;
53          addComponent( buttons[ 1 ] );
54
55          // buttons[2] -- weightx and weighty are 1: fill is BOTH
56          gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
57          addComponent( buttons[ 2 ] );
58
59          // comboBox -- weightx is 1: fill is BOTH
60          gbConstraints.weighty = 0;
61          gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
62          addComponent( comboBox );
63
64          // buttons[3] -- weightx is 1: fill is BOTH
65          gbConstraints.weighty = 1;
66          gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
67          addComponent( buttons[ 3 ] );
68
69          // buttons[4] -- weightx and weighty are 1: fill is BOTH
70          gbConstraints.gridwidth = GridBagConstraints.RELATIVE;
71          addComponent( buttons[ 4 ] );
72
73          // list -- weightx and weighty are 1: fill is BOTH
74          gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
75          addComponent( list );
76
77          setSize( 300, 200 );
78          show();
79       }
80
81       // addComponent is programmer-defined
82       private void addComponent( Component c )
83       {
84          gbLayout.setConstraints( c, gbConstraints );
85          container.add( c );      // add component
86       }
87
```

Fig. 13.18 Demonstrating the **`GridBagConstraints`** constants **`RELATIVE`** and **`REMAINDER`** (part 2 of 3).

```
88       public static void main( String args[] )
89       {
90          GridBagDemo2 app = new GridBagDemo2();
91
92          app.addWindowListener(
93             new WindowAdapter() {
94                public void windowClosing( WindowEvent e )
95                {
96                   System.exit( 0 );
97                }
98             }
```

```
99          );
100      }
101   }
```

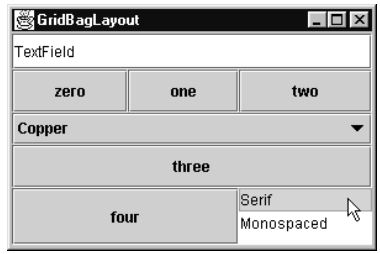


Fig. 13.18 Demonstrating the **`GridBagConstraints`** constants **`RELATIVE`** and **`REMAINDER`** (part 3 of 3).