Capitolo 3 – Componenti GUI avanzati

Outline

3.1 Introduction
3.2 `JTextArea`
3.3 Creating a Customized Subclass of `JPanel`
3.4 Creating a Self-Contained Subclass of `JPanel`
3.5 `JSlider`
3.6 Windows
3.7 Designing Programs that Execute as Applets or Applications
3.8 Using Menus with Frames
3.9 Using `JPopupMenus`
3.10 Pluggable Look-and-Feel
3.11 Using `JDesktopPane` and `JInternalFrame`
3.12 Layout Managers
3.13 `BoxLayout` Layout Manager
3.14 `CardLayout` Layout Manager
3.15 `GridBagLayout` Layout Manager
3.16 `GridBagConstraints` Constants `RELATIVE` and `REMAINDER`

3.1 Introduction

- Continue study of GUIs
  - Advanced components and layout managers
  - Begin with **`JTextArea`**
  - Customize **`JPanel`**
  - Design a program to run as an applet and application
  - Create and use menus
  - Pluggable look and feel
  - Multiple windows

3.2 JTextArea

- **JTextArea**
  - Area for manipulating multiple lines of text
  - Like **JTextField**, inherits from **JTextComponent**
    - Many of the same methods
  - Does not have automatic scrolling
  - Methods
    - **getSelectedText**
      - Returns selected text (dragging mouse over text)
    - **setText(string)**
  - Constructor
    - **JTextArea(string, numRows, numColumns)**
- **JScrollPane**
  - Provides scrolling for a component

3.2 JTextArea

- **JScrollPane**
  - Initialize with component
    - **new JScrollPane(myComponent)**
  - Can set scrolling policies (always, as needed, never)
  - Methods **setHorizontalScrollBarPolicy**, **setVerticalScrollBarPolicy**
    - Constants:
    - **JScrollPane.VERTICAL_SCROLLBAR_ALWAYS**
    - **JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED**
    - **JScrollPane.VERTICAL_SCROLLBAR_NEVER**
      - Similar for **HORIZONTAL**
    - If set to **JScrollPane.HORIZONTAL_SCROLLBAR_NEVER**, word wrap

3.2 JTextArea

- **Box** container
  - Uses **BoxLayout** layout manager
    - More detail in section 3.13
  - Arrange GUI components horizontally or vertically
  - **Box b = Box.createHorizontalBox();**
    - **static** method **Box.createHorizontalBox**
    - Arranges components attached to it from left to right, in order attached
    - Attach components to **Box** with **add**

```
 Copying selected text from one text area to another.
port java.awt.*;
port java.awt.event.*;
port javax.swing.*;

blic class TextAreaDemo extends JFrame {
 private JTextArea t1, t2;
 private JButton copy;

 public TextAreaDemo()
 {
    super( "TextArea Demo" );

    Box b = Box.createHorizontalBox();

    String s = "This is a demo string to\n
               "illustrate copying text\n"
               "from one TextArea to \n" +
               "another TextArea using an\n"+
               "external event\n";

    t1 = new JTextArea( s, 10, 15 );
    b.add( new JScrollPane( t1 ) );

    copy = new JButton( "Copy >>>" );
    copy.addActionListener(
       new ActionListener() {
          public void actionPerformed( ActionEvent e )
          {
             t2.setText( t1.getSelectedText() );
```

Create **Box** container.

Create a new **JTextArea**, initialized with String s, 10 rows, 15 columns.

Create a **JScrollPane**, providing scrolling for **t1**. Attach the **JScrollPane** to **Box b**.

Set the selected text of **t1** as the text of **t2**.

Outline

1. import

1.1 Declarations

1.2 Create Box container

1.3 JTextArea

ollPane

1.5 Create JButton

vent

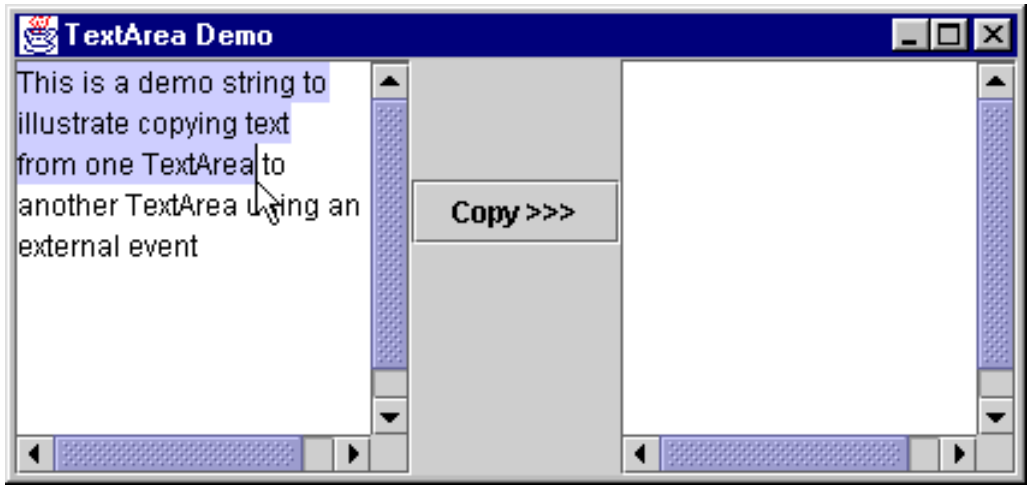
```
      }
   );
   b.add( copy );

   t2 = new JTextArea( 10, 15 );
   t2.setEditable( false );
   b.add( new JScrollPane( t2 ) );

   Container c = getContentPane();
   c.add( b );    // Box placed in BorderLayout.CENTER
   setSize( 425, 200 );
   show();
}

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

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

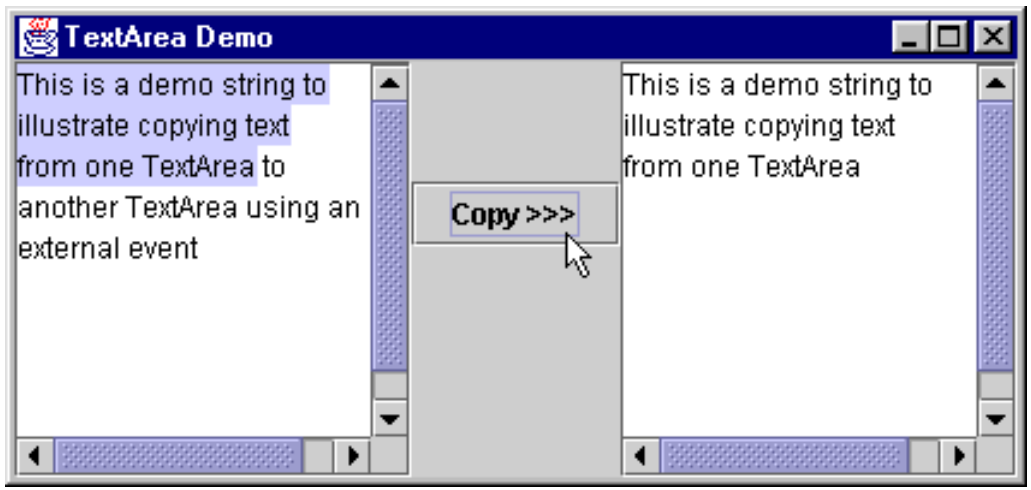
Add the **copy** button to the **Box** container **b**.

Outline

1.7 JTextArea

2. main





Outline

Program Output

3.3 Creating a Customized Subclass of `JPanel`

- **`JPanel`**
  - Can be used as a dedicated drawing area
    - Receives mouse events
    - Extended to create new components
  - Combining Swing GUI and drawing in one window can lead to errors
    - Fix problem by separating GUI and graphics
- Method **`paintComponent`**
  - Swing components that inherit from **`JComponent`** contain this method
  - Helps draw properly

3.3 Creating a Customized Subclass of `JPanel`

- Method **`paintComponent`**
  - Override as follows

```
public void paintComponent( Graphics g )
{
   super.paintComponent( g );

   // your additional drawing code
}
```

    - Call to superclass constructor - first statement
      - Ensures drawing occurs in proper order
      - If not called, errors usually occur
- **`JFrame`** and **`JApplet`**
  - Not subclasses of **`JComponent`**
    - Override method **`paint`**

3.3 Creating a Customized Subclass of `JPanel`

- Creating customized subclasses
  - Inherit from **`JPanel`**

```
6  public class CustomPanel extends JPanel {
```

  - Override method **`paintComponent`**

```
10    public void paintComponent( Graphics g )
11    {
12       super.paintComponent( g );
```


```
Fig. 3.2: CustomPanel.java
A customized JPanel class.
port java.awt.*;
port javax.swing.*;

blic class CustomPanel extends JPanel {
 public final static int CIRCLE = 1, SQUARE = 2;
 private int shape;

 public void paintComponent( Graphics g )
 {
    super.paintComponent( g );

    if ( shape == CIRCLE )
       g.fillOval( 50, 10, 60, 60 );
    else if ( shape == SQUARE )
       g.fillRect( 50, 10, 60, 60 );
 }

 public void draw( int s )
 {
    shape = s;
    repaint();
 }
```

Create a customized subclass of **`JPanel`** (inherit from **`JPanel`**).

Override **`paintComponent`**, call to superclass constructor is first statement.

`repaint` schedules a repaint operation, which calls **`paintComponent`**.

Outline

1. `import`

1.1 Class CustomPanel (extends

1.2 Instance variables

2. `paintComponent`

3. `draw`

```
 Using a customized Panel object.
port java.awt.*;
port java.awt.event.*;
port javax.swing.*;

blic class CustomPanelTest extends JFrame {
 private JPanel buttonPanel;
 private CustomPanel myPanel;
 private JButton circle, square;

 public CustomPanelTest()
 {
    super( "CustomPanel Test" );

    myPanel = new CustomPanel();   // instantiate canvas
    myPanel.setBackground( Color.green );

    square = new JButton( "Square" );
    square.addActionListener(
       new ActionListener() {
          public void actionPerformed( ActionEvent e )
          {
             myPanel.draw( CustomPanel.SQUARE );
          }
       }
    );

    circle = new JButton( "Circle" );
```

Create a new **`CustomPanel`** object.

Call **`draw`**, which calls **`repaint`**, which calls **`paintComponent`**.

Outline

1. Class `CustomPanelTest` (extends `JFrame`)

1.1 `CustomPanel`

1.2 `JButton`

2. Event handler

2.1 `draw`

```
      circle.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               myPanel.draw( CustomPanel.CIRCLE );
            }
         }
      );

      buttonPanel = new JPanel();
      buttonPanel.setLayout( new GridLayout( 1, 2 ) );
      buttonPanel.add( circle );
      buttonPanel.add( square );

      Container c = getContentPane();
      c.add( myPanel, BorderLayout.CENTER );
      c.add( buttonPanel, BorderLayout.SOUTH );

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

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

Outline

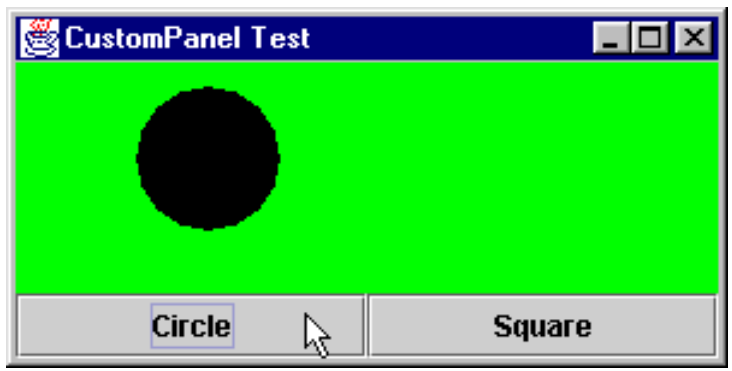
2.2 Event handler

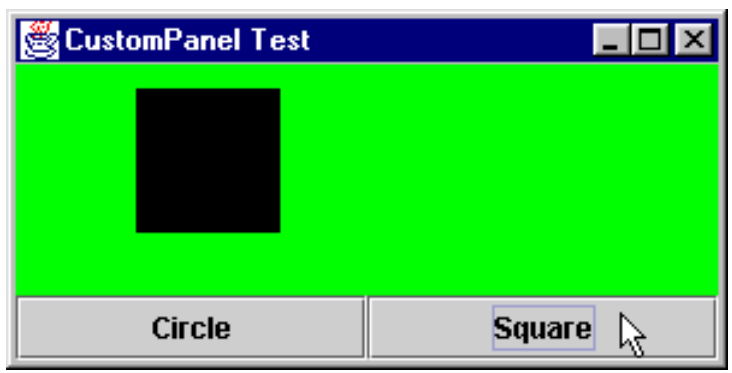
2.3 `draw`

2.4 `JPanel`

3. `main`

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





Program Output

3.4 Creating a Self-Contained Subclass of `JPanel`

- Events
  - **`JPanels`** do not create events like buttons
  - Can recognize lower-level events
    - Mouse and key events
- Example
  - Create a subclass of **`JPanel`** named **`SelfContainedPanel`** that listens for its own mouse events
    - Draws an oval on itself (overrides **`paintComponent`**)
  - Import **`SelfContainedPanel`** into another class
    - The other class contains its own mouse handlers
  - Add an instance of **`SelfContainedPanel`** to the content pane

3.4 Creating a Self-Contained Subclass of of `JPanel`

- Method **`getPreferredSize`**
  - Inherited from **`java.awt.Component`**
  - Preferred size of a component when in a GUI
  - Returns width and height as a **`Dimension`** object (**`java.awt`**)

```
Fig. 3.5: SelfContainedPanelTest.java
Creating a self-contained subclass of JPanel
that processes its own mouse events.
port java.awt.*;
port java.awt.event.*;
port javax.swing.*;
port com.deitel.jhtp3.ch13.SelfContainedPanel;

blic class SelfContainedPanelTest extends JFrame {
 private SelfContainedPanel myPanel;

 public SelfContainedPanelTest()
 {
    myPanel = new SelfContainedPanel();
    myPanel.setBackground( Color.yellow );

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

    addMouseMotionListener(
       new MouseMotionListener() {
          public void mouseDragged( FMouseEvent e )
          {
             setTitle( "Dragging: x=" + e.getX() +
                       "; y=" + e.getY() );
          }
```

Import **SelfContainedPanel**, in a user-defined package.

Create **SelfContainedPanel** object, add to content pane.

Separate event handler for application window.

Outline

1. **import SelfContainedPanel**

1.1 Create object

t

tion

window

```
         public void mouseMoved( MouseEvent e )
         {
            setTitle( "Moving: x=" + e.getX() +
                      "; y=" + e.getY() );
         }
      }
   );

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

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

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

Outline

2. main

```
Fig. 3.5: SelfContainedPanel.java
A self-contained JPanel class that
handles its own mouse events.
ckage com.deitel.jhtp3.ch13;

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

blic class SelfContainedPanel extends JPanel {
 private int x1, y1, x2, y2;

 public SelfContainedPanel()
 {
    addMouseListener(
       new MouseAdapter() {
          public void mousePressed( MouseEvent e )
          {
             x1 = e.getX();
             y1 = e.getY();
          }

          public void mouseReleased( MouseEvent e )
          {
             x2 = e.getX();
             y2 = e.getY();
             repaint();
          }
       }
    );
```

Anonymous inner class for the mouse event handler, different from the application event handler.

1. package

1.1 Class `SelfContainedPanel` (extends `JPanel`)

```
      addMouseMotionListener(
         new MouseMotionAdapter() {
            public void mouseDragged( MouseEvent e )
            {
               x2 = e.getX();
               y2 = e.getY();
               repaint();
            }
         }
      );
   }

   public Dimension getPreferredSize()
   {
      return new Dimension( 150, 100 );
   }

   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      g.drawOval( Math.min( x1, x2 ), Math.min( y1, y2 ),
                  Math.abs( x1 - x2 ), Math.abs( y1 - y2 ) );
   }
```

Preferred size of component.

Override **paintComponent**. Superclass constructor first statement.

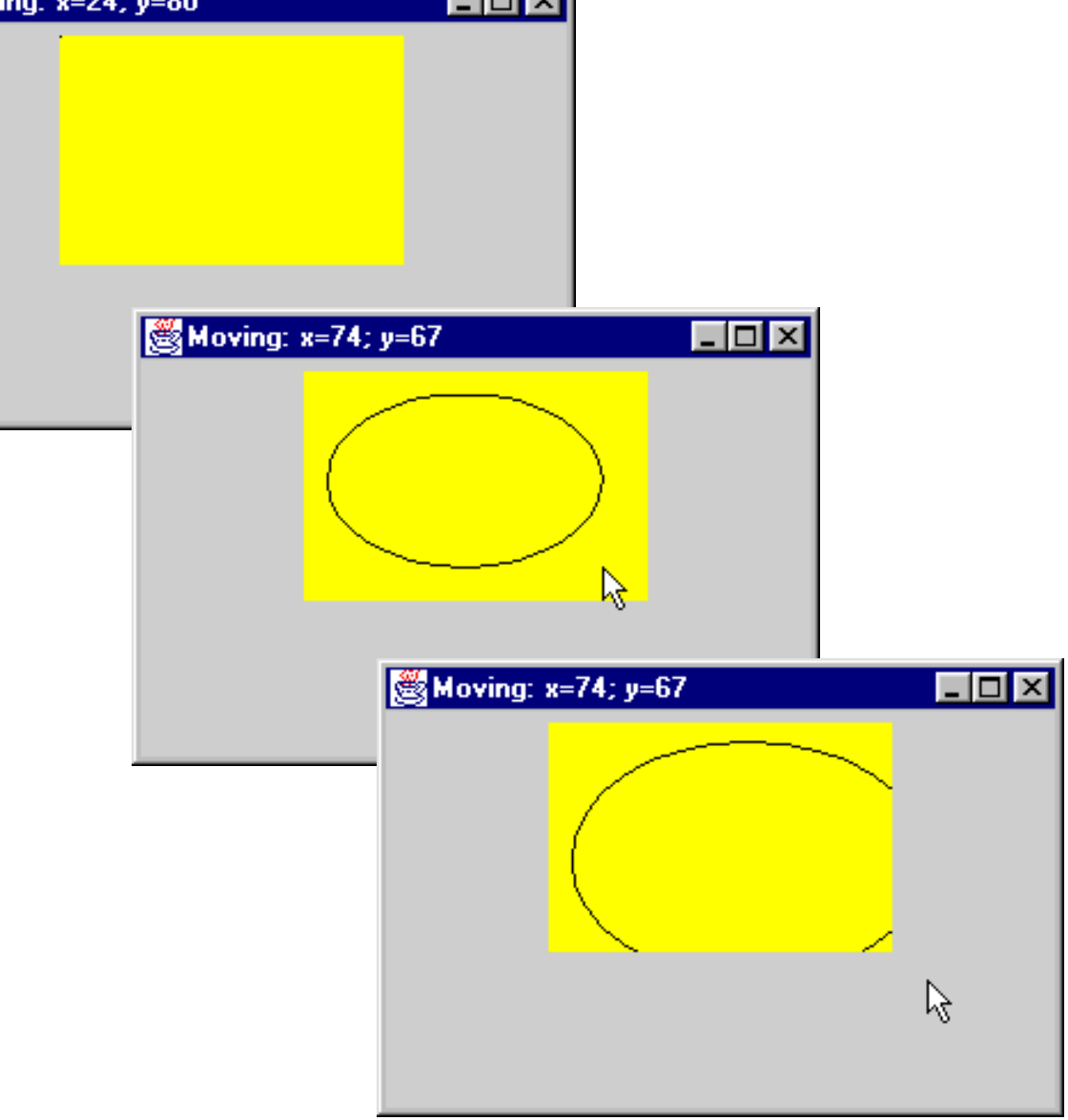
Outline

2. **getPreferredSize**

3. **paintComponent**

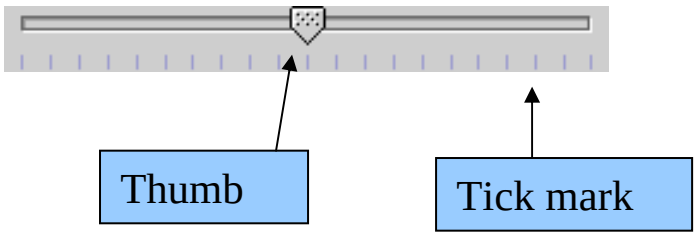
Outline

Program Output



3.5 JSlider

- **JSlider**s
  - Select from a range of integer values



  - Highly customizable
    - Snap-to ticks, major and minor ticks, labels
  - When has focus (currently selected GUI component)
    - Use mouse or keyboard
      - Arrow or keys to move thumb, *Home*, *End*
  - Have horizontal or vertical orientation
    - Minimum value at left/bottom, maximum at right/top
    - Thumb indicates current value

3.5 JSlider

- Methods
  - Constructor
  - **JSlider(orientation_CONSTANT, min, max, initialValue)**
    - **orientation_CONSTANT**
      - **SwingConstants.HORIZONTAL**
      - **SwingConstants.VERTICAL**
    - **min, max** - range of values for slider
    - **initialValue** - starting location of thumb

```
19       diameter = new JSlider( SwingConstants.HORIZONTAL,
20                               0, 200, 10 );
```


3.5 JSlider

- Methods
  - **setMajorTickSpacing(n)**
    - Each tick mark represents **n** values in range
  - **setPaintTicks(boolean)**
    - **false** (default) - tick marks not shown
  - **getValue()**
    - Returns current thumb position
- Events
  - JSliders generates **ChangeEvent**s
    - **addChangeListener**
    - Define method **stateChanged**

3.5 JSlider

- Example program
  - Application class **SliderDemo**
    - Creates a **JSlider**
    - **JSlider** event handler sets diameter
  - Class **OvalPanel** (subclass of **JPanel**)
    - Method **setDiameter**
    - Updates diameter, **repaint**

```
Fig. 3.5: SliderDemo.java
 Using JSliders to size an oval.
port java.awt.*;
port java.awt.event.*;
port javax.swing.*;
port javax.swing.event.*;

blic class SliderDemo extends JFrame {
 private JSlider diameter;
 private OvalPanel myPanel;

 public SliderDemo()
 {
    super( "Slider Demo" );

    myPanel = new OvalPanel();
    myPanel.setBackground( Color.yellow );

    diameter = new JSlider( SwingConstants.HORIZONTAL,
                            0, 200, 10 );
    diameter.setMajorTickSpacing( 10 );
    diameter.setPaintTicks( true );
    diameter.addChangeListener(
       new ChangeListener() {
          public void stateChanged( ChangeEvent e )
          {
             myPanel.setDiameter( diameter.getValue() );
          }
       }
    );
```

Create a new **OvalPanel** object.

Create a new **JSlider** object, notice constructor arguments.

Custo
visible

JSliders generate **ChangeEvent**s. Call method **setDiameter**, passing current thumb value as argument.

Outline

1. **import**

1.1 **OvalPanel** object

1.2 JSlider object

1.3 **setMajorTick**

ks

2. Event handler

```
   Container c = getContentPane();
   c.add( diameter, BorderLayout.SOUTH );
   c.add( myPanel, BorderLayout.CENTER );

   setSize( 220, 270 );
   show();
}

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

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

Outline

3. main

```
Fig. 3.5: OvalPanel.java
A customized JPanel class.
port java.awt.*;
port javax.swing.*;

blic class OvalPanel extends JPanel {
 private int diameter = 10;

 public void paintComponent( Graphics g )
 {
    super.paintComponent( g );
    g.fillOval( 10, 10, diameter, diameter );
 }

 public void setDiameter( int d )
 {
    diameter = ( d >= 0 ? d : 10 );  // default diameter 10
    repaint();
 }

 // the following methods are used by layout managers
 public Dimension getPreferredSize()
 {
    return new Dimension( 200, 200 );
 }

 public Dimension getMinimumSize()
 {
    return getPreferredSize();
 }
```

Draw an oval (circle) using **`diameter`**.

Update **`diameter`** then **`repaint`**.

Outline

1. Class `OvalPanel` (extends `JPanel`)

1.1 Override `paintComponent`

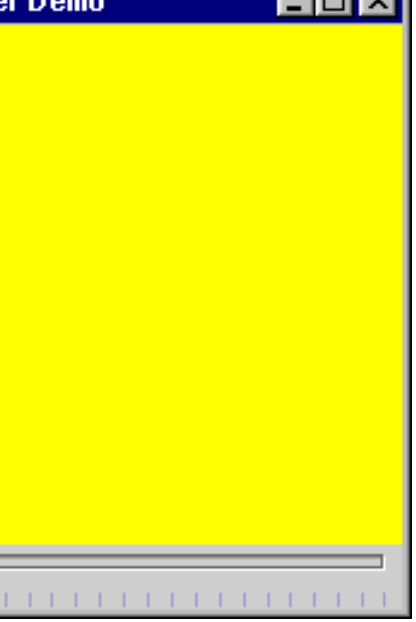
1.2 `setDiameter`

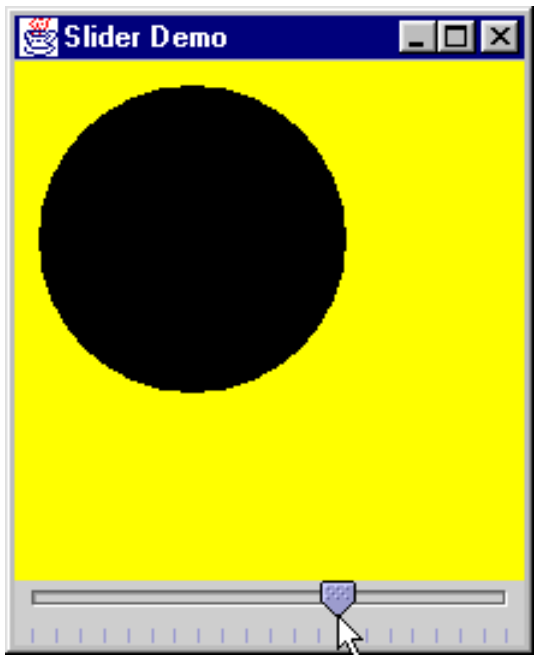
1.2.1 `repaint`

1.3 `getPreferredSize`

1.4 `getMinimumSize`

Program Output





3.6 Windows

- **`JFrame`**
  - Inherits from **`java.awt.Frame`**, derived from **`java.awt.Window`**
  - **`JFrame`** is a window with a title bar and a border
    - Heavyweight component
    - Window part of local platform's GUI components
- **`JFrame`** operations when user closes window
  - Controlled with method **`setDefaultCloseOperation`**
    - Interface **`WindowConstants`** (**`javax.swing`**) has three constants to use
    - **`DISPOSE_ON_CLOSE`**, **`DO_NOTHING_ON_CLOSE`**, **`HIDE_ON_CLOSE`** (default)

3.6 Windows

- Windows take up valuable resources
  - Explicitly remove windows when not needed
  - Method **`dispose`** (of class **`Window`**, indirect superclass of **`JFrame`**)
    - Or, use **`setDefaultCloseOperation`**
  - **`DO_NOTHING_ON_CLOSE`** -
    - You determine what happens when user wants to close window
- Display
  - By default, window not displayed until method **`show`** called
  - Can display by calling method **`setVisible( true )`**
  - Method **`setSize`**
    - Set a window's size else only title bar will appear

3.6 Windows

- Windows generate window events
  - **`addWindowListener`**
  - **`WindowListener`** interface has 7 methods
    - **`windowActivated`**
    - **`windowClosed`** (called after window closed)
    - **`windowClosing`** (called when user initiates closing)
    - **`windowDeactivated`**
    - **`windowIconified`** (minimized)
    - **`windowDeiconified`**
    - **`windowOpened`**

3.7 Designing Programs that Execute as Applets or Applications

- Java programs
  - Sometimes desirable to run as application and applet
    - Run from web browser as applet or download and run as application
- **`JFrame`**s
  - Used for GUI based applications
    - When **`JFrame`** closed, program terminates
  - In this section, convert applet into a GUI-based application
  - Program can execute as an applet
    - Define **`main`**, so it can execute as an application
  - Program has three buttons, to draw a rectangle, square, or circle
    - Uses custom subclass of **`JPanel`**

3.7 Designing Programs that Execute as Applets or Applications

- Sizing
  - HTML document that loads applet specifies width and height
  - When program executes as an application
    - Can give arguments to program (command-line arguments)
    - **`java DrawShapes 600 400`**
  - Arguments passed to **`main`** as array of **`String`**s called **`args`**
    - First argument is first element
    - Length of array is number of arguments

3.7 Designing Programs that Execute as Applets or Applications

```
41    public static void main( String args[] )
42    {
43       int width, height;
45       if ( args.length != 2 ) {  // no command-line arguments
46          width = 300;
48       }
49       else {
50          width = Integer.parseInt( args[ 0 ] );
51          height = Integer.parseInt( args[ 1 ] );
52       }
```

- Test for proper number of arguments
  - If so, assign them to variables **`width`** and **`height`**

3.7 Designing Programs that Execute as Applets or Applications

- Applets
  - When executing in **`appletviewer`**, window supplied by **`appletviewer`**
  - Applications must create their own windows

```
55       JFrame applicationWindow =
56          new JFrame( "An applet running as an application" );
```

  - Create **`JFrame`**, attach applet to it
    - Provide **`WindowAdapter`** to close window
  - When executing in **`appletviewer`**, object of applet class created
    - In application, must explicitly create object

```
68       DrawShapes appletObject = new DrawShapes();
```


3.7 Designing Programs that Execute as Applets or Applications

- In applets
  - **`init`**, **`start`**, and **`paint`** automatically called
    - Must be explicitly called in applications

```
73        appletObject.init();
74        appletObject.start();
```

  - **`appletviewer`** automatically attaches applet to window
    - In application, explicitly attach to content pane

```
77        applicationWindow.getContentPane().add( appletObject );
```

    - Default **`BorderLayout`**, **`CENTER`**
    - Occupies entire window

3.7 Designing Programs that Execute as Applets or Applications

- Display
  - Application window must be sized and displayed on screen
  - When windows displayed
    - Components receive calls to **`paint`** (or **`paintComponent`**)
    - Thus, applet's **`paint`** method gets called

```
80        applicationWindow.setSize( width, height );
84        applicationWindow.show();
```


```
Fig. 3.6: DrawShapes.java
 Draw random lines, rectangles and ovals
port java.awt.*;
port java.awt.event.*;
port javax.swing.*;

blic class DrawShapes extends JApplet {
 private JButton choices[];
 private String names[] = { "Line", "Rectangle", "Oval" };
 private JPanel buttonPanel;
 private DrawPanel drawingArea;
 private int width = 300, height = 200;

 public void init()
 {
    drawingArea = new DrawPanel( width, height );

    choices = new JButton[ names.length ];
    buttonPanel = new JPanel();
    buttonPanel.setLayout(
       new GridLayout( 1, choices.length ) );
    ButtonHandler handler = new ButtonHandler();

    for ( int i = 0; i < choices.length; i++ ) {
       choices[ i ] = new JButton( names[ i ] );
       buttonPanel.add( choices[ i ] );
       choices[ i ].addActionListener( handler );
    }
```

Program extends **JApplet**, can can execute as a normal applet.

Define **init** as normal - set up GUI components, buttons, etc.

Outline

wShapes
pplet)

1.1 Declarations

2. init

```
   Container c = getContentPane();
   c.add( buttonPanel, BorderLayout.NORTH );
   c.add( drawingArea, BorderLayout.CENTER );
}

public void setWidth( int w )
   { width = ( w >= 0 ? w : 300 ); }

public void setHeight( int h )
   { height = ( h >= 0 ? h : 200 ); }

public static void main( String args[] )
{
   int width, height;

   if ( args.length != 2 ) {  // no command-line arguments
      width = 300;
      height = 200;
   }
   else {
      width = Integer.parseInt( args[ 0 ] );
      height = Integer.parseInt( args[ 1 ] );
   }

   // create window in which applet will execute
   JFrame applicationWindow =
      new JFrame( "An applet running as an application" );

   applicationWindow.addWindowListener(
      new WindowAdapter() {
```

Define **main**, used by applications.

Test for command line arguments.

Explicitly create a window.

Outline

2.1 GUI

2.2 setWidth

2.3 setHeight

3.1 Command-line arguments

3.2 Create JFrame

3.3 Event handler

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

// create one applet instance
DrawShapes appletObject = new DrawShapes();
appletObject.setWidth( width );
appletObject.setHeight( height );

// call applet's init and start methods
appletObject.init();
appletObject.start();

// attach applet to center of window
applicationWindow.getContentPane().add( appletObject );

// set the window's size
applicationWindow.setSize( width, height );

// showing the window causes all GUI components
// attached to the window to be painted
applicationWindow.show();
}

private class ButtonHandler implements Ac
   public void actionPerformed( ActionEve
   {
      for ( int i = 0; i < choices.length; i++ )
```

Explicitly create an applet instance, call **`init`** and **`start`** methods.

Add **`appletObject`** to content pane.

Display window, which calls **`paint`** (or **`paintComponent`**) methods for each component.

```
            if ( e.getSource() == choices[ i ] ) {
               drawingArea.setCurrentChoice( i );
               break;
            }
      }
   }
```

```
// subclass of JPanel to allow drawing in a separate area
class DrawPanel extends JPanel {
   private int currentChoice = -1;  // don't draw first time
   private int width = 100, height = 100;

   public DrawPanel( int w, int h )
   {
      width = ( w >= 0 ? w : 100 );
      height = ( h >= 0 ? h : 100 );
   }

   public void paintComponent( Graphics g )
   {
      super.paintComponent( g );

      switch( currentChoice ) {
         case 0:
            g.drawLine( randomX(), randomY(),
                        randomX(), randomY() );
            break;
         case 1:
            g.drawRect( randomX(), randomY(),
```


5. Class `Drawpanel` (`extends JPanel`)

5.1 `paintComponent`

```
                  randomX(), randomY() );
         break;
      case 2:
         g.drawOval( randomX(), randomY(),
                     randomX(), randomY() );
         break;
   }
}

public void setCurrentChoice( int c )
{
   currentChoice = c;
   repaint();
}

private int randomX()
   { return (int) ( Math.random() * width ); }

private int randomY()
   { return (int) ( Math.random() * height ); }
```

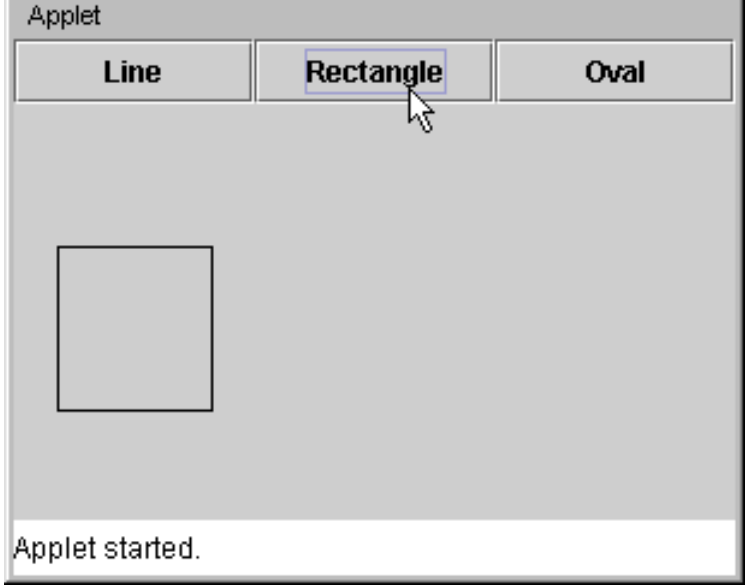
Outline

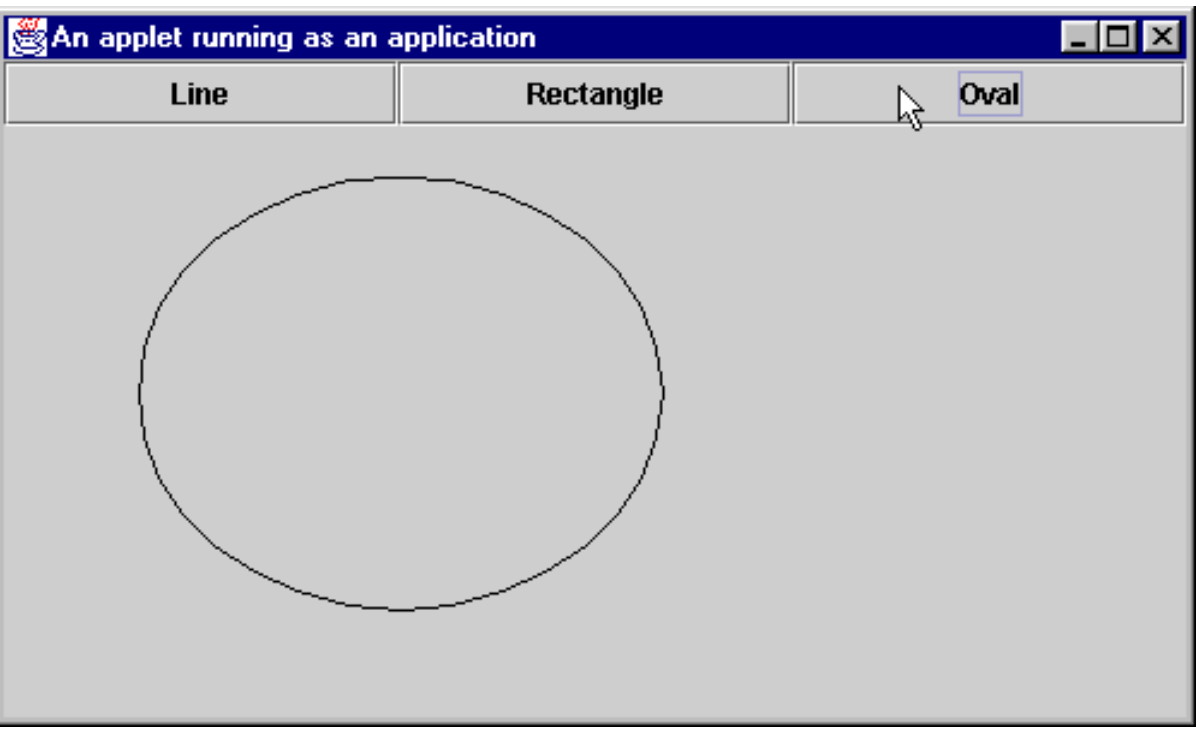
5.2 `setCurrentChoice`

5.3 `repaint`

5.4 `randomX`

5.5 `randomY`





Outline

Program Output

3.8 Using Menus with Frames

- Menus
  - Important part of GUIs
  - Perform actions without cluttering GUI
  - Attached to objects of classes that have method **`setJMenuBar`**
    - **`JFrame`** and **`JApplet`**
  - **`ActionEvent`**s
- Classes used to define menus
  - **`JMenuBar`** - container for menus, manages menu bar
  - **`JMenuItem`** - manages menu items
    - Menu items - GUI components inside a menu
    - Can initiate an action or be a submenu
    - Method **`isSelected`**

3.8 Using Menus with Frames

- Classes used to define menus (continued)
  - **`JMenu`** - manages menus
    - Menus contain menu items, and are added to menu bars
    - Can be added to other menus as submenus
    - When clicked, expands to show list of menu items
  - **`JCheckBoxMenuItem`** (extends **`JMenuItem`**)
    - Manages menu items that can be toggled
    - When selected, check appears to left of item
  - **`JRadioButtonMenuItem`** (extends **`JMenuItem`**)
    - Manages menu items that can be toggled
    - When multiple **`JRadioButtonMenuItem`**s are part of a group (**`ButtonGroup`**), only one can be selected at a time
    - When selected, filled circle appears to left of item

3.8 Using Menus with Frames

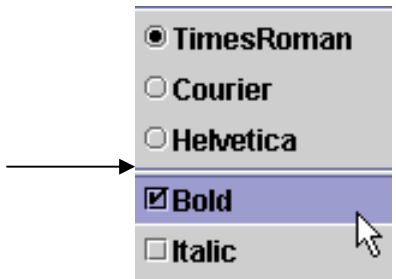
- Mnemonics
  - Quick access to menu items (File)
    - Can be used with classes that have subclass **`javax.swing.AbstractButton`**
  - Method **`setMnemonic`**

```
JMenu fileMenu = new JMenu( "File" )
fileMenu.setMnemonic( 'F' );
```

    - Press **`Alt + F`** to access menu
- Methods
  - **`setSelected( true )`**
    - Of class **`AbstractButton`**
    - Sets button/item to selected state

3.8 Using Menus with Frames

- Methods (continued)
  - **`addSeparator()`**
    - Of class **`JMenu`**
    - Inserts separator line into menu



- Dialog boxes
  - Modal - No other window can be accessed while it is open (default)
    - Modeless - other windows can be accessed

3.8 Using Menus with Frames

- Dialog boxes
  - **`JOptionPane.showMessageDialog( parentWindow, text, title, messageType )`**
  - **`parentWindow`** - determines where dialog box appears
    - **`null`** - displayed at center of screen
    - Window specified - dialog box centered horizontally over parent

```
32                JOptionPane.showMessageDialog( MenuTest.this,
33                   "This is an example\nof using menus",
34                   "About", JOptionPane.PLAIN_MESSAGE );
```


3.8 Using Menus with Frames

- **`AbstractButton`** methods
  - Superclass of **`MenuItem`**
  - **`isSelected`**
    - Returns **`boolean`**
  - **`getText`**
    - Returns **`String`** containing name

3.8 Using Menus with Frames

- Using menus
  - Create menu bar
    - Set menu bar for **`JFrame`**
      - **`setJMenuBar( myBar );`**
  - Create menus
    - Set Mnemonics
  - Create menu items
    - Set Mnemonics
    - Set event handlers
  - If using **`JRadioButtonMenuItem`**s
    - Create a group: **`myGroup = new ButtonGroup();`**
    - Add **`JRadioButtonMenuItem`**s to the group

3.8 Using Menus with Frames

- Using menus (continued)
  - Add menu items to appropriate menus
    - **`myMenu.add( myItem );`**
    - Insert separators if necessary: **`myMenu.addSeparator();`**
  - If creating submenus, add submenu to menu
    - **`myMenu.add( mySubMenu );`**
  - Add menus to menu bar
    - **`myMenuBar.add( myMenu );`**
- Example
  - Use menus to alter text in a **`JLabel`**
  - Change color, font, style
  - Have a "File" menu with a "About" and "Exit" items

```
Fig. 3.7: MenuTest.java
 Demonstrating menus
port javax.swing.*;
port java.awt.event.*;
port java.awt.*;

blic class MenuTest extends JFrame {
 private Color colorValues[] =
    { Color.black, Color.blue, Color.red, Color.green };
 private JRadioButtonMenuItem colorItems[], fonts[];
 private JCheckBoxMenuItem styleItems[];
 private JLabel display;
 private ButtonGroup fontGroup, colorGroup;
 private int style;

 public MenuTest()
 {
    super( "Using JMenus" );

    JMenuBar bar = new JMenuBar();  // create menubar
    setJMenuBar( bar );  // set the menubar fo

    // create File menu and Exit menu item
    JMenu fileMenu = new JMenu( "File" );
    fileMenu.setMnemonic( 'F' );
    JMenuItem aboutItem = new JMenuItem( "About..." );
    aboutItem.setMnemonic( 'A' );
    aboutItem.addActionListener(
       new ActionListener() {
          public void actionPerformed( ActionEvent e )
```

Create menu bar, set menu bar for **`JFrame`**

Create **File** menu and set mnemonic.

Anonymous inner class for event handler.

Outline

1. `import`

1.1 `extends JFrame`

1.2 Declare objects

1.3 Create menubar

1.4 Set menubar for `JFrame`

2. Create File menu

2.1 Create menu item

2.2 Event handler

```
         {
            JOptionPane.showMessageDialog( MenuTest.this,
               "This is an example\nof using menus",
               "About", JOptionPane.PLAIN_MESSAGE );
         }
      }
   );
   fileMenu.add( aboutItem );

   JMenuItem exitItem = new JMenuItem( "Exit" );
   exitItem.setMnemonic( 'x' );
   exitItem.addActionListener(
      new ActionListener() {
         public void actionPerformed( ActionEvent e )
         {
            System.exit( 0 );
         }
      }
   );
   fileMenu.add( exitItem );
   bar.add( fileMenu );    // add File menu

   // create the Format menu, its submenus and menu items
   JMenu formatMenu = new JMenu( "Format" );
   formatMenu.setMnemonic( 'r' );

   // create Color submenu
   String colors[] =
      { "Black", "Blue", "Red", "Green" };
   JMenu colorMenu = new JMenu( "Color" );
```

Add **aboutItem** to **fileMenu**.

Create **exitItem**, set mnemonic, define event handler.

Add **fileMenu** to the menubar.

Outline

2.2 Event handler

d menu item

nu item

dler

2.6 Add menu item

2.7 Add menu to menubar

3. Create Format menu menu

```
colorItems = new JRadioButtonMenuItem[ colors.length ];

colorGroup = new ButtonGroup();
ItemHandler itemHandler = new ItemHandler();

for ( int i = 0; i < colors.length; i++ ) {
   colorItems[ i ] =
      new JRadioButtonMenuItem( colors[ i ] );
   colorMenu.add( colorItems[ i ] );
   colorGroup.add( colorItems[ i ] );
   colorItems[ i ].addActionListener( itemHandler );
}

colorItems[ 0 ].setSelected( true );
formatMenu.add( colorMenu );
formatMenu.addSeparator();

// create Font submenu
String fontNames[] =
   { "TimesRoman", "Courier", "Helvetica" };
JMenu fontMenu = new JMenu( "Font" );
fontMenu.setMnemonic( 'n' );
fonts = new JRadioButtonMenuItem[ fontName
fontGroup = new ButtonGroup();

for ( int i = 0; i < fonts.length; i++ ) {
   fonts[ i ] =
      new JRadioButtonMenuItem( fontNames[ i ] );
   fontMenu.add( fonts[ i ] );
   fontGroup.add( fonts[ i ] );
```

Create a new **ButtonGroup** to use with colors.

Create instance of event handler (named inner class).

Loop, create menu items, add to **colorMenu**, add to **colorGroup** register event handler.

Create Font submenu: loop, add to **fontMenu**, add to **fontGroup**, register event handler (next slide).

Outline

tems

to

3.3 Event handler

mat

menu

4. Font submenu

```
   fonts[ i ].addActionListener( itemHandler );
}

fonts[ 0 ].setSelected( true );
fontMenu.addSeparator();

String styleNames[] = { "Bold", "Italic" };
styleItems = new JCheckBoxMenuItem[ styleNames.length ];
StyleHandler styleHandler = new StyleHandler();

for ( int i = 0; i < styleNames.length; i++ ) {
   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 );

setSize( 500, 200 );
```

Add style items (separate event handler).

4.1 Create and add menu items

4.2 Add Font submenu to Format menu

(text ied)

5.1 Add to content pane

```
   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 ) );
```

Outline

6. `main`

7. Event handler

7.1 `getText`

Checks to see which item was selected, changes **`display`** accordingly.

Returns name of menu item.

Outline

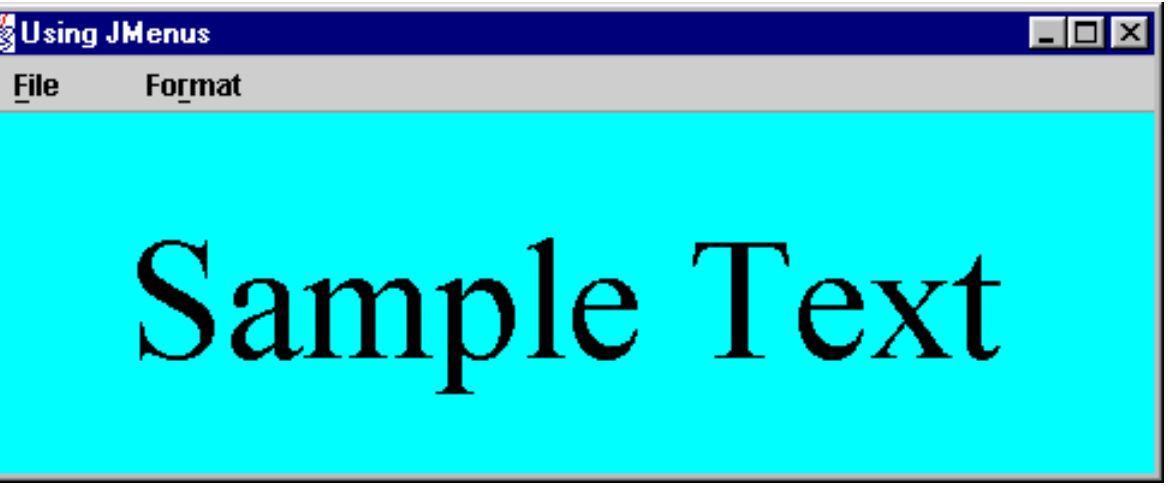
8. Event handler

```
            break;
         }

      repaint();
   }
}

class StyleHandler implements ItemListe
   public void itemStateChanged( ItemEv
   {
      style = 0;

      if ( styleItems[ 0 ].isSelected() )
         style += Font.BOLD;

      if ( styleItems[ 1 ].isSelected() )
         style += Font.ITALIC;

      display.setFont( new Font(
         display.getFont().getName(), style, 72 ) );

      repaint();
   }
}
```

Event handler for style changes. Change style depending on which items are selected.

Outline

Program Output



3.9 Using JPopupMenus

- Context-sensitive popup menus
  - Created with class **JPopupMenu**
    - Subclass of **JComponent**
  - Options specific to component
    - Popup trigger event
    - Most systems, right mouse click
- Example program
  - Create **JPopupMenu** to select background color
  - On right mouse click, JPopupMenu displayed
    - If radio button clicked, **actionPerformed** changes background

3.9 Using JPopupMenus

```
16       final JPopupMenu popupMenu = new JPopupMenu();
```

– Local variables must be declared final to be used in anonymous inner classes
  - Anonymous inner class event handler for window

```
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 );
```

– Items added as normal

3.9 Using JPopupMenus

```
35       addMouseListener(
36          new MouseAdapter() {
37             public void mousePressed( MouseEvent e )
38                { checkForTriggerEvent( e ); }
39
40             public void mouseReleased( MouseEvent e )
41                { checkForTriggerEvent( e ); }
```

- Anonymous inner class event handler
  - Handles mouse events

```
43             private void checkForTriggerEvent( MouseEvent e )
44             {
45                if ( e.isPopupTrigger() )
46                   popupMenu.show( e.getComponent(),
47                                   e.getX(), e.getY() );
48             }
```

- **`isPopupTrigger`**
  - Returns **`true`** if popup-trigger event occurred

3.9 Using JPopupMenus

```
43             private void checkForTriggerEvent( MouseEvent e )
44             {
45                if ( e.isPopupTrigger() )
46                   popupMenu.show( e.getComponent(),
47                                   e.getX(), e.getY() );
48             }
```

- **show (component, x, y)**
  - Displays JPopupMenu
  - **component**
    - Origin component, helps determine location of **JPopupMenu**
    - **getComponent** - component that had event
  - **x,y**
    - Coordinates (with respect to origin component) to display **JPopupMenu**

```
Fig. 13.8: PopupTest.java
 Demonstrating JPopupMenus
port javax.swing.*;
port java.awt.event.*;
port java.awt.*;

blic class PopupTest extends JFrame {
 private JRadioButtonMenuItem items[];
 private Color colorValues[] =
    { Color.blue, Color.yellow, Color.red };

 public PopupTest()
 {
    super( "Using JPopupMenus" );

    final JPopupMenu popupMenu = new JPopupMenu();
    ItemHandler handler = new ItemHandler();
    String colors[] = { "Blue", "Yellow", "Red" };
    ButtonGroup colorGroup = new ButtonGroup();
    items = new JRadioButtonMenuItem[ 3 ];

    // construct each menu item and add to popup menu; also
    // enable event handling for each menu item
    for ( int i = 0; i < items.length; i++ ) {
       items[ i ] = new JRadioButtonMenuItem( colors[ i ] );
       popupMenu.add( items[ i ] );
       colorGroup.add( items[ i ] );
       items[ i ].addActionListener( handler );
    }
```

Declare **`JPopupMenu final`**, so it may be used in anonymous inner class.

Create and add menu items as usual.

Outline

1. Class `PopupTest`

1.1 `JPopupMenu`

```
   getContentPane().setBackground( co
   // define a MouseListener for the w
   // a JPopupMenu when the popup trig
   addMouseListener(
      new MouseAdapter() {
         public void mousePressed( MouseEvent e )
            { checkForTriggerEvent( e ); }

         public void mouseReleased( MouseEvent e )
            { checkForTriggerEvent( e ); }

         private void checkForTriggerEvent( MouseEvent e )
         {
            if ( e.isPopupTrigger() )
               popupMenu.show( e.getComponent(),
                               e.getX(), e.getY() );
         }
      }
   );

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

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

Anonymous inner class for mouse events. Note use of **`isPopupTrigger`** and **`show`**.

2. Event handler

2.1 `show`

2.2 `getComponent`

3. `main`

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

private class ItemHandler implements ActionListener {
   public void actionPerformed( ActionEvent e )
   {
      // determine which menu item was selected
      for ( int i = 0; i < items.length; i++ )
         if ( e.getSource() == items[ i ] ) {
            getContentPane().setBackground(
               colorValues[ i ] );
            repaint();
            return;
         }
   }
}
```

Event handler for the menu items (generate **`ActionEvent`**s).

3.10 Pluggable Look-and-Feel

- Look and feel
  - Programs using Abstract Windowing Toolkit (**`java.awt`**) have same look and feel as platform
  - Lightweight GUI components
    - Uniform functionality across platforms (metal look and feel)
    - Can customize for UNIX or Windows look and feel
- Class **`UIManager`** (**`javax.swing`**)
  - **`public static`** inner class **`LookAndFeelInfo`**
    - Has look and feel info

```
 9    private UIManager.LookAndFeelInfo looks[];
51       looks = UIManager.getInstalledLookAndFeels();
```

    - **`getInstalledLookandFeels`**
      - Returns array of objects describing installed look and feels

3.10 Pluggable Look-and-Feel

```
62          UIManager.setLookAndFeel(
63             looks[ value ].getClassName() );
```

- **`static`** method **`setLookandFeel`**
  - Changes look and feel
- **`getClassName`**
  - Determines name corresponding to **`UIManager.LookAndFeelInfo`**

```
64          SwingUtilities.updateComponentTreeUI( this );
```

- Update the look and feel of every component attached to argument

```
61       try {
66       catch ( Exception e ) {
```

- Blocks used for error-handling (more next chapter)

```
Fig. 13.9: LookAndFeelDemo.java
 Changing the look and feel.
port javax.swing.*;
port java.awt.*;
port java.awt.event.*;

blic class LookAndFeelDemo extends JFrame {
 private String strings[] = { "Metal", "Motif", "Windows" };
 private UIManager.LookAndFeelInfo looks[];
 private JRadioButton radio[];
 private ButtonGroup group;
 private JButton button;
 private JLabel label;
 private JComboBox comboBox;

 public LookAndFeelDemo()
 {
    super( "Look and Feel Demo" );

    Container c = getContentPane();

    JPanel northPanel = new JPanel();
    northPanel.setLayout( new GridLayout( 3, 1, 0, 5 ) );
    label = new JLabel( "This is a Metal look-and-feel",
                        SwingConstants.CENTER );
    northPanel.add( label );
    button = new JButton( "JButton" );
    northPanel.add( button );
    comboBox = new JComboBox( strings );
    northPanel.add( comboBox );
```


1. Class LookAndFeelDemo

1.1 Declarations

1.2 GUI

```
   c.add( northPanel, BorderLayout.NORTH );

   JPanel southPanel = new JPanel();
   radio = new JRadioButton[ strings.length ];
   group = new ButtonGroup();
   ItemHandler handler = new ItemHandler();
   southPanel.setLayout(
      new GridLayout( 1, radio.length ) );

   for ( int i = 0; i < radio.length; i++ ) {
      radio[ i ] = new JRadioButton( strings[ i ] );
      radio[ i ].addItemListener( handler );
      group.add( radio[ i ] );
      southPanel.add( radio[ i ] );
   }

   c.add( southPanel, BorderLayout.SOUTH );

   // get the installed look-and-feel information
   looks = UIManager.getInstalledLookAndFeels();

   setSize( 300, 200 );
   show();

   radio[ 0 ].setSelected( true );
}
```

Put objects into **looks** array.

Outline

1.3 Buttons

1.4 getIntalledLookAndFeels

```
private void changeTheLookAndFeel( int value )
{
   try {
      UIManager.setLookAndFeel(
         looks[ value ].getClassName() );
      SwingUtilities.updateComponentTreeUI( this );
   }
   catch ( Exception e ) {
      e.printStackTrace();
   }
}

public static void main( String args[] )
{
   LookAndFeelDemo dx = new LookAndFeelDemo();

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

Set the appropriate look and feel.

Update components for new Look and Feel.

Outline

2. changeTheLookAndFeel

2.2 updateComponentTreeUI

3. main

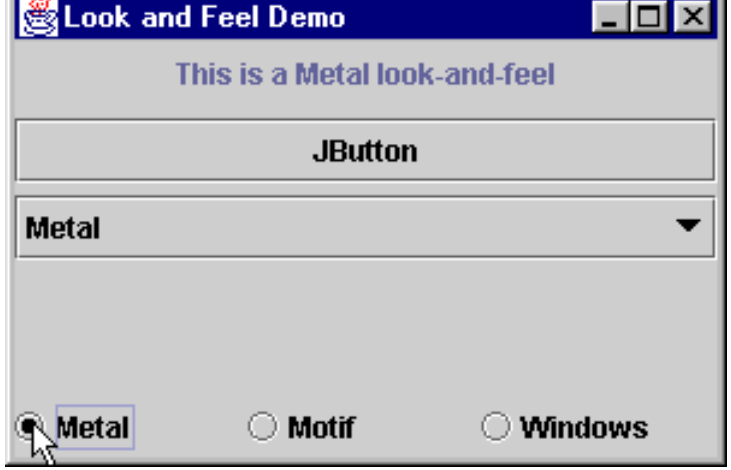
```
private class ItemHandler implements ItemListener {
   public void itemStateChanged( ItemEvent e )
   {
      for ( int i = 0; i < radio.length; i++ )
         if ( radio[ i ].isSelected() ) {
            label.setText( "This is a " +
               strings[ i ] + " look-and-feel" );
            comboBox.setSelectedIndex( i );
            changeTheLookAndFeel( i );
         }
   }
}
```

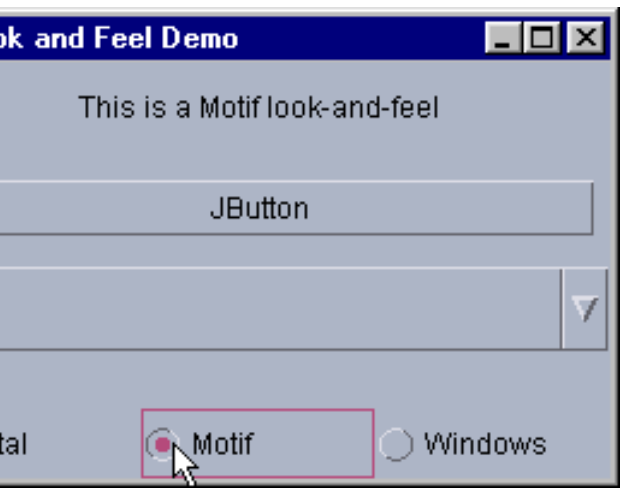
Outline

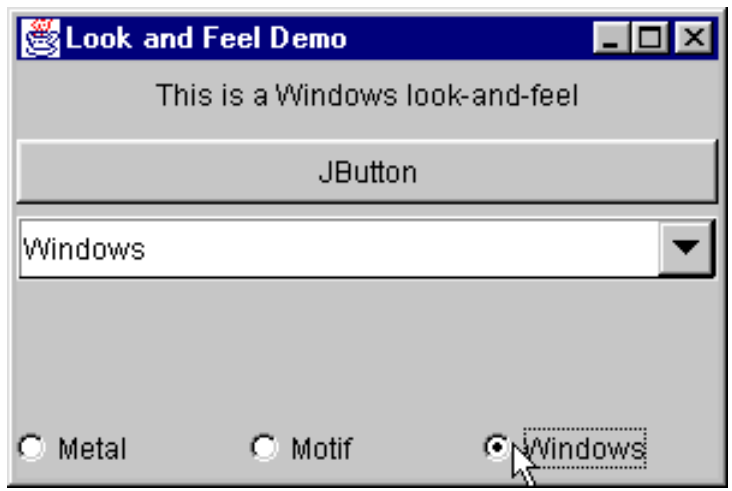
4. Event handler

Outline

Program Output







3.11 Using JDesktopPane and JInternalFrame

- Multiple document interface (MDI)
  - Parent window containing child windows
  - Manage several open documents, all being processed
    - Email programs, word processors
    - Switch between documents without closing
  - Class **JDesktopPane** (**javax.swing**) and **JInternalFrame** support MDIs

```
final JDesktopPane theDesktop = new JDesktopPane();
getContentPane().add( theDesktop );
```

  - Used to manage **JInternalFrame** child windows
  - **final**, can be used in anonymous inner classes

3.11 Using JDesktopPane and JInternalFrame

```
25                JInternalFrame frame =
26                   new JInternalFrame(
27                      "Internal Frame",
28                      true, true, true, true );
```

- **JInternalFrame** constructor
  - Title, resizable, closable, maximizable, minimizable

```
30                Container c = frame.getContentPane();
```

- Like **JFrame** and **JApplet**, has content pane to attach components

```
37                frame.setOpaque( true );
38                theDesktop.add( frame );
```

- **setOpaque** - default **false** (transparent)
  - Windows and icons visible through background
- **add** - adds **JInternalFrame** to **JDesktopPane**

3.11 Using JDesktopPane and JInternalFrame

- Upcoming program
  - Have a menu with an add button
    - Adds **JInternalFrame**s to **JDesktopPane**
  - Add custom subclass of **JPanel** to **JInternalFrame**
    - Has icon

```
 Demonstrating JDesktopPane.
port javax.swing.*;
port java.awt.event.*;
port java.awt.*;

blic class DesktopTest extends JFrame {
 public DesktopTest()
 {
    super( "Using a JDesktopPane" );

    JMenuBar bar = new JMenuBar();
    JMenu addMenu = new JMenu( "Add" );
    JMenuItem newFrame = new JMenuItem( "Internal
    addMenu.add( newFrame );
    bar.add( addMenu );
    setJMenuBar( bar );

    final JDesktopPane theDesktop = new JDesktopPane();
    getContentPane().add( theDesktop );

    newFrame.addActionListener(
       new ActionListener() {
          public void actionPerformed( A
             JInternalFrame frame =
                new JInternalFrame(
                   "Internal Frame",
                   true, true, true, true );
```

Create **JDesktopPane** and add it to the window's content pane.

Create a new **JInternalFrame** to add to the **JDesktopPane**.

Outline

1. Class DesktopTest

1.1 JDesktopPane

2. Event handler

2.1 JInternalFrame

```
            MyJPanel panel = new MyJPanel();

            c.add( panel, BorderLayout.CENTER );
            frame.setSize(
               panel.getImageWidthHeight().width,
               panel.getImageWidthHeight().height );
            frame.setOpaque( true );
            theDesktop.add( frame );
         }
      }
   );

   setSize( 500, 400 );
   show();
}

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

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

Add **MyJPanel** to new **JInternalFrame**.

2.3 MyJPanel

2.4 add

2.5 setSize

2.6 theDesktop.add

3. main

```
class MyJPanel extends JPanel {
   private ImageIcon imgIcon;

   public MyJPanel()
   {
      imgIcon = new ImageIcon( "jhtp3.gif" );
   }

   public void paintComponent( Graphics g )
   {
      imgIcon.paintIcon( this, g, 0, 0 );
   }

   public Dimension getImageWidthHeight()
   {
      return new Dimension( imgIcon.getIconWidth(),
                            imgIcon.getIconHeight() );
   }
```

Outline

4. Class `MyJPanel`

4.1 Override `paintComponent`

3.12 Layout Managers

- Previous chapter
  - **`FlowLayout`**, **`BorderLayout`**, **`GridLayout`**
- This chapter
  - Three additional layout managers
    - **`BoxLayout`**, **`CardLayout`**, **`GridBagLayout`**

3.13 BoxLayout Layout Manager

- BoxLayout
  - Arranges components horizontally (left-right) or vertically (top-bottom)
  - Container class **Box**
    - Uses **BoxLayout** as default layout manager

```
16       Box boxes[] = new Box[ 4 ];
18       boxes[ 0 ] = Box.createHorizontalBox();
19       boxes[ 1 ] = Box.createVerticalBox();
20       boxes[ 2 ] = Box.createHorizontalBox();
21       boxes[ 3 ] = Box.createVerticalBox();
```

  - Initialize array of **Box**es
    - **createHorizontalBox** and **createVerticalBox** return **Box** container with horizontal or vertical **BoxLayout**

3.13 BoxLayout Layout Manager

```
29          boxes[ 1 ].add( Box.createVerticalStrut( 25 ) );
30          boxes[ 1 ].add( new JButton( "boxes[1]: " + i ) );
```

- **createVerticalStrut(height)**
  - Adds vertical strut, invisible component used for fixed spacing between components
  - Also, **createHorizontalStrut**

```
35          boxes[ 2 ].add( Box.createHorizontalGlue() );
36          boxes[ 2 ].add( new JButton( "boxes[2]: " + i ) );
```

- Horizontal glue
  - Occupies additional space between components
  - Normally, extra space added onto last component
  - **createVerticalGlue**

3.13 BoxLayout Layout Manager

```
41          boxes[ 3 ].add(
42             Box.createRigidArea( new Dimension( 12, 8 ) ) );
43          boxes[ 3 ].add( new JButton( "boxes[3]: " + i ) );
```

- Creates rigid area between components
  - Invisible GUI component, fixed pixel width and height
  - **`static`** method **`createRigidArea`** takes a **`Dimension`** object

```
47          JPanel panel = new JPanel();
48          panel.setLayout(
49             new BoxLayout( panel, BoxLayout.Y_AXIS ) );
```

- Set layout using setLayout
  - **`BoxLayout`** constructor: takes container which it controls layout, and **`BoxLayout.X_AXIS`** or **`BoxLayout.Y_AXIS`**

3.13 BoxLayout Layout Manager

```
52          panel.add( Box.createGlue() );
```

- **Box.createGlue()**
  - Adds a glue component to container

```
Fig. 3.12: BoxLayoutDemo.java
 Demonstrating BoxLayout.
port javax.swing.*;
port java.awt.*;
port java.awt.event.*;

blic class BoxLayoutDemo extends JFrame {
 public BoxLayoutDemo()
 {
    super( "Demostrating BoxLayout" );
    final int SIZE = 3;

    Container c = getContentPane();
    c.setLayout( new BorderLayout( 30, 30 ) );

    Box boxes[] = new Box[ 4 ];

    boxes[ 0 ] = Box.createHorizontalBox();
    boxes[ 1 ] = Box.createVerticalBox();
    boxes[ 2 ] = Box.createHorizontalBox();
    boxes[ 3 ] = Box.createVerticalBox();

    // add buttons to boxes[ 0 ]
    for ( int i = 0; i < SIZE; i++ )
       boxes[ 0 ].add( new JButton( "boxes[0]: " + i ) );

    // create strut and add buttons to boxes[ 1 ]
    for ( int i = 0; i < SIZE; i++ ) {
       boxes[ 1 ].add( Box.createVerticalStrut( 25 ) );
       boxes[ 1 ].add( new JButton( "boxes[1]: " + i ) );
    }
```

Create vertical strut (fixed spacing) between buttons.

Outline

1. Class BoxLayoutDemo

1.1 Box array

1.2 Add buttons

1.3 createVerticalStrut

```
// create horizontal glue and add buttons to boxes[ 2 ]
for ( int i = 0; i < SIZE; i++ ) {
   boxes[ 2 ].add( Box.createHorizontalGlue() );
   boxes[ 2 ].add( new JButton( "boxes[2]: " + i ) );
}

// create rigid area and add buttons to boxes
for ( int i = 0; i < SIZE; i++ ) {
   boxes[ 3 ].add(
      Box.createRigidArea( new Dimension( 12, 8 ) ) );
   boxes[ 3 ].add( new JButton( "boxes[3]: " + i ) );
}

// create horizontal glue and add buttons to panel
JPanel panel = new JPanel();
panel.setLayout(
   new BoxLayout( panel, BoxLayout.Y_AXIS ) );

for ( int i = 0; i < SIZE; i++ ) {
   panel.add( Box.createGlue() );
   panel.add( new JButton( "panel: " + i ) );
}

// place panels on frame
c.add( boxes[ 0 ], BorderLayout.NORTH );
c.add( boxes[ 1 ], BorderLayout.EAST );
c.add( boxes[ 2 ], BorderLayout.SOUTH );
c.add( boxes[ 3 ], BorderLayout.WEST );
c.add( panel, BorderLayout.CENTER );

setSize( 350, 300 );
```

Add glue between components (even spacing).

Create a rigid (fixed) area between components

Use the **BoxLayout** layout manager and add components.

Add **Box**es to content pane, in separate regions.

Outline

1.4

1.5 createRigidArea

2. JPanel

2.2 createGlue

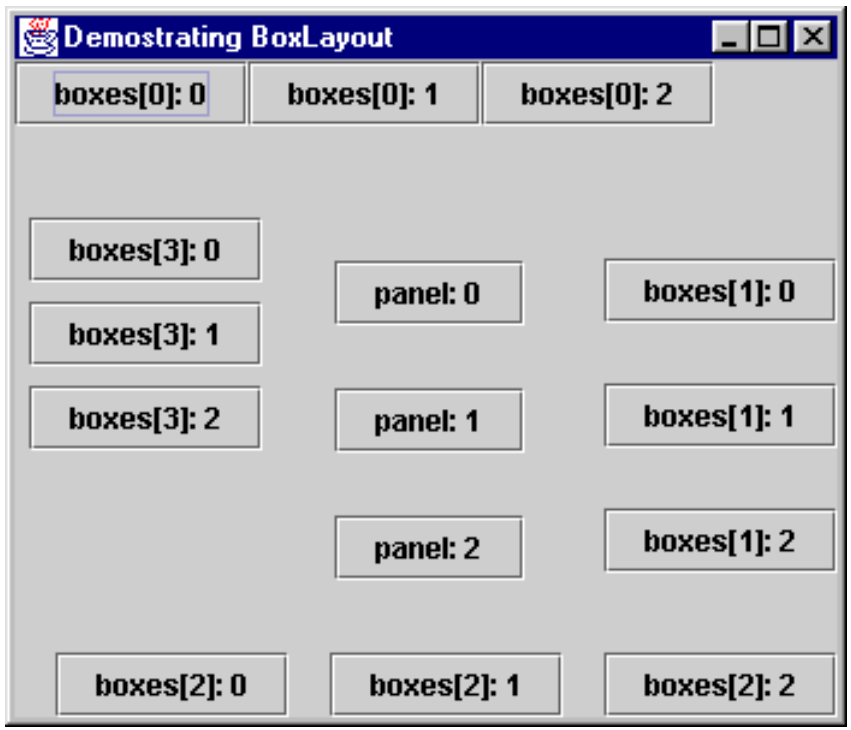
```
   show();
}

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

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

Outline

3. `main`

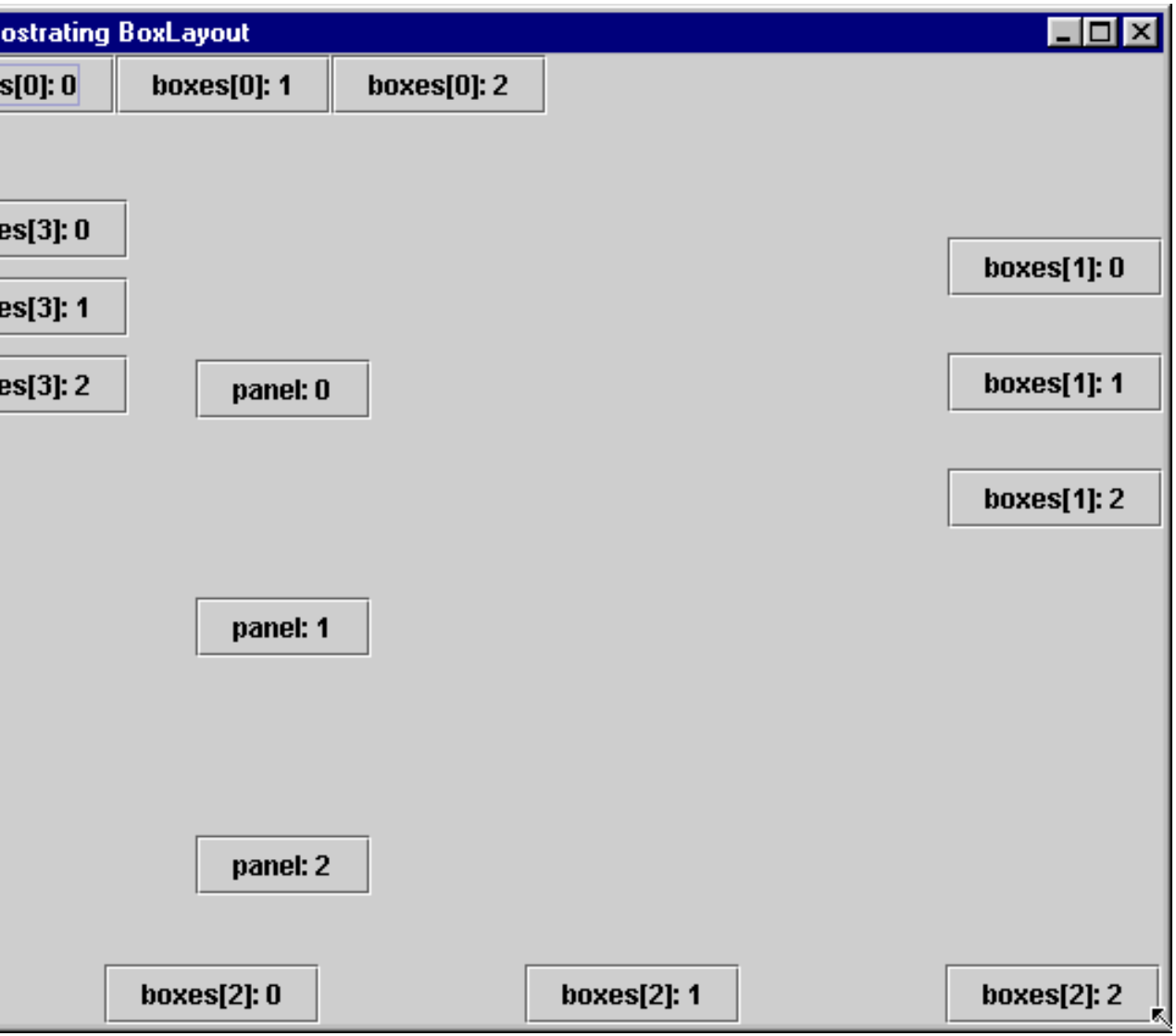
Outline

Program Output



Outline

Program Output



3.14 `CardLayout` Layout Manager

- **`CardLayout`**
  - Arrange components into "deck" of cards
    - Only top card visible
  - Any card can be placed on top using methods
  - Each card usually a container
    - Can use its any layout manager
- Usage
  - Create a **`CardLayout`** object
    - Use it as the layout manager for a container

```
22       deck = new JPanel();
23       cardManager = new CardLayout();
24       deck.setLayout( cardManager );
```


3.14 `CardLayout` Layout Manager

- Add "cards" to the container
  - **`add`** method with identifier

```
31        deck.add( card1, label1.getText() ); // add card to deck
```

- Methods **`first`**, **`next`**, **`previous`**, **`last`**
  - Each take **`Container`** for which to display card
  - Display first, next, previous, or last card in "deck"

```
74          cardManager.first( deck ); // show first card

76          cardManager.next( deck );  // show next card

78          cardManager.previous( deck );  // show previous card

80          cardManager.last( deck );  // show last card
```


```
 Demonstrating CardLayout.
port javax.swing.*;
port java.awt.*;
port java.awt.event.*;

blic class CardDeck extends JFrame
                    implements ActionListener {
 private CardLayout cardManager;
 private JPanel deck;
 private JButton controls[];
 private String names[] = { "First card", "Next card",
                            "Previous card", "Last card" };

 public CardDeck()
 {
    super( "CardLayout " );

    Container c = getContentPane();

    // create the JPanel with CardLayout
    deck = new JPanel();
    cardManager = new CardLayout();
    deck.setLayout( cardManager );

    // set up card1 and add it to JPanel deck
    JLabel label1 =
       new JLabel( "card one", SwingConstants.CENTER );
    JPanel card1 = new JPanel();
    card1.add( label1 );
    deck.add( card1, label1.getText() ); // add card to deck
```

Create a **JPanel** and set the layout manager.

Create a new **JPanel** to use as a card. Place a **JLabel** inside it, and add to **deck**.

Outline

1. Class CardDeck

1.1 JPanel

1.2 setLayout

1.3 JPanel

1.4 add card1

1.5 add to deck

Outline

2. Other cards

2.1 Layout buttons

```
// set up card2 and add it to JPanel deck
JLabel label2 =
   new JLabel( "card two", SwingConstants.CENTER );
JPanel card2 = new JPanel();
card2.setBackground( Color.yellow );
card2.add( label2 );
deck.add( card2, label2.getText() ); // add card to deck

// set up card3 and add it to JPanel deck
JLabel label3 = new JLabel( "card three" );
JPanel card3 = new JPanel();
card3.setLayout( new BorderLayout() );
card3.add( new JButton( "North" ), BorderLayout.NORTH );
card3.add( new JButton( "West" ), BorderLayout.WEST );
card3.add( new JButton( "East" ), BorderLayout.EAST );
card3.add( new JButton( "South" ), BorderLayout.SOUTH );
card3.add( label3, BorderLayout.CENTER );
deck.add( card3, label3.getText() ); // add card to deck

// create and layout buttons that will control deck
JPanel buttons = new JPanel();
buttons.setLayout( new GridLayout( 2, 2 ) );
controls = new JButton[ names.length ];

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

Create and add two more cards to **deck**.

Create **JPanel** to hold control buttons.

```
   // add JPanel deck and JPanel buttons to the applet
   c.add( buttons, BorderLayout.WEST );
   c.add( deck, BorderLayout.EAST );

   setSize( 450, 200 );
   show();
}

public void actionPerformed( ActionEvent e )
{
   if ( e.getSource() == controls[ 0 ] )
      cardManager.first( deck ); // show first card
   else if ( e.getSource() == controls[ 1 ] )
      cardManager.next( deck );  // show next card
   else if ( e.getSource() == controls[ 2 ] )
      cardManager.previous( deck );  // show previous card
   else if ( e.getSource() == controls[ 3 ] )
      cardManager.last( deck );  // show last card
}

public static void main( String args[] )
{
   CardDeck cardDeckDemo = new CardDeck();

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

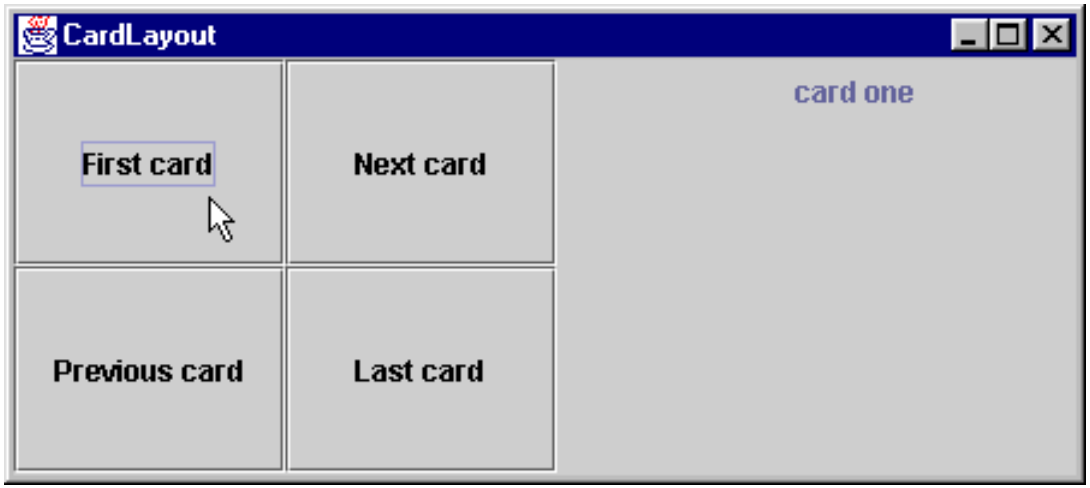
Add control **`JPanel`** and **`deck`** to content pane.

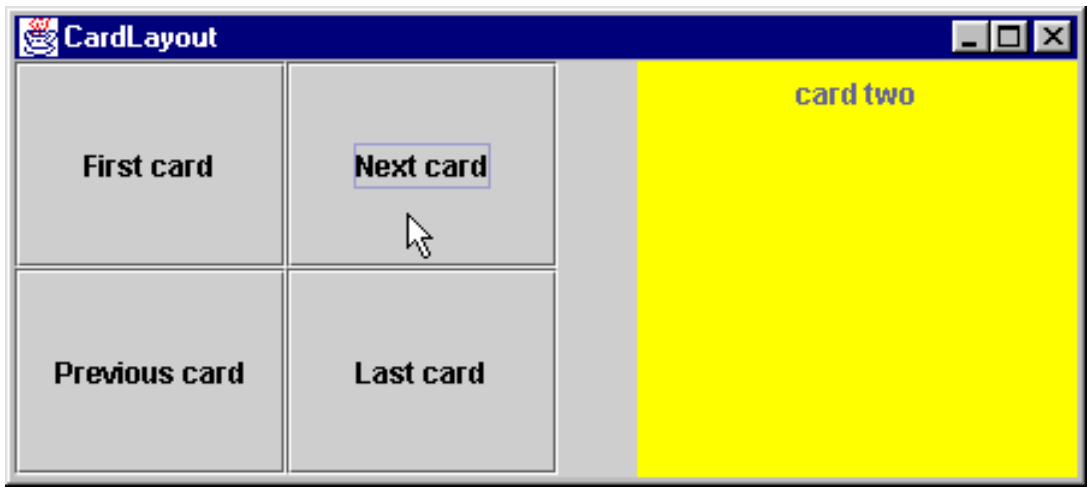
Call **`first`**, **`next`**, **`previous`**, or **`last`**.

Outline

4. `main`

```
    );
}
```

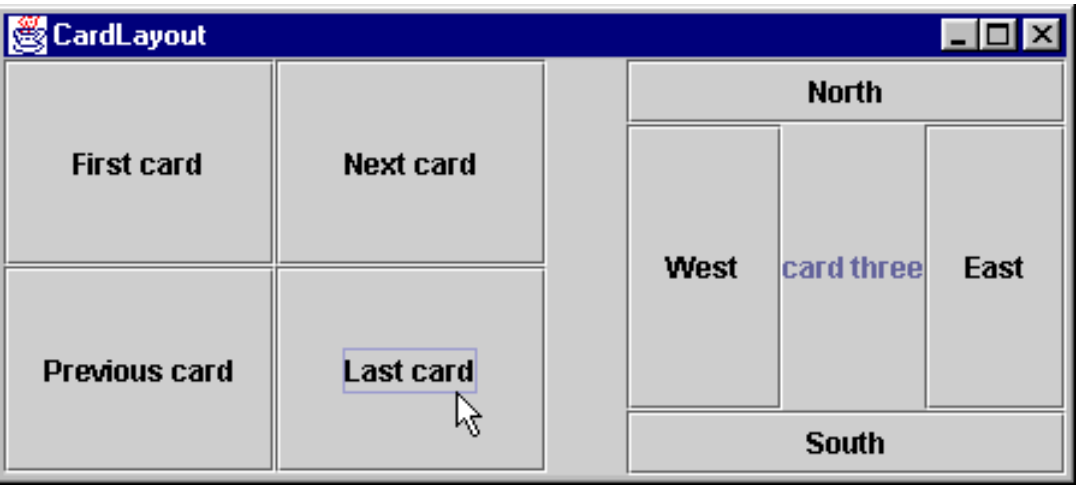




Outline

Program Output

Outline



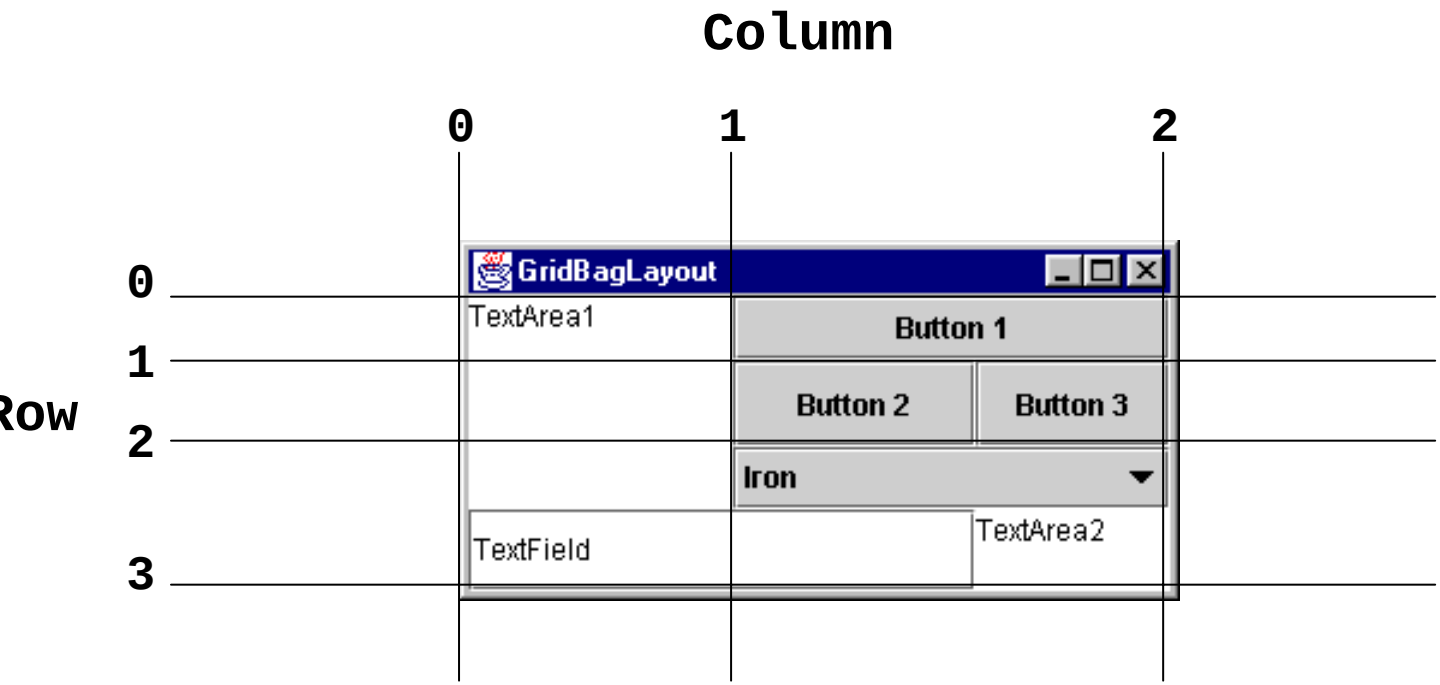
Program Output

3.15 `GridBagLayout` Layout Manager

- Most complex and powerful layout manager
  - Arrange components in grid (similar to **`GridLayout`**)
  - Flexible - components can vary in size
    - Occupy multiple rows or columns
    - Added in any order

3.15 GridBagLayout Layout Manager

- Usage
  - First, draw GUI on paper
  - Divide into grid, starting with row 0 and column 0
    - Used to properly place components

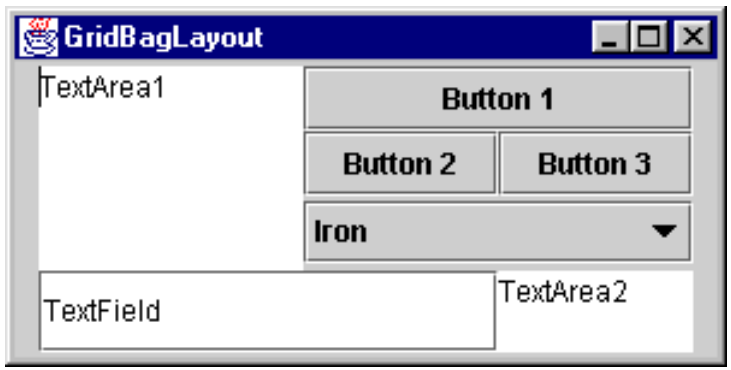


3.15 `GridBagLayout` Layout Manager

- Usage
  - Construct a **`GridBagConstraints`** object
    - Specifies how components are placed
  - Instance variables
    - **`gridx`** - column
    - **`gridy`** - row
    - **`gridwidth`** - number of columns occupied
    - **`gridheight`** - number of rows occupied
    - **`weightx`** - extra horizontal space, if resized
    - **`weighty`** - extra vertical space
  - Should have positive weight values, otherwise components "huddle" in center (0 default)

3.15 `GridBagLayout` Layout Manager

- Usage
  - Weights set to zero (components huddle)



  - **`GridBagConstraints`** instance variable **`fill`**
    - Specifies if component can grow if window resized
    - Assign **`fill`** a **`GridBagConstraints`** constant
      - **`NONE`** (default), **`VERTICAL`**, **`HORIZONTAL`**, **`BOTH`**

3.15 GridBagLayout Layout Manager

- Usage
  - Instance variable **anchor**
    - Location of component if does not fill entire area
    - Assign **anchor** a **GridBagConstraints** constant
    - **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST**, **NORTHWEST**, **CENTER** (default)
  - Method **setConstraints** (class **GridBagLayout**)
    - Takes **Component** and **GridBagConstraints** argument

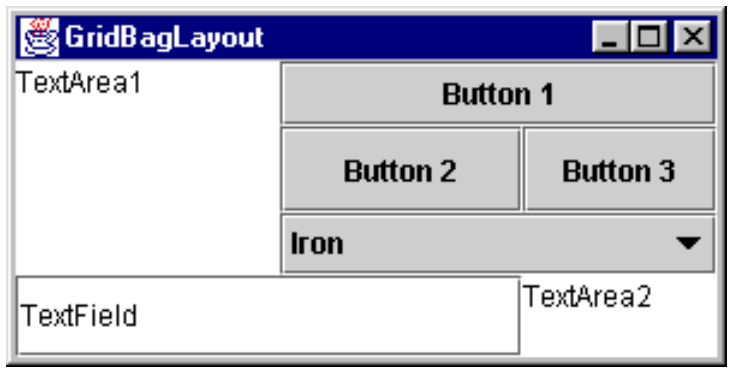
```
85        gbLayout.setConstraints( c, gbConstraints );
```

  - After constraints set, add component to content pane

```
86        container.add( c );      // add component
```


3.15 `GridBagLayout` Layout Manager

- Example program
  - Create the GUI shown



  - Define method **`addComponents`** to set the instance variables of a **`GridBagConstaints`** object
    - Then, call **`setConstraints`** and add component to content pane

```
Fig. 3.17: GridBagDemo.java
 Demonstrating GridBagLayout.
port javax.swing.*;
port java.awt.*;
port java.awt.event.*;

blic class GridBagDemo extends JFrame {
 private Container container;
 private GridBagLayout gbLayout;
 private GridBagConstraints gbConstraints;

 public GridBagDemo()
 {
    super( "GridBagLayout" );

    container = getContentPane();
    gbLayout = new GridBagLayout();
    container.setLayout( gbLayout );

    // instantiate gridbag constraints
    gbConstraints = new GridBagConstraints();

    JTextArea ta = new JTextArea( "TextArea1", 5, 10 );
    JTextArea tx = new JTextArea( "TextArea2", 2, 2 );
    String names[] = { "Iron", "Steel", "Brass" };
    JComboBox cb = new JComboBox( names );
    JTextField tf = new JTextField( "TextField" );
    JButton b1 = new JButton( "Button 1" );
    JButton b2 = new JButton( "Button 2" );
    JButton b3 = new JButton( "Button 3" );
```

Create a **GridBagLayout** and set the layout manager.

Create a **GridBagConstraints** object, used to set the constraints.

Outline

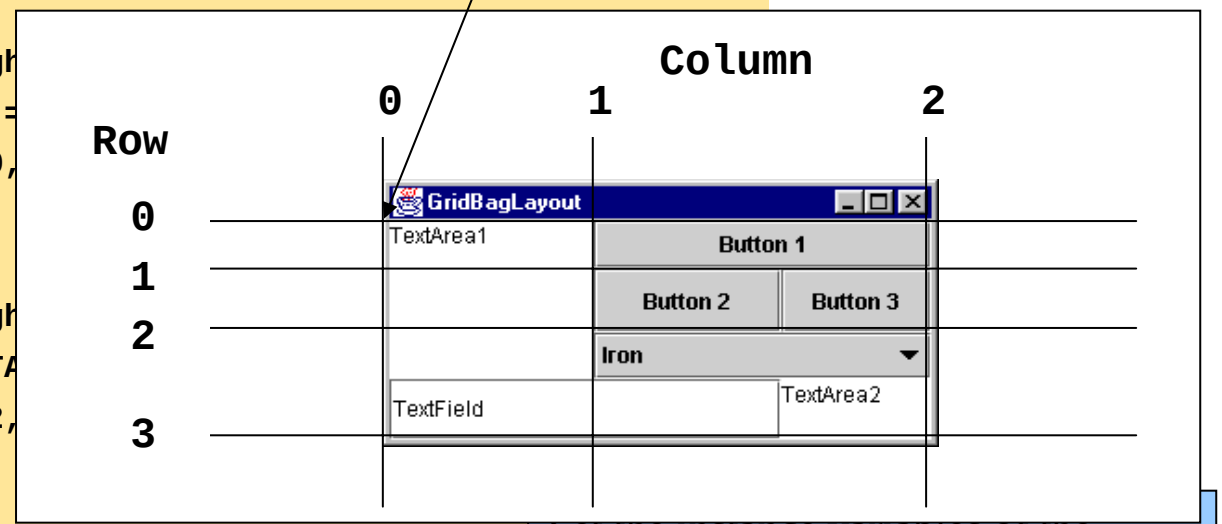
1. GridBagLayout

1.1 GridBagConstraints

1.2 GUI components

```
// text area
// weightx and weighty are both 0: the d
// anchor for all components is CENTER:
gbConstraints.fill = GridBagConstraints.
addComponent( ta, 0, 0, 1, 3 );

// button b1
// weightx and weigh
gbConstraints.fill =
addComponent( b1, 0,

// combo box
// weightx and weigh
// fill is HORIZONTA
addComponent( cb, 2,

// button b2
gbConstraints.weightx = 1000;  // can grow wid
gbConstraints.weighty = 1;     // can grow taller
gbConstraints.fill = GridBagConstraints.BOTH;
addComponent( b2, 1, 1, 1, 1 );
```

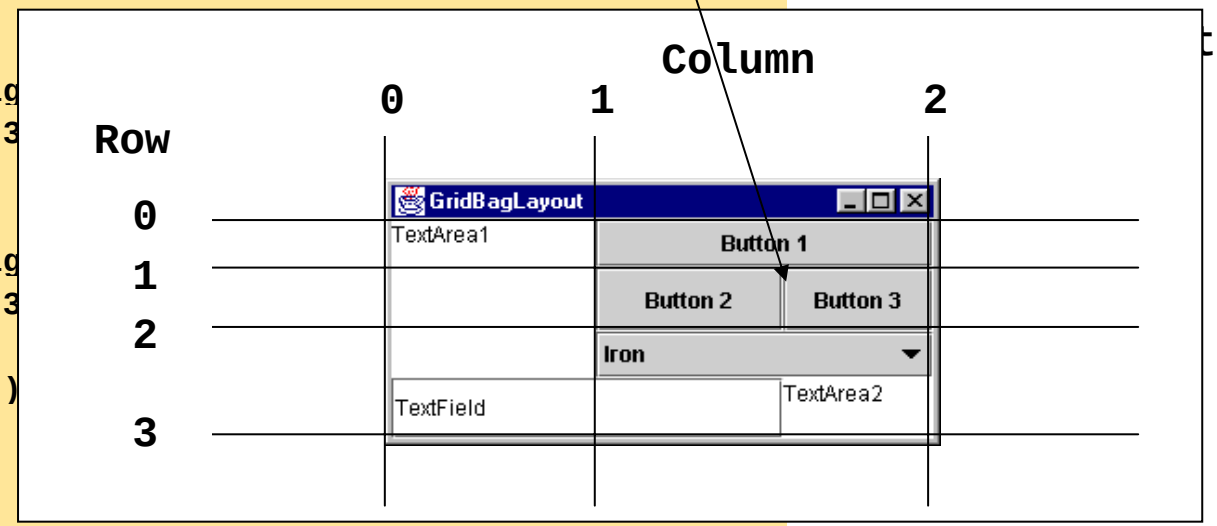
Use method **addComponent** to set instance variables of the **GridBagConstaints** object. The text area is at row 0, column 0, occupies 1 column and 3 rows.



Set the instance variables of the **GridBagConstraints** object.

```
   // button b3
   // fill is BOTH
   gbConstraints.weightx = 0;
   gbConstraints.weighty = 0;
   addComponent( b3, 1, 2, 1, 1 );


   // textfield
   // weightx and weig
   addComponent( tf, 3

   // textarea
   // weightx and weig
   addComponent( tx, 3

   setSize( 300, 150 )
   show();
}

// addComponent is programmer defined
private void addComponent( Component c,
   int row, int column, int width, int height )
{
   // set gridx and gridy
   gbConstraints.gridx = column;
   gbConstraints.gridy = row;

   // set gridwidth and gridheight
   gbConstraints.gridwidth = width;
   gbConstraints.gridheight = height;
```

Method **addComponent**. Button 3 is at row 1, column 2 (Button 2 is at column 1), occupies 1 column and 1 row.

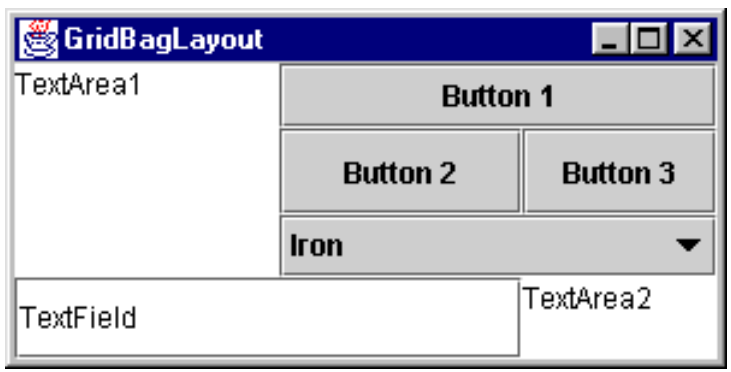


Set instance variables to parameters.

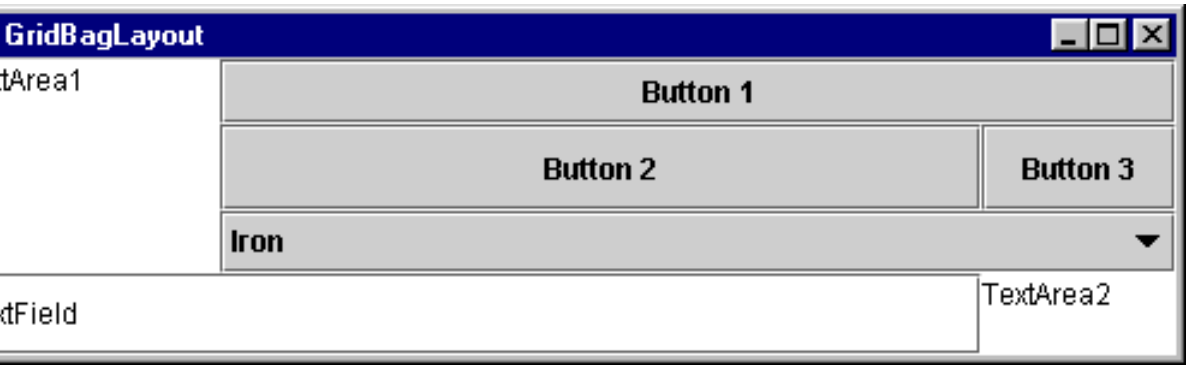
```
   // set constraints
   gbLayout.setConstraints( c, gbConstraints );
   container.add( c );      // add component
}

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

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

Call **`setConstraints`** and add component to content pane.

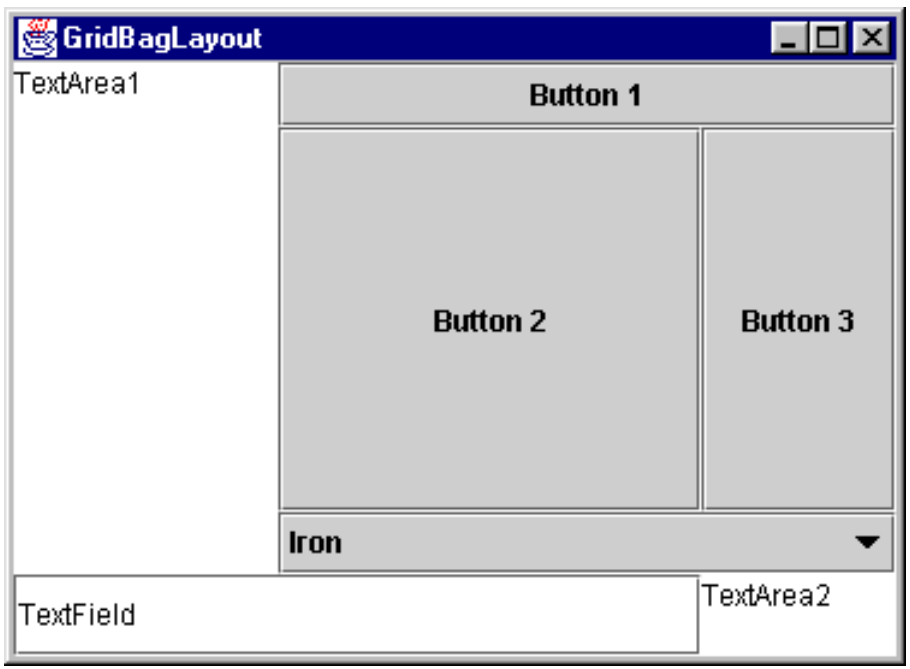
4. `main`



Outline

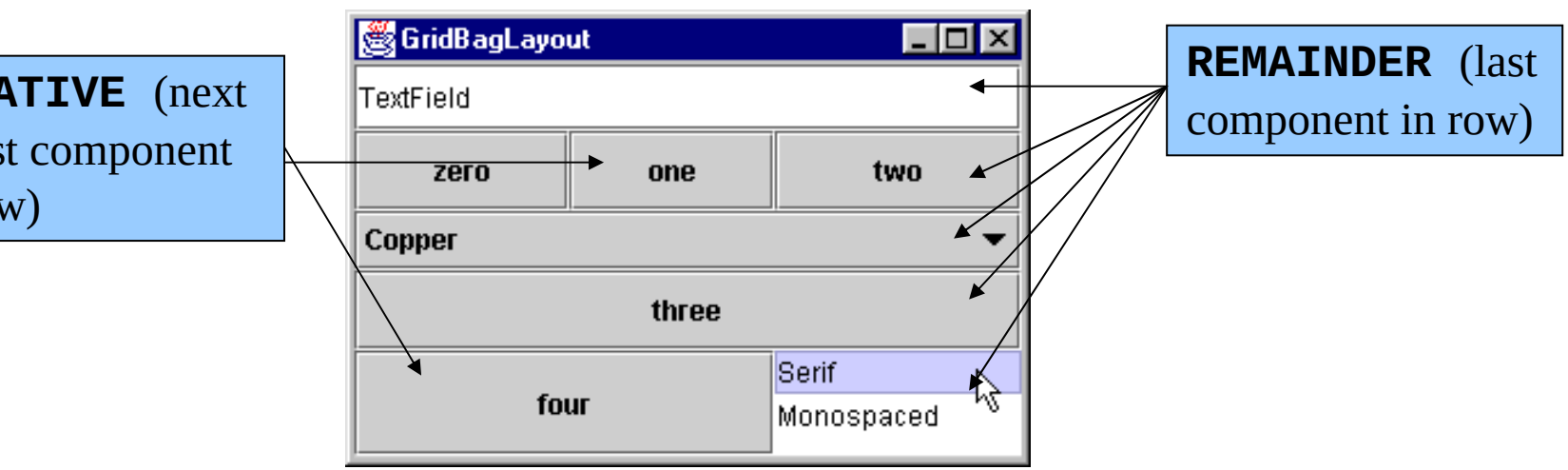


Program Output



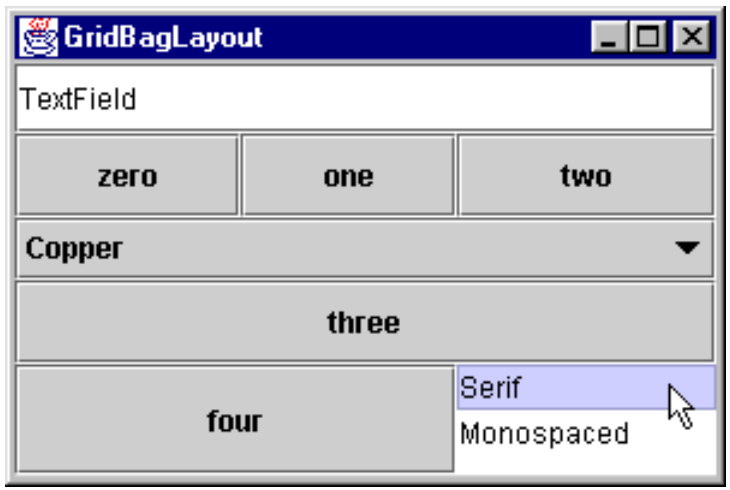
3.16 GridBagConstraints Constants RELATIVE and REMAINDER

- Variation of **GridBagLayout**
  - Does not use **gridx** or **gridy**
  - Instead, set **gridwidth** and **gridheight** to **GridBagConstraints** constants
    - **REMAINDER**- component last in row
    - **RELATIVE** - next-to-last component placed to right of previous



3.16 GridBagConstraints Constants RELATIVE and REMAINDER

- Example
  - Create GUI shown



  - If a component is not the last or next-to-last (i.e., button 0)
    - Set **`gridwidth`** to **`1`** so it occupies a column

```
Fig. 3.18: GridBagDemo2.java
 Demonstrating GridBagLayout constants.
port javax.swing.*;
port java.awt.*;
port java.awt.event.*;

blic class GridBagDemo2 extends JFrame {
 private GridBagLayout gbLayout;
 private GridBagConstraints gbConstraints;
 private Container container;

 public GridBagDemo2()
 {
    super( "GridBagLayout" );

    container = getContentPane();
    gbLayout = new GridBagLayout();
    container.setLayout( gbLayout );

    // instantiate gridbag constraints
    gbConstraints = new GridBagConstraints();

    // create GUI components
    String metals[] = { "Copper", "Aluminum", "Silver" };
    JComboBox comboBox = new JComboBox( metals );

    JTextField textField = new JTextField( "TextField" );

    String fonts[] = { "Serif", "Monospaced" };
    JList list = new JList( fonts );
```

Outline

1. GridBagLayout

1.1

1.2 Components

As before, create an object in order to set the instance variables.

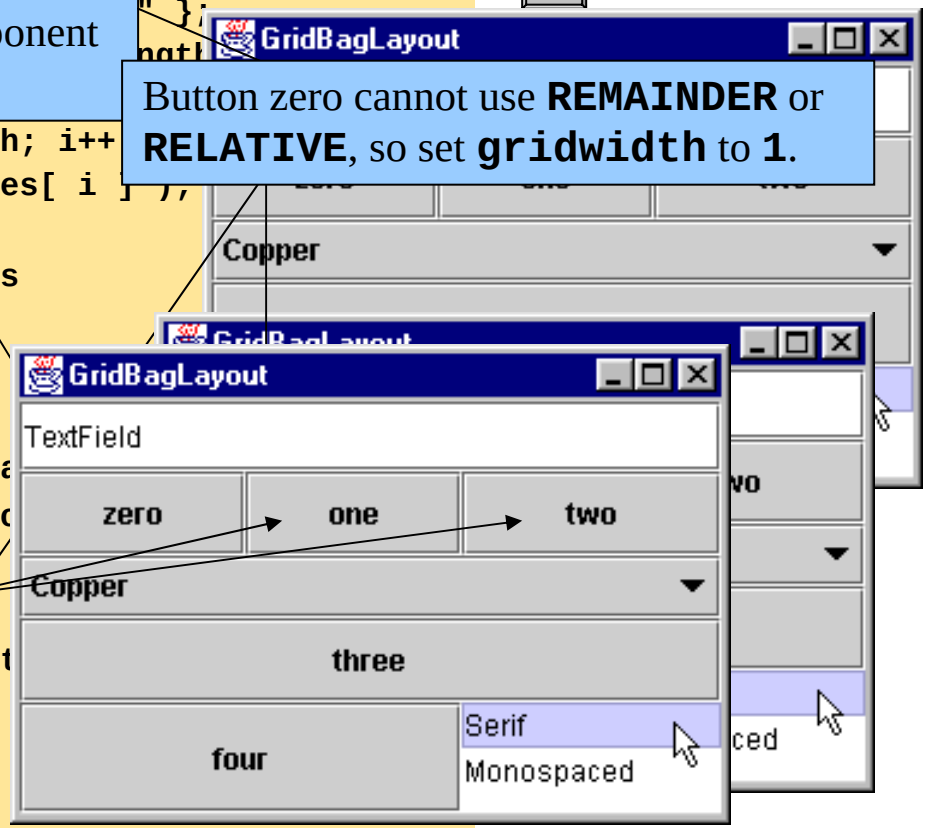
Outline

```
String names
   { "zero",                                    " };
JButton but                                     ngth
for ( int i = 0; i < buttons.length; i++
   buttons[ i ] = new JButton( names[ i ] );

// define GUI component constraints
// textField
gbConstraints.weightx = 1;
gbConstraints.weighty = 1;
gbConstraints.fill = GridBagConstra
gbConstraints.gridwidth = GridBagCo
a
                                          eight
gbConstraints.gridwidth = 1;
addComponent( buttons[ 0 ] );

// buttons[1] -- weightx and weighty are 1: fill is BOTH
gbConstraints.gridwidth = GridBagConstraints.RELATIVE;
addComponent( buttons[ 1 ] );

// buttons[2] -- weightx and weighty are 1: fill is BOTH
gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
addComponent( buttons[ 2 ] );
```

Set **gridwidth** to **REMAINDER** (last component in row).

Button zero cannot use **REMAINDER** or **RELATIVE**, so set **gridwidth** to **1**.

The other components in the row can use **RELATIVE** and **REMAINDER**.



```
   // comboBox -- weightx is 1: fill is BOTH
   gbConstraints.weighty = 0;
   gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
   addComponent( comboBox );

   // buttons[3] -- weightx is 1: fill is BOTH
   gbConstraints.weighty = 1;
   gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
   addComponent( buttons[ 3 ] );

   // buttons[4] -- weightx and weighty are 1: fill is BOTH
   gbConstraints.gridwidth = GridBagConstraints.RELATIVE;
   addComponent( buttons[ 4 ] );

   // list -- weightx and weighty are 1: fill is BOTH
   gbConstraints.gridwidth = GridBagConstraints.REMAINDER;
   addComponent( list );

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

// addComponent is programmer-defined
private void addComponent( Component c )
{
   gbLayout.setConstraints( c, gbConstraints );
   container.add( c );      // add component
}
```

Outline

3. Define addComponent

3.1 setConstraints

```
public static void main( String args[] )
{
   GridBagDemo2 app = new GridBagDemo2();

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

Outline

4. `main`

Program Output