UCLA PIC 20A
Java Programming

●**Instructor**: **Ivo Dinov,**

Asst. Prof. In Statistics, Neurology and Program in Computing

●**Teaching Assistant:** Yon Seo Kim, **PIC**

University of California, Los Angeles, Summer 2002

http://www.stat.ucla.edu/~dinov/

Chapter 10 – GUIs

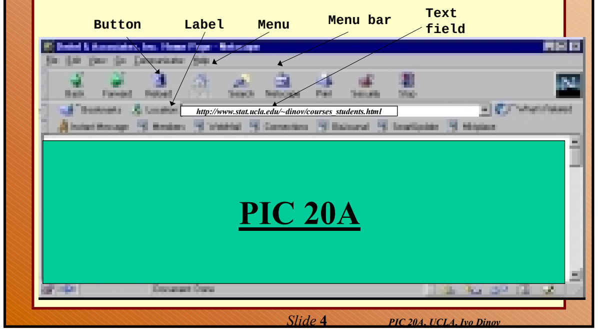
- JLabel
- Event Handling Model
- JTextField and JPasswordField
- How Event Handling Works
- JButton
- JCheckBox and JRadioButton
- JComboBox
- JList
- Multiple-Selection Lists
- Mouse Event Handling
- Adapter Classes
- Keyboard Event Handling
- Layout Managers
  1. FlowLayout
  2. BorderLayout
  3. GridLayout

Introduction

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

Introduction

- Example GUI: Netscape Communicator

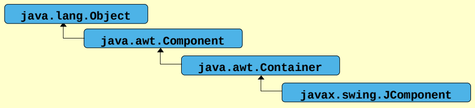


Swing Overview

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

Swing Overview

- Swing component inheritance hierarchy



`java.lang.Object`

`java.awt.Component`

`java.awt.Container`

`javax.swing.JComponent`

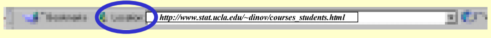
- ❑**`Component`** defines methods that can be used in its subclasses (for example, **`paint`** and **`repaint`**)
- ❑**`Container`** - collection of related components
  - ☐ When using **`JFrames`**, attach components to the content pane (a **`Container`**)
  - ☐ Method **`add`**
- ❑**`JComponent`**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

Swing Overview

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

JLabel

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



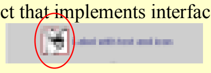
- Methods

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

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

JLabel

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



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

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

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

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

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

JLabel

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

```
// Fig. 12.4: LabelTest.java
// Demonstrating the JLabel class.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class LabelTest extends JFrame {
   private JLabel label1, label2, label

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

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

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

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

      // JLabel constructor no arguments
```

- 1. import
- 1.1 Class Labeltest (extends JFrame)
- 1.2 Declarations
- 1.3 getContentPane
- 2. Initialize JLabels
- 2.1 setToolTipText

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

Initialize text in **JLabel** constructor.

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

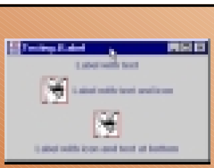
Create a new **ImageIcon** (assumed to be in same directory as program). More Chapter 16.

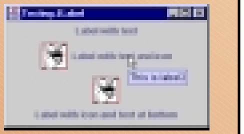
Set **ImageIcon** and alignment of text in **JLabel** constructor.

```
      label3 = new JLabel();
      label3.setText( "Label with icon and text at bottom" );
      label3.setIcon( bug );
      label3.setHorizontalTextPosition(
         SwingConstants.CENTER );
      label3.setVerticalTextPosition(
         SwingConstants.BOTTOM );
      label3.setToolTipText( "This is label3" );
      c.add( label3 );

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

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

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

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

- 2.2 setHorizontalText Position
- 2.3 setVerticalText Position
- 2.3 setToolTipText
- 3. main





- Program Output

Event Handling Model

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

Event Handling Model

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

`JTextField` and `JPasswordField`

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

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

`JTextField` and `JPasswordField`

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

`JTextField` and `JPasswordField`

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

```
// Fig. 12.7: TextFieldTest.java
// Demonstrating the JTextField class.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

   public TextFieldTest()
   {
      super( "Testing JTextField and JPasswordField" );

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

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

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

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

- ● 1. `import`
- ● 1.1 Declarations
- ● 1.2 Constructor
- ● 1.3 GUI components
- ● 2. Initialize text fields
- ● 2.1 `setEditable`

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

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

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

      TextFieldHandler handler = new TextFieldH
      text1.addActionListener( handler );
      text2.addActionListener( handler );
      text3.addActionListener( handler );
      password.addActionListener( handler );

      setSize( 325, 100 );
      show();
   }

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

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

   // inner class for event handling
```

`JPasswordField` initialized with text, which appears as asterisks.

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

- ● 2.2 `JPasswordField`
- ● 2.3 Event handler
- ● 3. `main`

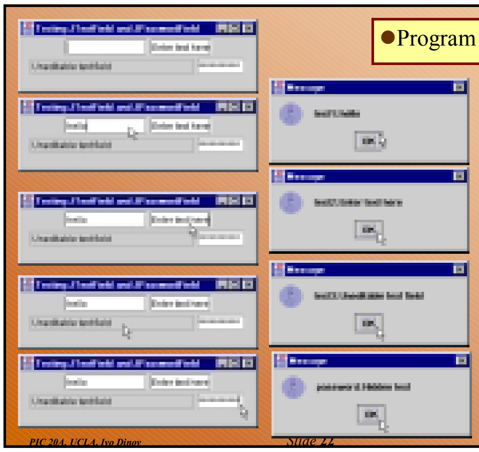
```
   private class TextFieldHandler implements ActionListener {
      public void actionPerformed( ActionEvent e )
      {
         String s = "";

         if ( e.getSource() == text1 )
            s = "text1: " + e.getActionCommand();
         else if ( e.getSource() == text2 )
            s = "text2: " + e.getActionCommand();
         else if ( e.getSource() == text3 )
            s = "text3: " + e.getActionCommand();
         else if ( e.getSource() == password ) {
            JPasswordField pwd =
               (JPasswordField) e.getSource();
            s = "password: " +
                new String( pwd.getPassw
         }

         JOptionPane.showMessageDialog( null, s );
      }
   }
}
```

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

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

- ● 4. Inner class `TextFieldHandler` (event handler)
- ● 4.1 `getSource`
- ● 4.2 `getActionCommand`
- ● 4.3 Downcast reference

- ● Program Output



text1: hello

text2: Enter text here

text3: Uneditable text field

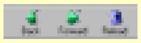
password: Hidden text

How Event Handling Works

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

JButton

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



JButton

- Methods of class **JButton**
  - Constructors

    JButton myButton = new JButton("Label");

    JButton myButton = new JButton("Label", myIcon);
  - **setRolloverIcon(myIcon)**
    - ❑Sets image to display when mouse over button
- Class **ActionEvent**
  - **getActionCommand**
    - ❑Returns label of button that generated event

```
// ButtonTest.java
// Creating JButtons.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonTest extends JFrame
   private JButton plainButton, fancyB

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

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

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

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

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

- 1. `import`
- 1.1 Declarations
- 2. Initialize buttons and `Icons`
- 2.1 `setRolloverIcon`
- 2.2 Register event handler

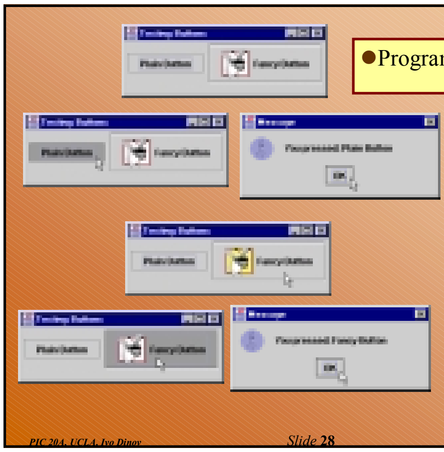
Create **JButtons**. Initialize **fancyButton** with an **ImageIcon**.

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

```
      plainButton.addActionListener( handler );

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

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

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

   // inner class for button event handling
   private class ButtonHandler implements ActionListener {
      public void actionPerformed( ActionEvent e )
      {
         JOptionPane.showMessageDialog( null,
            "You pressed: " + e.getActionCommand() );
      }
   }
}
```

- 3. `main`
- 4. Inner class event handler

getActionCommand returns label of button that generated event.



You pressed: Plain Button

You pressed: Fancy Button

- Program Output

JCheckBox and JRadioButton

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

  - Text appears to right of checkbox
  - Constructor

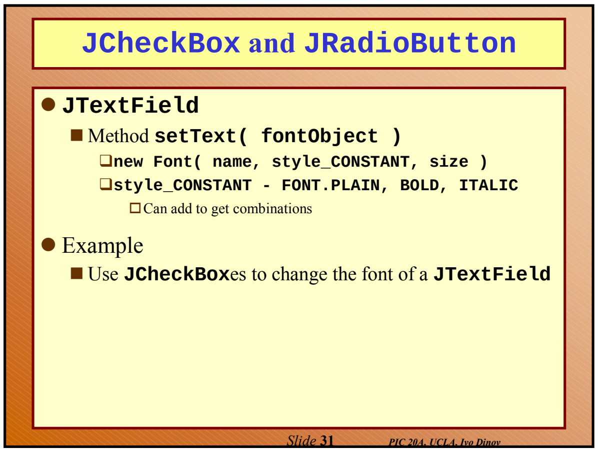
    JCheckBox myBox = new JCheckBox("Title");

JCheckBox and JRadioButton

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

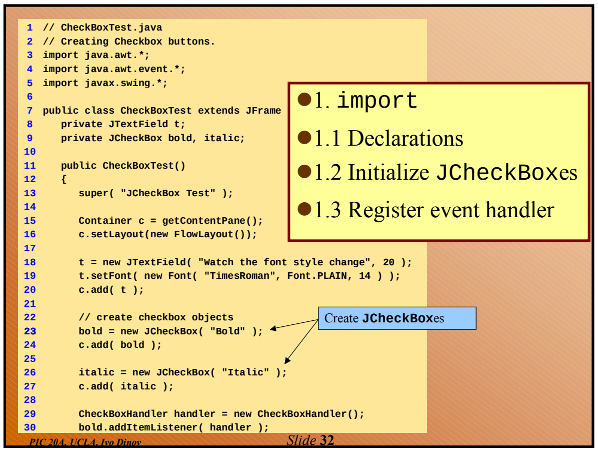
```
   private class CheckBoxHandler implements ItemListener {
      public void itemStateChanged( ItemEvent e )
      {
```

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



JCheckBox and JRadioButton

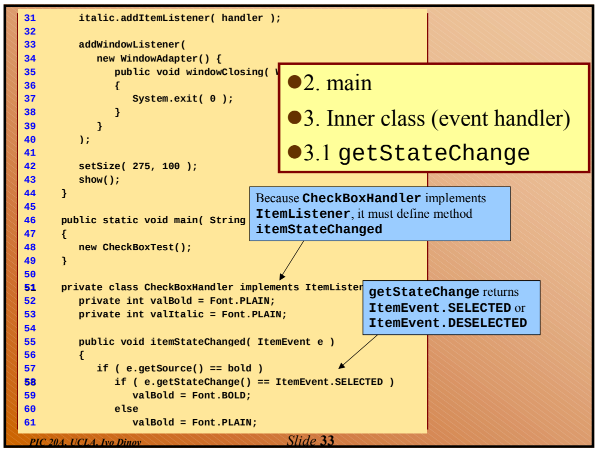
- **JTextField**
  - Method **setText(fontObject)**
    - ❑**new Font(name, style_CONSTANT, size)**
    - ❑**style_CONSTANT - FONT.PLAIN, BOLD, ITALIC**
      - □Can add to get combinations
- Example
  - Use **JCheckBox**es to change the font of a **JTextField**



```
// CheckBoxTest.java
// Creating Checkbox buttons.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

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

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

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

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

      italic = new JCheckBox( "Italic" );
      c.add( italic );

      CheckBoxHandler handler = new CheckBoxHandler();
      bold.addItemListener( handler );
```

- 1. **import**
- 1.1 Declarations
- 1.2 Initialize **JCheckBox**es
- 1.3 Register event handler

Create **JCheckBox**es

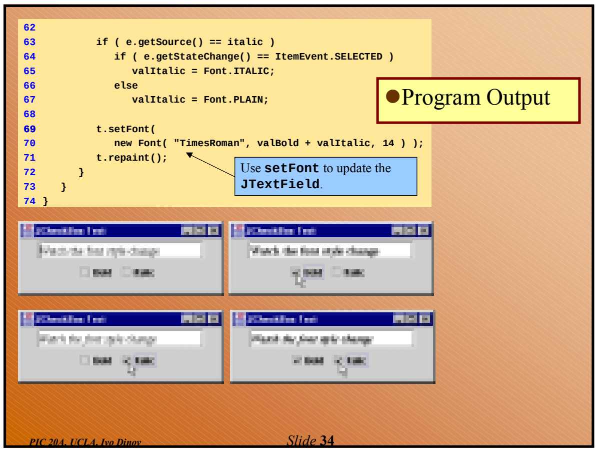


```
      italic.addItemListener( handler );

      addWindowListener(
         new WindowAdapter() {
            public void windowClosing( W
            {
               System.exit( 0 );
            }
         }
      );

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

   public static void main( String
   {
      new CheckBoxTest();
   }

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

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

- 2. main
- 3. Inner class (event handler)
- 3.1 **getStateChange**

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

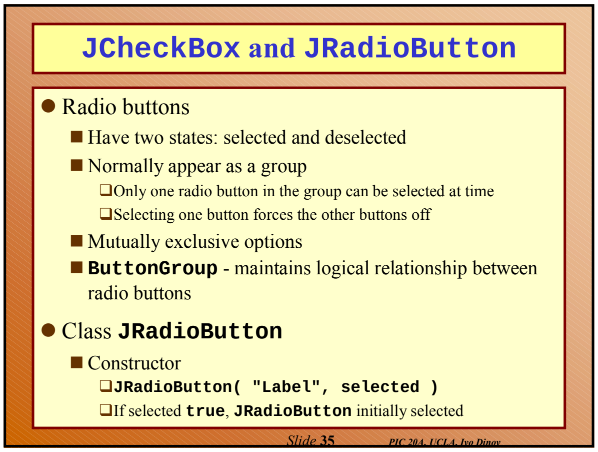
getStateChange returns **ItemEvent.SELECTED** or **ItemEvent.DESELECTED**



```

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

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

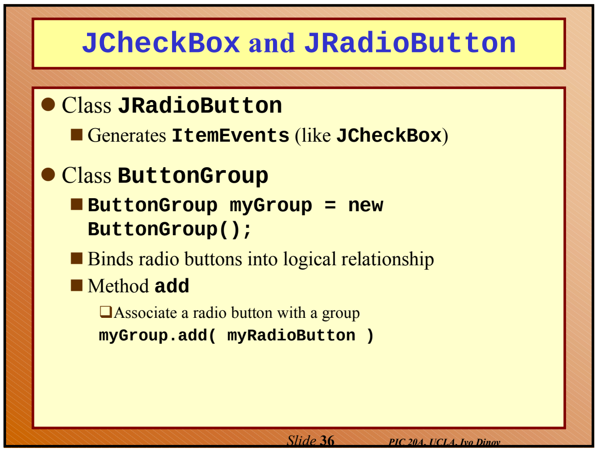
- Program Output

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



JCheckBox and JRadioButton

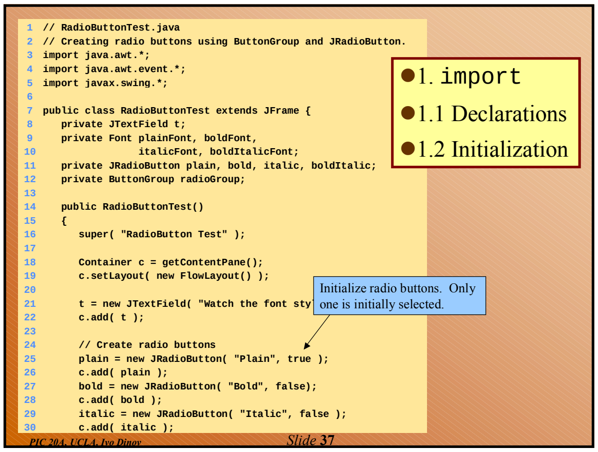
- Radio buttons
  - Have two states: selected and deselected
  - Normally appear as a group
    - ❑Only one radio button in the group can be selected at time
    - ❑Selecting one button forces the other buttons off
  - Mutually exclusive options
  - **ButtonGroup** - maintains logical relationship between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



JCheckBox and JRadioButton

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

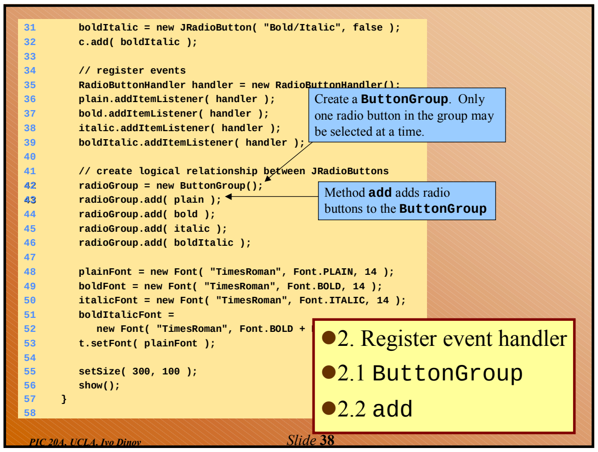
    myGroup.add(myRadioButton)



```
// RadioButtonTest.java
// Creating radio buttons using ButtonGroup and JRadioButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

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

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

      t = new JTextField( "Watch the font sty
      c.add( t );

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

- 1. `import`
- 1.1 Declarations
- 1.2 Initialization

Initialize radio buttons. Only one is initially selected.

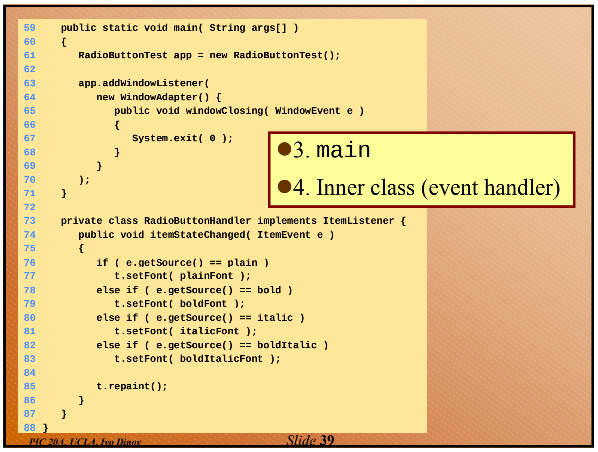


```
      boldItalic = new JRadioButton( "Bold/Italic", false );
      c.add( boldItalic );

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

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

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

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

```

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

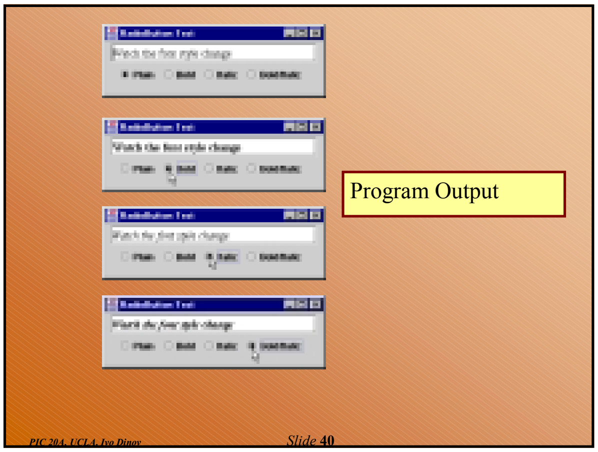
Method **`add`** adds radio buttons to the **`ButtonGroup`**

- 2. Register event handler
- 2.1 `ButtonGroup`
- 2.2 `add`

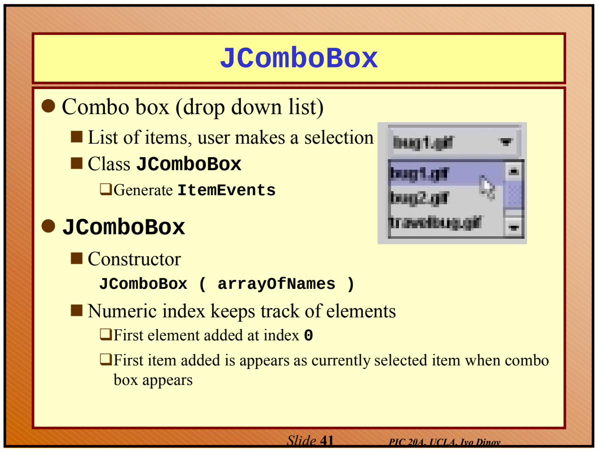


```
   public static void main( String args[] )
   {
      RadioButtonTest app = new RadioButtonTest();

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

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

         t.repaint();
      }
   }
}
```

- 3. `main`
- 4. Inner class (event handler)

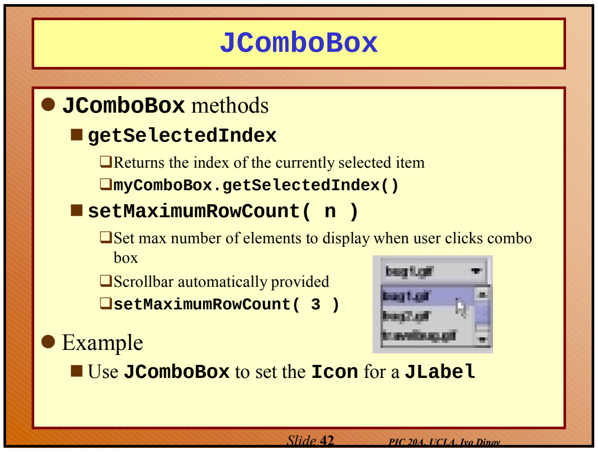


Program Output


bug1.gif
bug1.gif
bug2.gif
travelbug.gif

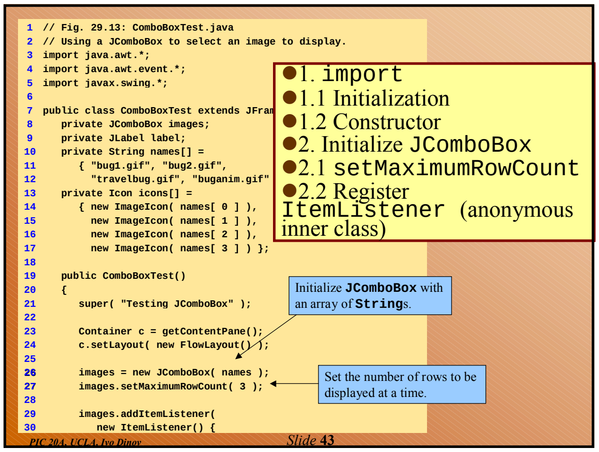
JComboBox

- Combo box (drop down list)
  - List of items, user makes a selection
  - Class **`JComboBox`**
    - Generate **`ItemEvents`**
- **`JComboBox`**
  - Constructor
    `JComboBox ( arrayOfNames )`
  - Numeric index keeps track of elements
    - First element added at index **`0`**
    - First item added is appears as currently selected item when combo box appears


bug1.gif
bug1.gif
bug2.gif
travelbug.gif

JComboBox

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



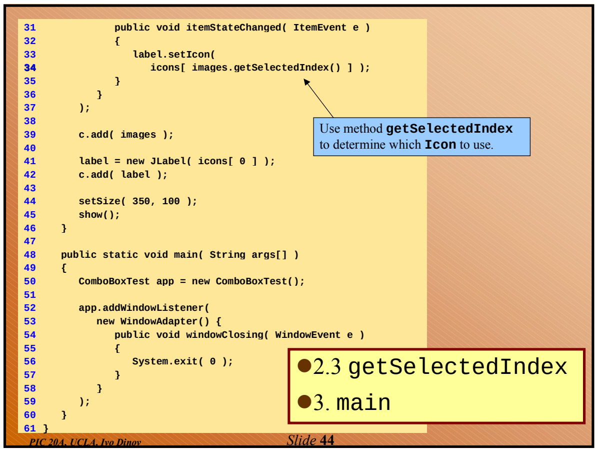
```java
// Fig. 29.13: ComboBoxTest.java
// Using a JComboBox to select an image to display.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ComboBoxTest extends JFram
   private JComboBox images;
   private JLabel label;
   private String names[] =
      { "bug1.gif", "bug2.gif",
        "travelbug.gif", "buganim.gif"
   private Icon icons[] =
      { new ImageIcon( names[ 0 ] ),
        new ImageIcon( names[ 1 ] ),
        new ImageIcon( names[ 2 ] ),
        new ImageIcon( names[ 3 ] ) };

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

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

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

      images.addItemListener(
         new ItemListener() {
```

- 1. `import`
- 1.1 Initialization
- 1.2 Constructor
- 2. Initialize `JComboBox`
- 2.1 `setMaximumRowCount`
- 2.2 Register `ItemListener` (anonymous inner class)

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

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

PIC 20A, UCLA, Ivo Dinov *Slide* **43**

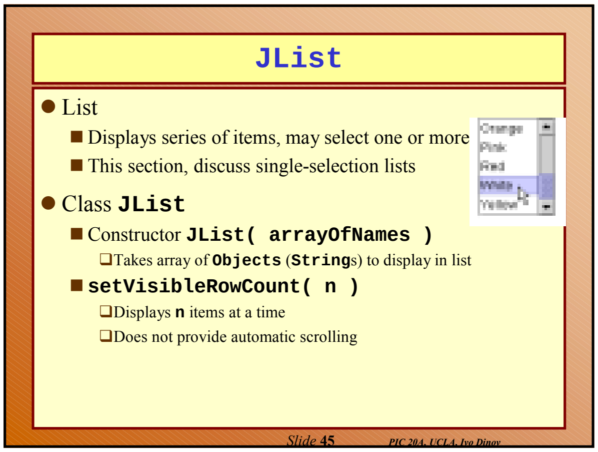


```java
            public void itemStateChanged( ItemEvent e )
            {
               label.setIcon(
                  icons[ images.getSelectedIndex() ] );
            }
         }
      );

      c.add( images );

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

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

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

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

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

- 2.3 `getSelectedIndex`
- 3. `main`

PIC 20A, UCLA, Ivo Dinov *Slide* **44**

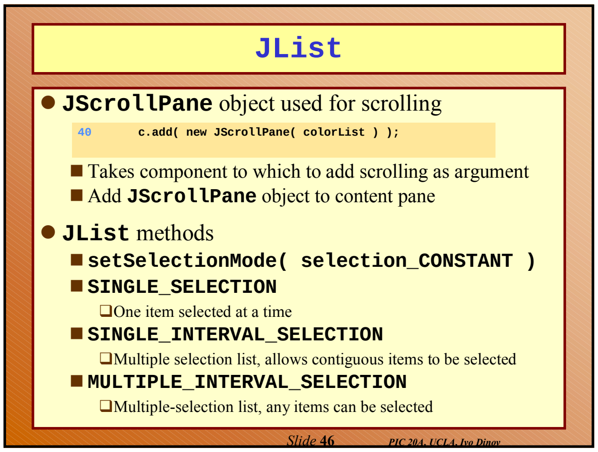


JList

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

Orange
Pink
Red
White
Yellow

Slide **45** *PIC 20A, UCLA, Ivo Dinov*



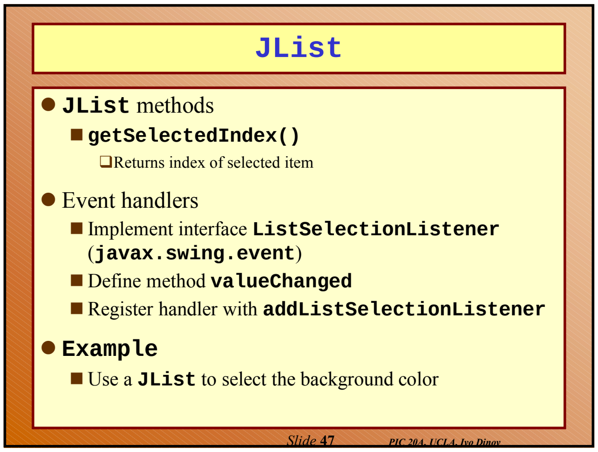
JList

- **`JScrollPane`** object used for scrolling

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

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

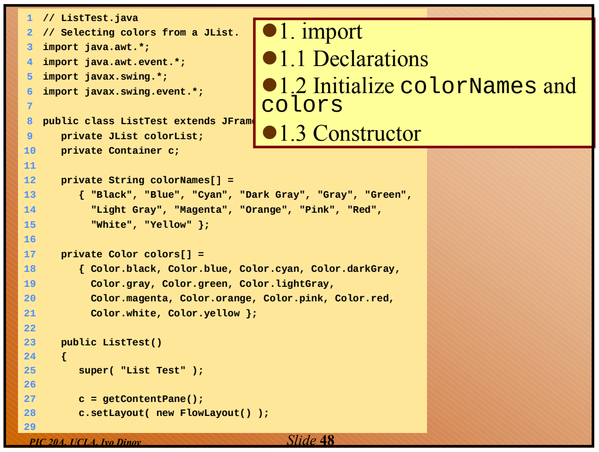
Slide **46** *PIC 20A, UCLA, Ivo Dinov*



JList

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

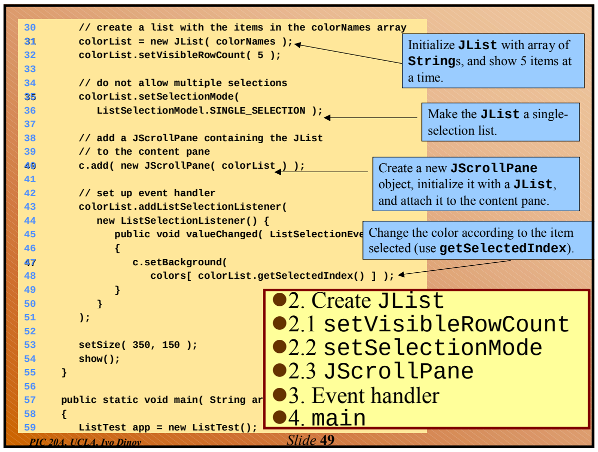
Slide **47** *PIC 20A, UCLA, Ivo Dinov*



```java
// ListTest.java
// Selecting colors from a JList.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class ListTest extends JFrame
   private JList colorList;
   private Container c;

   private String colorNames[] =
      { "Black", "Blue", "Cyan", "Dark Gray", "Gray", "Green",
        "Light Gray", "Magenta", "Orange", "Pink", "Red",
        "White", "Yellow" };

   private Color colors[] =
      { Color.black, Color.blue, Color.cyan, Color.darkGray,
        Color.gray, Color.green, Color.lightGray,
        Color.magenta, Color.orange, Color.pink, Color.red,
        Color.white, Color.yellow };

   public ListTest()
   {
      super( "List Test" );

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

```

- 1. import
- 1.1 Declarations
- 1.2 Initialize `colorNames` and `colors`
- 1.3 Constructor

PIC 20A, UCLA, Ivo Dinov *Slide* **48**



```
      // create a list with the items in the colorNames array
      colorList = new JList( colorNames );
      colorList.setVisibleRowCount( 5 );

      // do not allow multiple selections
      colorList.setSelectionMode(
         ListSelectionModel.SINGLE_SELECTION );

      // add a JScrollPane containing the JList
      // to the content pane
      c.add( new JScrollPane( colorList ) );

      // set up event handler
      colorList.addListSelectionListener(
         new ListSelectionListener() {
            public void valueChanged( ListSelectionEve
            {
               c.setBackground(
                  colors[ colorList.getSelectedIndex() ] );
            }
         }
      );

      setSize( 350, 150 );
      show();
   }

   public static void main( String ar
   {
      ListTest app = new ListTest();
```

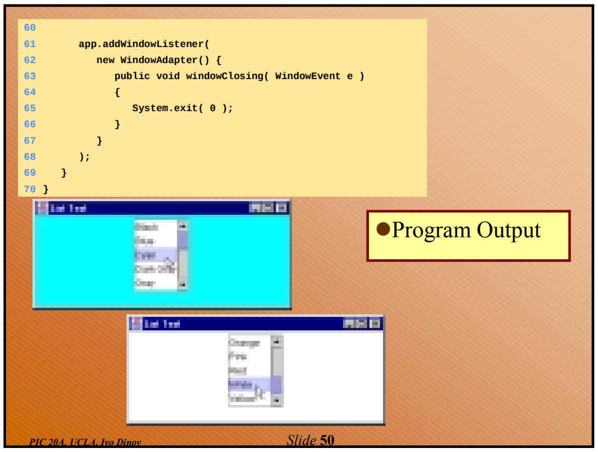
Initialize **`JList`** with array of **`String`**s, and show 5 items at a time.

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

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

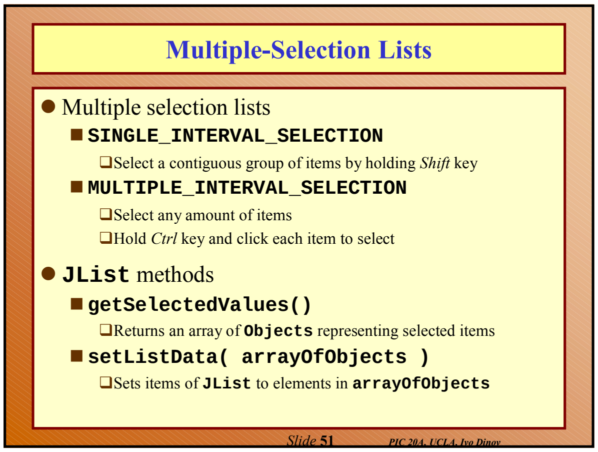
Change the color according to the item selected (use **`getSelectedIndex`**).

- 2. Create `JList`
- 2.1 `setVisibleRowCount`
- 2.2 `setSelectionMode`
- 2.3 `JScrollPane`
- 3. Event handler
- 4. `main`



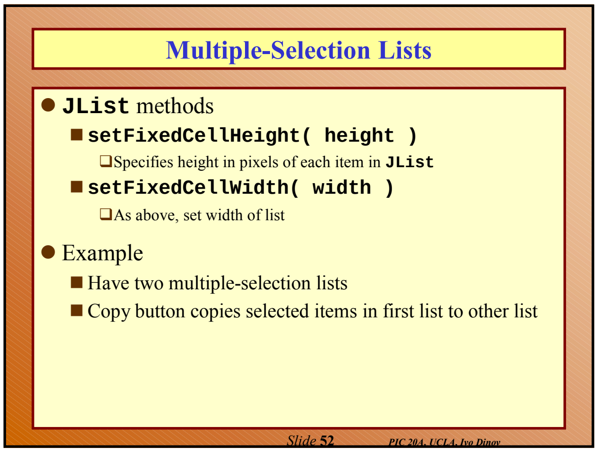
```

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

- Program Output



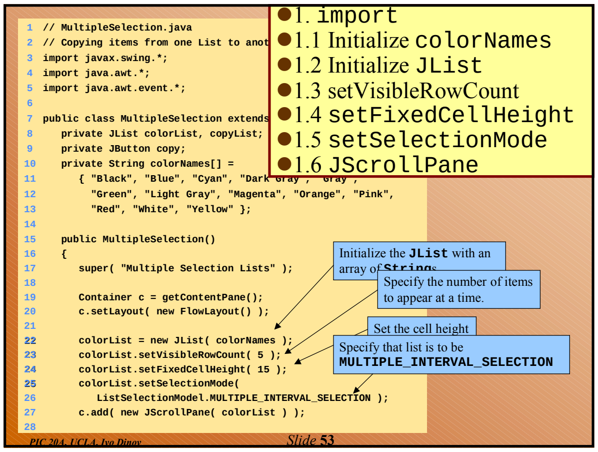
Multiple-Selection Lists

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



Multiple-Selection Lists

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



```
// MultipleSelection.java
// Copying items from one List to anot
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MultipleSelection extends
   private JList colorList, copyList;
   private JButton copy;
   private String colorNames[] =
      { "Black", "Blue", "Cyan", "Dark Gray", "Gray",
        "Green", "Light Gray", "Magenta", "Orange", "Pink",
        "Red", "White", "Yellow" };

   public MultipleSelection()
   {
      super( "Multiple Selection Lists" );

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

      colorList = new JList( colorNames );
      colorList.setVisibleRowCount( 5 );
      colorList.setFixedCellHeight( 15 );
      colorList.setSelectionMode(
         ListSelectionModel.MULTIPLE_INTERVAL_SELECTION );
      c.add( new JScrollPane( colorList ) );

```

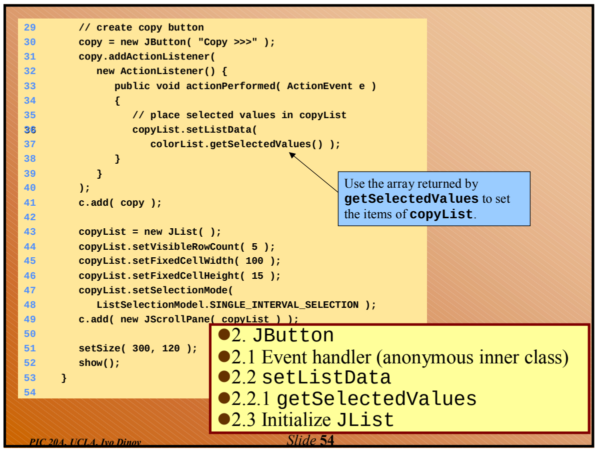
- 1. `import`
- 1.1 Initialize `colorNames`
- 1.2 Initialize `JList`
- 1.3 setVisibleRowCount
- 1.4 `setFixedCellHeight`
- 1.5 `setSelectionMode`
- 1.6 `JScrollPane`

Initialize the **`JList`** with an array of **`Strings`**

Specify the number of items to appear at a time.

Set the cell height

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

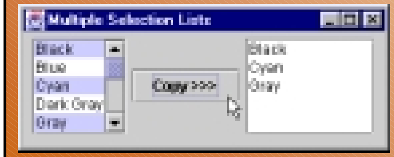


```
      // create copy button
      copy = new JButton( "Copy >>>" );
      copy.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               // place selected values in copyList
               copyList.setListData(
                  colorList.getSelectedValues() );
            }
         }
      );
      c.add( copy );

      copyList = new JList( );
      copyList.setVisibleRowCount( 5 );
      copyList.setFixedCellWidth( 100 );
      copyList.setFixedCellHeight( 15 );
      copyList.setSelectionMode(
         ListSelectionModel.SINGLE_INTERVAL_SELECTION );
      c.add( new JScrollPane( copyList ) );

      setSize( 300, 120 );
      show();
   }

```

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

- 2. `JButton`
- 2.1 Event handler (anonymous inner class)
- 2.2 `setListData`
- 2.2.1 `getSelectedValues`
- 2.3 Initialize `JList`

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

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



- 3. `main`
- Program Output

Mouse Event Handling

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

Mouse Event Handling

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

Mouse Event Handling

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

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

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

```
// MouseTracker.java
// Demonstrating mouse events.

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

public class MouseTracker extends JFrame
             implements MouseListener, MouseMotionListener {
   private JLabel statusBar;

   public MouseTracker()
   {
      super( "Demonstrating Mouse Events" );

      statusBar = new JLabel();
      getContentPane().add( statusBar, BorderLayout.SOUTH );

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

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

   // MouseListener event handle
   public void mouseClicked( Mou
   {
```

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

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

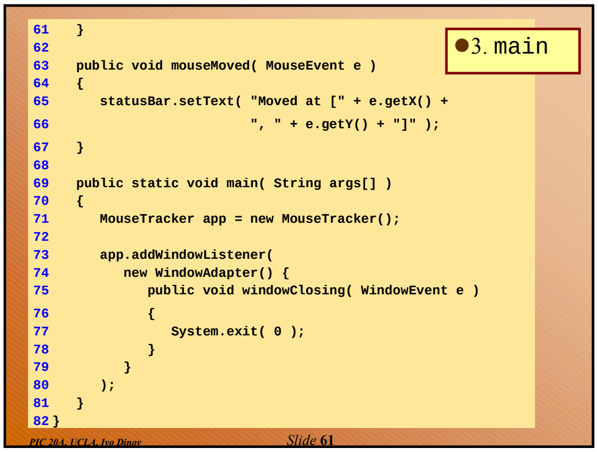
Application is its own event handler

- 1. `import`
- 1. Class MouseTracker (`implements MouseListener`, `MouseMotion Listener`)
- 1.2 Register event handlers (`this`)
- 2. Define event handler methods

```
      statusBar.setText( "Clicked at [" + e.getX() +
                         ", " + e.getY() + "]" );
   }

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

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

   public void mouseEntered( MouseEvent e )
   {
      statusBar.setText( "Mouse in window" );
   }

   public void mouseExited( MouseEvent e )
   {
      statusBar.setText( "Mouse outside window" );
   }

   // MouseMotionListener event handlers
   public void mouseDragged( MouseEvent e )
   {
      statusBar.setText( "Dragged at [" + e.getX() +
                         ", " + e.getY() + "]" );
```

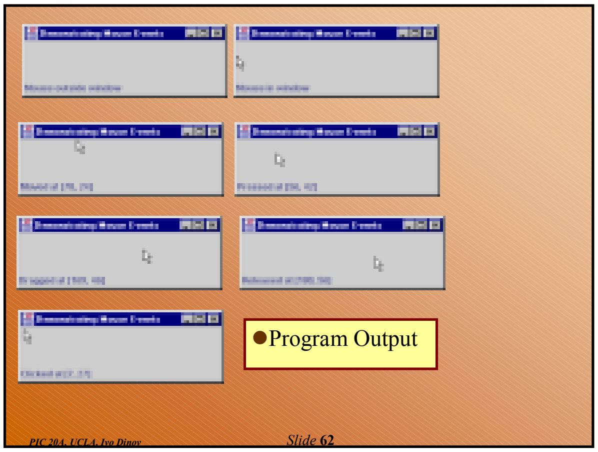
`getX` and **`getY`** return the coordinates of where the mouse event occurred.

- 2.1 `getX` and `getY`



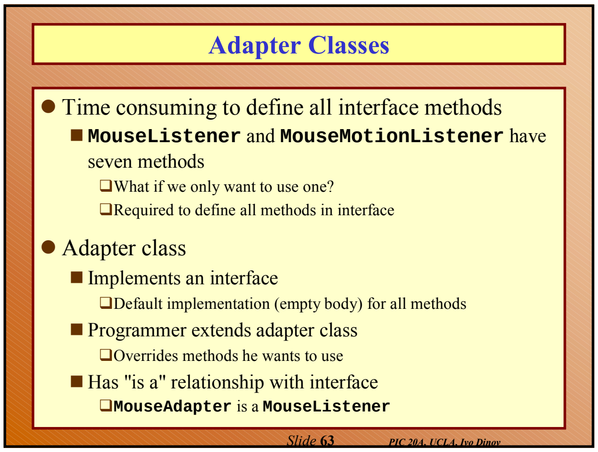
```
   }

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

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

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

● 3. `main`



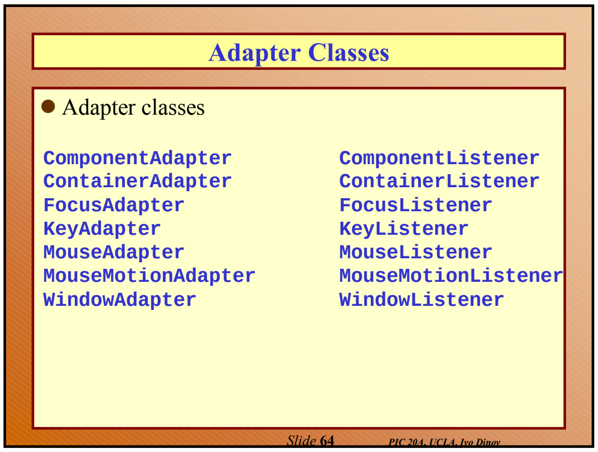
```
Mouse outside window
Mouse in window
Moved at [78, 79]
Pressed at [56, 42]
Dragged at [105, 46]
Released at [190, 50]
Clicked at [7, 17]
```

● Program Output



Adapter Classes

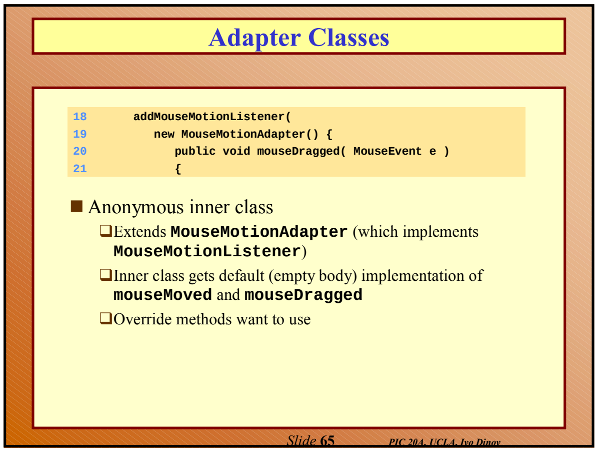
- Time consuming to define all interface methods
  - **`MouseListener`** and **`MouseMotionListener`** have seven methods
    - ❑What if we only want to use one?
    - ❑Required to define all methods in interface
- Adapter class
  - Implements an interface
    - ❑Default implementation (empty body) for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



Adapter Classes

- Adapter classes

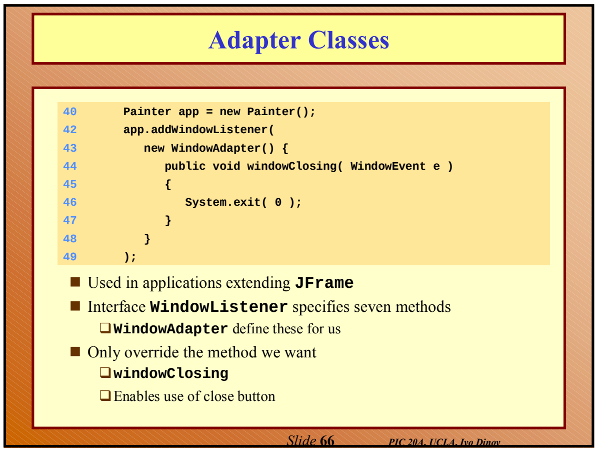
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`



Adapter Classes

```
      addMouseMotionListener(
         new MouseMotionAdapter() {
            public void mouseDragged( MouseEvent e )
            {
```

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



Adapter Classes

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

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

Adapter Classes

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

```java
// Painter.java
// Using class MouseMotionAdapter
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Painter ext
   private int xValue =

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

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

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

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

- 1. `import`
- 1.1 `addMouseMotion Listener`

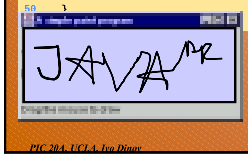
1.2. `MouseMotionAdapter`

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

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

```java

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

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

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

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



- 2. `paint`
- 3. `main`
- 3.1 `addWindowListener`
- 3.2 `WindowAdapter`

Adapter Classes

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

Adapter Classes

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

```java
// MouseDetails.java
// Demonstrating mouse clicks and
// distinguishing between mouse buttons.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

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

   public MouseDetails()
   {
      super( "Mouse clicks and buttons" );

      addMouseListener( new MouseClickHandler() );

      setSize( 350, 150 );
      show();
   }

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

```

- 1. `import`
- 1.1 Constructor
- 1.2 Register event handler
- 2. `paint`

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

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

   // inner class to handle mouse events
   private class MouseClickHandler extends MouseAdapter {
      public void mouseClicked( MouseEvent e )
      {
         xPos = e.getX();
         yPos = e.getY();

         String s =
            "Clicked " + e.getClickCount() + " time(s)";

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

```

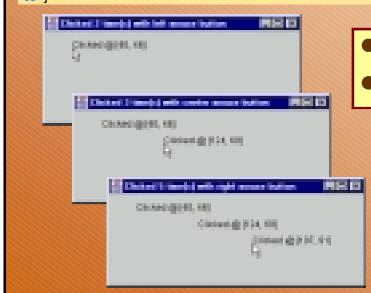
Use a named inner class as the event handler. Can still inherit from MouseAdapter (**extends MouseAdapter**).

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

- 3. `main`
- 4. Inner class (event handler)
- 4.1 `getClickCount`
- 4.2 `isMetaDown`
- 4.3 `isAltDown`

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

Set the title bar.



Clicked 1 time(s) with left mouse button

Clicked 1 time(s) with center mouse button

Clicked 1 time(s) with right mouse button

- 4.4 `setTitle`
- Program Output

Keyboard Event Handling

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

Keyboard Event Handling

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

Keyboard Event Handling

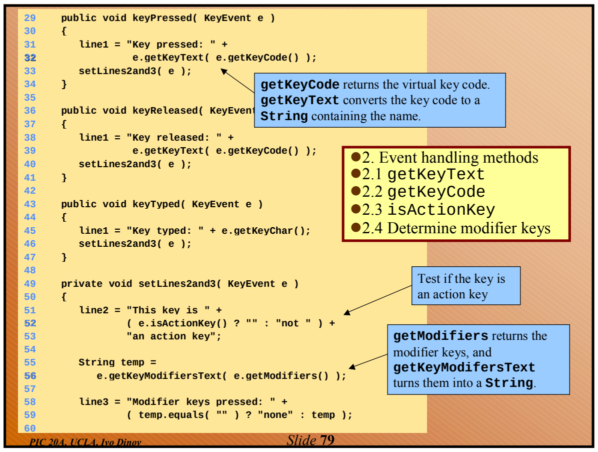
- **KeyEvent** methods
  - **getModifiers** (from class **InputEvent**)
    - Returns which modifiers were pressed
  - **getKeyModifierText (e.getModifiers)**
    - Returns string containing names of modifier keys
- Upcoming example
  - Create a **JTextArea**
  - Modify text depending on what keys are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

```
// KeyDemo.java
// Demonstrating keystroke events.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

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

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

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

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

      getContentPane().add( textArea );

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

```

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

Register the event handler.

- 1. import
- 1.1 Class `KeyDemo` (implements `KeyListener`)
- 1.2 `addKeyListener`



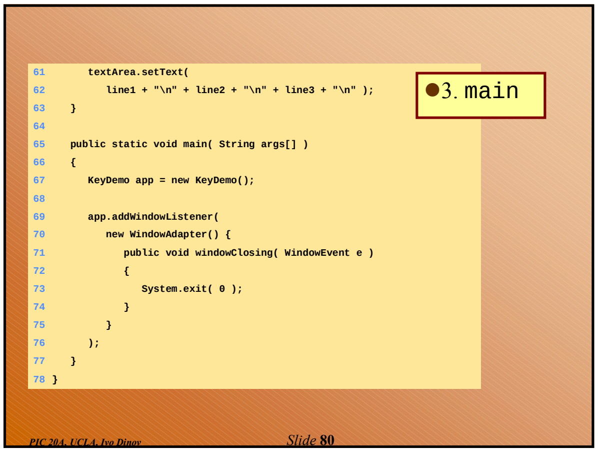
```
public void keyPressed( KeyEvent e )
{
   line1 = "Key pressed: " +
            e.getKeyText( e.getKeyCode() );
   setLines2and3( e );
}

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

public void keyTyped( KeyEvent e )
{
   line1 = "Key typed: " + e.getKeyChar();
   setLines2and3( e );
}

private void setLines2and3( KeyEvent e )
{
   line2 = "This key is " +
           ( e.isActionKey() ? "" : "not " ) +
           "an action key";

   String temp =
      e.getKeyModifiersText( e.getModifiers() );

   line3 = "Modifier keys pressed: " +
           ( temp.equals( "" ) ? "none" : temp );

```

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

- 2. Event handling methods
- 2.1 `getKeyText`
- 2.2 `getKeyCode`
- 2.3 `isActionKey`
- 2.4 Determine modifier keys

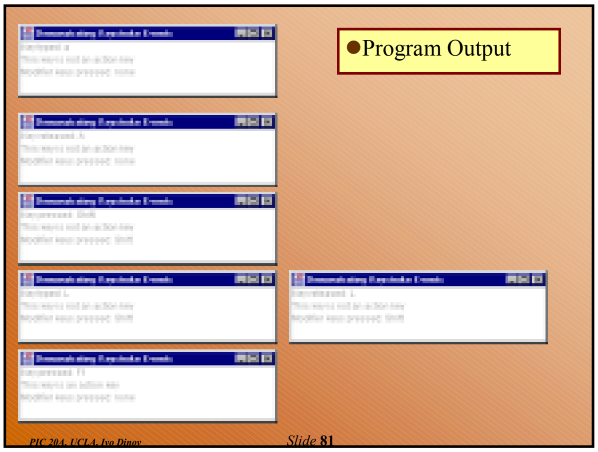
Test if the key is an action key

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



```
      textArea.setText(
         line1 + "\n" + line2 + "\n" + line3 + "\n" );
   }

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

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

- 3. `main`



- Program Output

```
Key typed: a
This key is not an action key
Modifier keys pressed: none
```

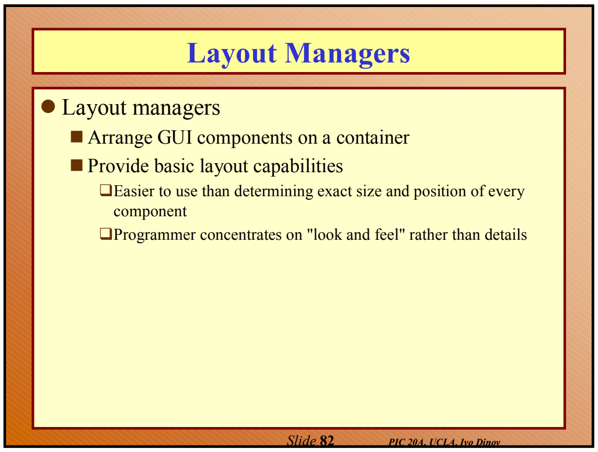
```
Key released: A
This key is not an action key
Modifier keys pressed: none
```

```
Key pressed: Shift
This key is not an action key
Modifier keys pressed: Shift
```

```
Key typed: L
This key is not an action key
Modifier keys pressed: Shift
```

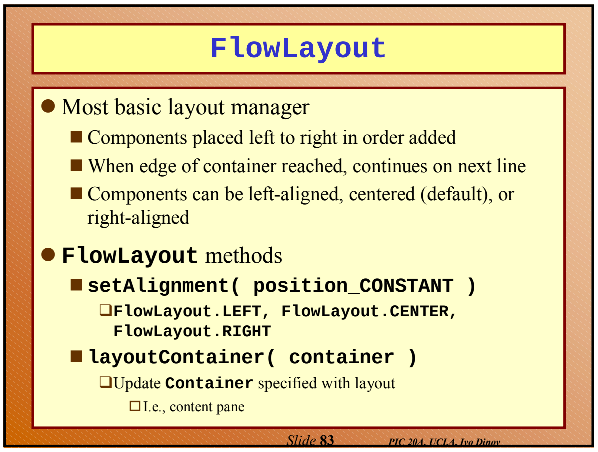
```
Key released: L
This key is not an action key
Modifier keys pressed: Shift
```

```
Key pressed: F1
This key is an action key
Modifier keys pressed: none
```



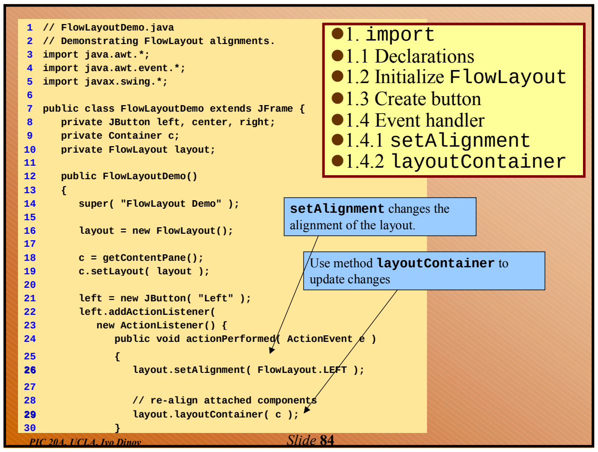
Layout Managers

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



`FlowLayout`

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

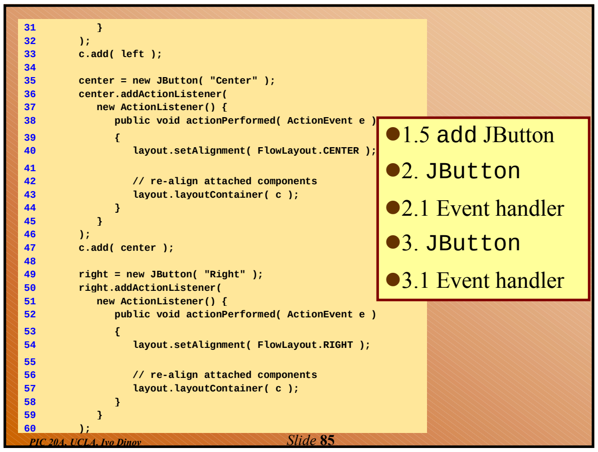


```
// FlowLayoutDemo.java
// Demonstrating FlowLayout alignments.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

   public FlowLayoutDemo()
   {
      super( "FlowLayout Demo" );

      layout = new FlowLayout();

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

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

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

- 1. `import`
- 1.1 Declarations
- 1.2 Initialize `FlowLayout`
- 1.3 Create button
- 1.4 Event handler
- 1.4.1 `setAlignment`
- 1.4.2 `layoutContainer`

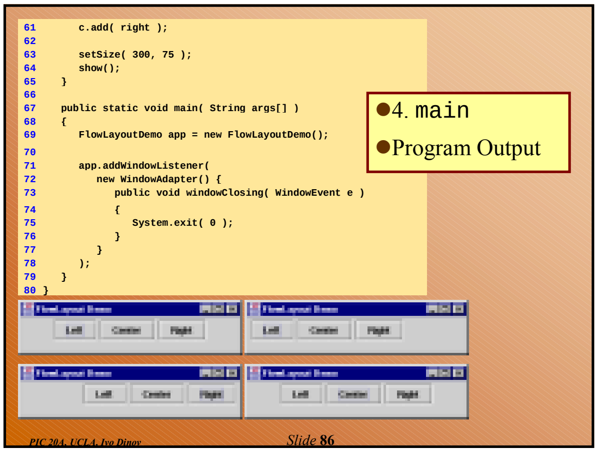
setAlignment changes the alignment of the layout.

Use method **layoutContainer** to update changes



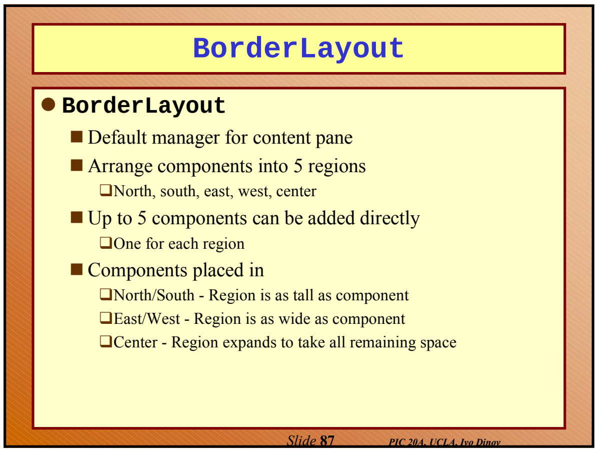
```
         }
      );
      c.add( left );

      center = new JButton( "Center" );
      center.addActionListener(
         new ActionListener() {
            public void actionPerformed( ActionEvent e )
            {
               layout.setAlignment( FlowLayout.CENTER );

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

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

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

- 1.5 add JButton
- 2. `JButton`
- 2.1 Event handler
- 3. `JButton`
- 3.1 Event handler



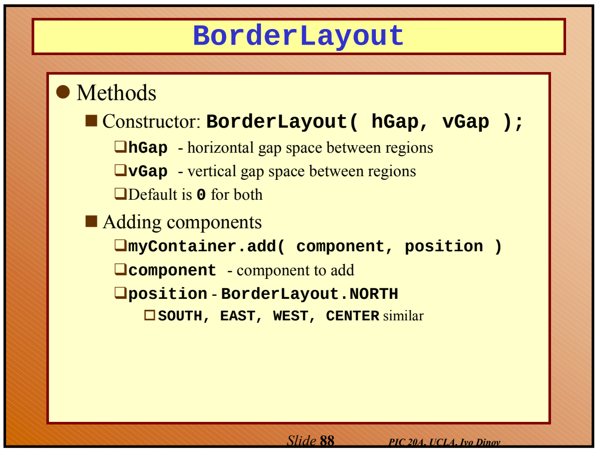
```
      c.add( right );

      setSize( 300, 75 );
      show();
   }

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

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

- 4. `main`
- Program Output



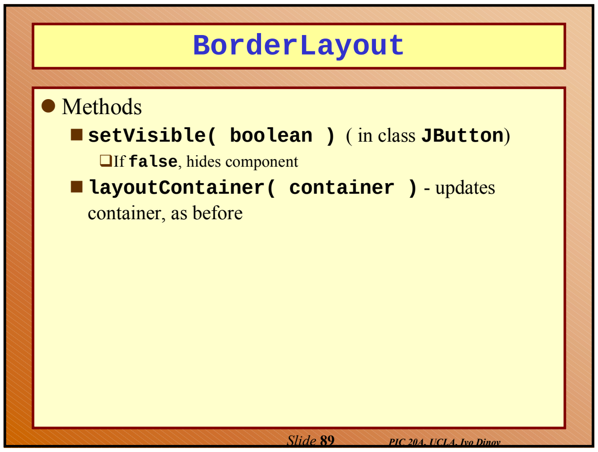
BorderLayout

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



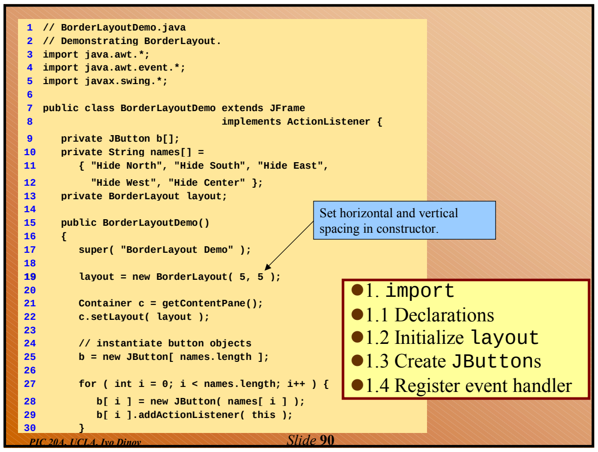
BorderLayout

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



BorderLayout

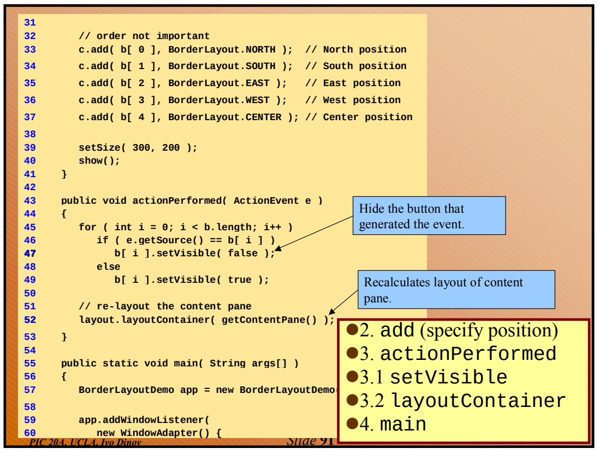
- Methods
  - **`setVisible( boolean )`** (in class **`JButton`**)
    - If **`false`**, hides component
  - **`layoutContainer( container )`** - updates container, as before



```
// BorderLayoutDemo.java
// Demonstrating BorderLayout.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class BorderLayoutDemo extends JFrame
                              implements ActionListener {
   private JButton b[];
   private String names[] =
      { "Hide North", "Hide South", "Hide East",
        "Hide West", "Hide Center" };
   private BorderLayout layout;

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

      layout = new BorderLayout( 5, 5 );

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

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

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

Set horizontal and vertical spacing in constructor.

- 1. `import`
- 1.1 Declarations
- 1.2 Initialize `layout`
- 1.3 Create `JButtons`
- 1.4 Register event handler

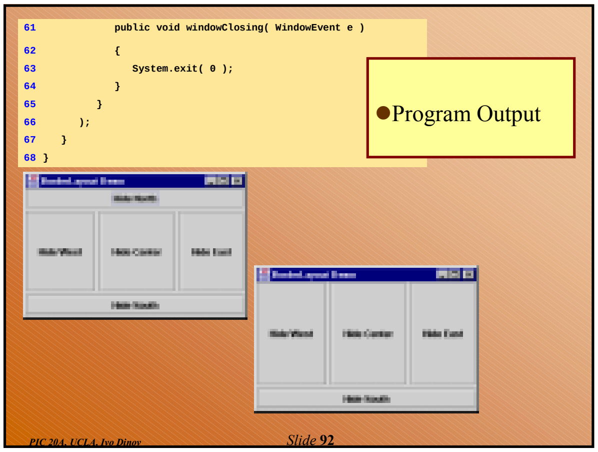


```
   // order not important
   c.add( b[ 0 ], BorderLayout.NORTH );  // North position
   c.add( b[ 1 ], BorderLayout.SOUTH );  // South position
   c.add( b[ 2 ], BorderLayout.EAST );   // East position
   c.add( b[ 3 ], BorderLayout.WEST );   // West position
   c.add( b[ 4 ], BorderLayout.CENTER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

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

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

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

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

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

Hide the button that generated the event.

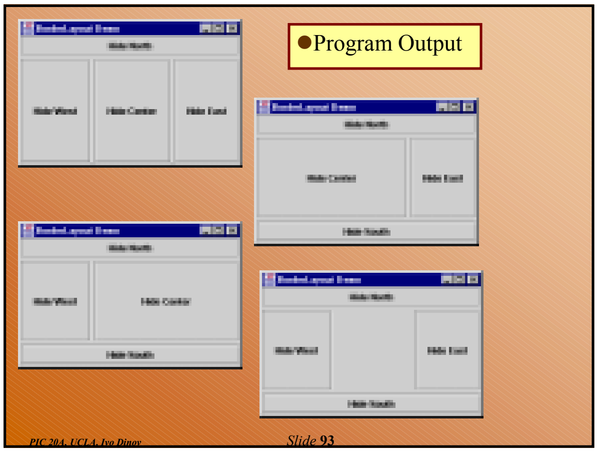
Recalculates layout of content pane.

- 2. `add` (specify position)
- 3. `actionPerformed`
- 3.1 `setVisible`
- 3.2 `layoutContainer`
- 4. `main`

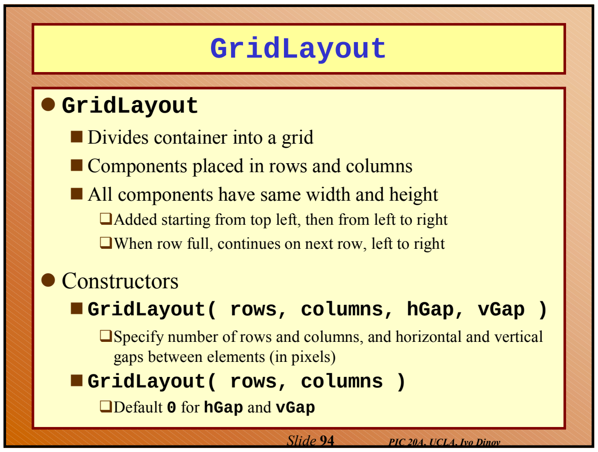


```
         public void windowClosing( WindowEvent e )
         {
            System.exit( 0 );
         }
      }
   );
}
}
```

- Program Output

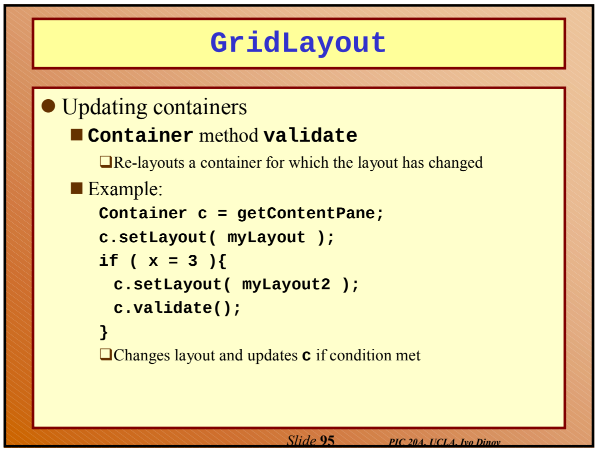


- Program Output



`GridLayout`

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

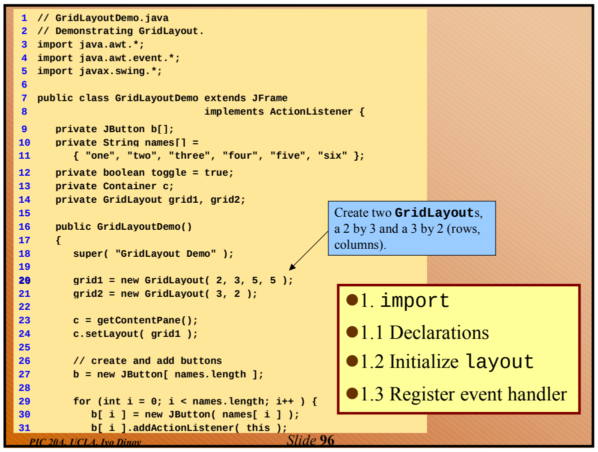


`GridLayout`

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

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

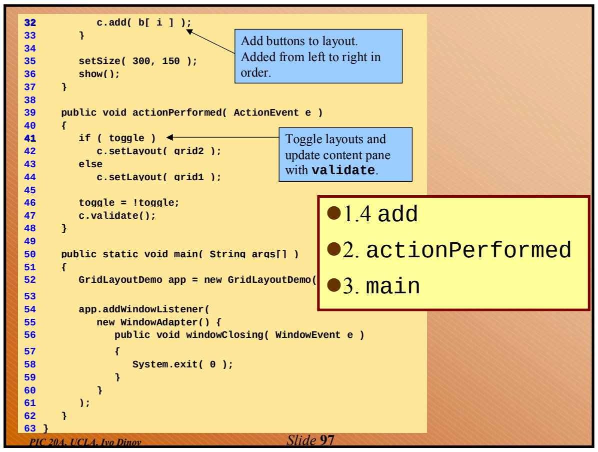
❑Changes layout and updates **`c`** if condition met



```
// GridLayoutDemo.java
// Demonstrating GridLayout.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class GridLayoutDemo extends JFrame
                              implements ActionListener {
   private JButton b[];
   private String names[] =
      { "one", "two", "three", "four", "five", "six" };
   private boolean toggle = true;
   private Container c;
   private GridLayout grid1, grid2;

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

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

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

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

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

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

- 1. `import`
- 1.1 Declarations
- 1.2 Initialize `layout`
- 1.3 Register event handler

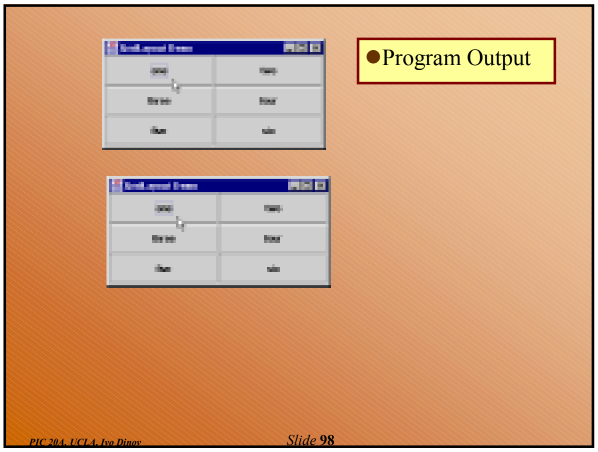


```
         c.add( b[ i ] );
      }

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

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

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

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

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

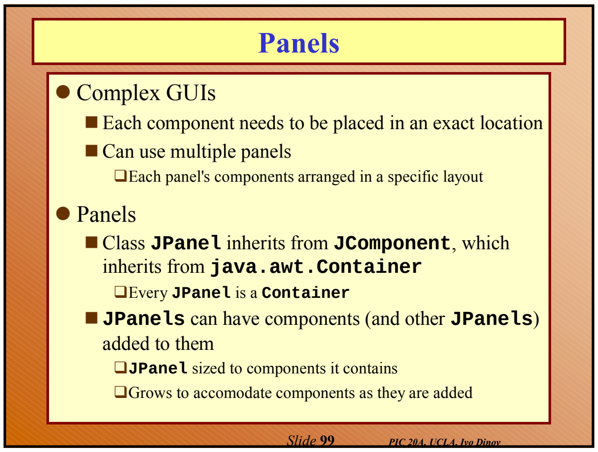
Add buttons to layout. Added from left to right in order.

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

- 1.4 `add`
- 2. `actionPerformed`
- 3. `main`

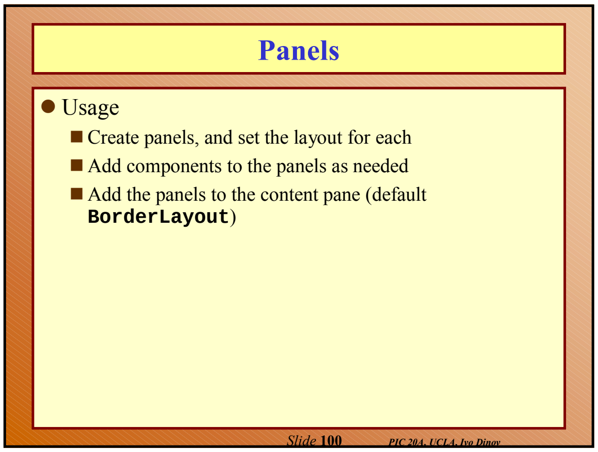


- Program Output



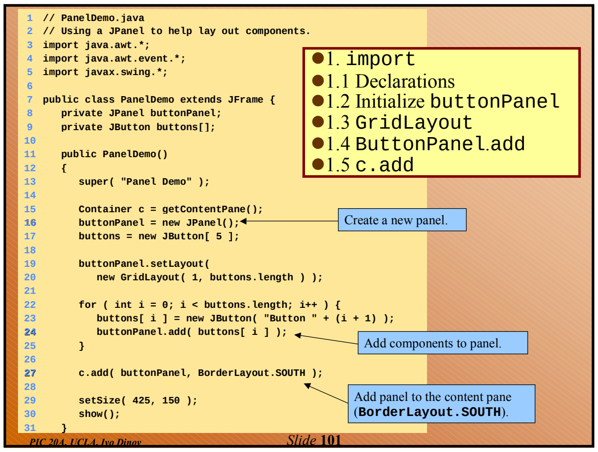
Panels

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



Panels

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



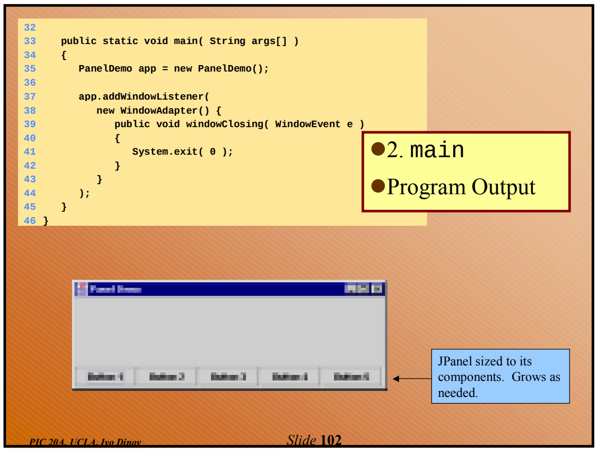
```
// PanelDemo.java
// Using a JPanel to help lay out components.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class PanelDemo extends JFrame {
   private JPanel buttonPanel;
   private JButton buttons[];

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

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

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

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

      c.add( buttonPanel, BorderLayout.SOUTH );

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

- 1. `import`
- 1.1 Declarations
- 1.2 Initialize `buttonPanel`
- 1.3 `GridLayout`
- 1.4 `ButtonPanel.add`
- 1.5 `c.add`

Create a new panel.

Add components to panel.

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



```

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

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

- 2. `main`
- Program Output

JPanel sized to its components. Grows as needed.