CMSC 202

Swing

Introduction to Swing

- Swing is a Java package used to create GUIs
- The Java *AWT (Abstract Window Toolkit)* package is the original Java package for doing *GUIs*
- A *GUI (graphical user interface)* is a windowing system that interacts with the user
- The Swing package is an improved version of the AWT
  - However, it does not completely replace the AWT
  - Some AWT classes are replaced by Swing classes, but other AWT classes are needed when using Swing
- Swing GUIs are designed using a form of object-oriented programming known as *event-driven programming*

Containers and Components

- A Swing GUI consists of two key types of items: Components and Containers although the distinction is often blurred because every Container is a Component
- Component -- a visual control (e.g. button)
- Container - holds and arranges groups of components
- All Swing GUIs have at least one “top level” container, the most common of which is the **JFrame**

A Simple Window

- A simple window can consist of an object of the **`JFrame`** class
  - A **`JFrame`** object includes a border and the usual three buttons for minimizing, changing the size of, and closing the window
  - The **`JFrame`** class is found in the **`javax.swing`** package
  - This code snippet creates a JFrame and initializes its attributes

```
JFrame = new JFrame( "This is the title");

// set the frame's size and position
frame.setSize( 300, 300);
frame.setLocation( 200, 200);

// close application when window closed
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

A Simple Window (cont'd)

Frame1.java contains the complete code that creates this window when executed.

JButton

- One of the most common components of a GUI is a button that is pushed to perform an action.
- A *button* object is created from the class **`JButton`** and can be added to a **`JFrame`**
  - The argument to the **`JButton`** constructor is the string that appears on the button when it is displayed.

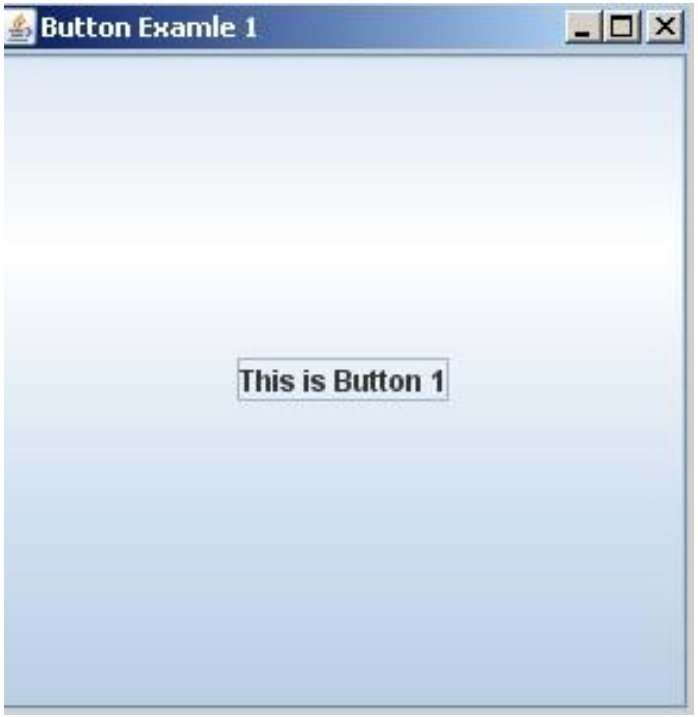
JButton Example

This code snippet creates a JButton and adds it to the frame.

```
// create and add a button to the frame
JButton button1 = new JButton("This is Button 1");
frame.add( button1 );
```

ButtonExample1.java contains the complete code that creates the window seen on the next slide.

A Frame with a Button



Note that the button fills the entire frame

Nothing Happens

- If you copy ButtonExample1.java into Eclipse and run it, you'll find that nothing happens when you push the button. What do we need to do to make "something happen" when the button is pushed?

  1. we need a method to do the "something" when the button is pushed
  2. we need to know when the button is pushed
     a. first we need to tell the button we want to be told when it's pushed
     b. the button needs a way to call our method when it's pushed

All of this is implemented using ***event-driven programming***

Events

- *Event-driven programming* is a programming style that uses a signal-and-response approach to programming
- An *event* is an object that acts as a signal from the *event source* to another object known as a *listener*
- The sending of an event is called *firing the event*
  - The object that fires the event is often a GUI component, such as a button that has been clicked

Listeners

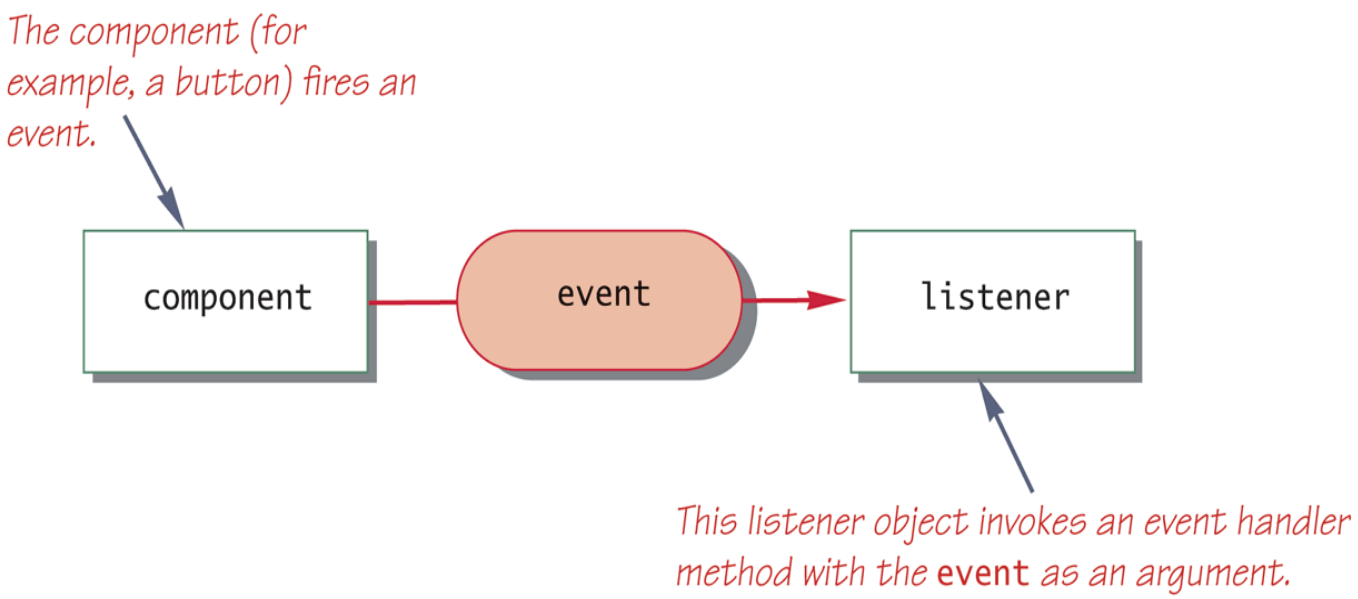
- A listener object performs some action in response to the event
  - A given component may have any number of listeners
  - Each listener may respond to a different kind of event, or multiple listeners might may respond to the same events

Event Handlers

- A listener object has methods that specify what will happen when events of various kinds are received by it
  - These methods are called *event handlers*
- The programmer using the listener object will define or redefine these event-handler methods

Event Firing and an Event Listener

Display 17.1 Event Firing and an Event Listener



Event-Driven Programming

- Event-driven programming is very different from most programming seen up until now
  - So far, programs have consisted of a list of statements executed in order
  - When that order changed, whether or not to perform certain actions (such as repeat statements in a loop, branch to another statement, or invoke a method) was controlled by the logic of the program

Event-Driven Programming

- In event-driven programming, objects are created that can fire events, and listener objects are created that can react to the events
- The program itself no longer determines the order in which things can happen
  - Instead, the events determine the order

Event-Driven Programming

- In an event-driven program, the next thing that happens depends on the next event
- In particular, *methods are defined that will never be explicitly invoked in any program*
  - Instead, methods are invoked automatically when an event signals that the method needs to be called

Events and Listener Interfaces

- In Java, every event type has a matching listener interface.
  - For MouseEvents implement the MouseListener interface
  - For WindowEvents implement the WindowListener interface
  - Etc., etc. etc.
- Some interfaces (e.g. MouseListener) have multiple methods because the event itself can happen in different ways (mouse pressed, mouse released, mouseMoved, etc)
- JButtons fire off ActionEvents, so in our example, we need to implement the ActionListener interface

ActionListener Interface

- The ActionListener interface consists of a single method

```
public void actionPerformed(ActionEvent event)
```

This is the method that we want to be called when the button is pushed.

This code snippet is a simple implementation of **`actionPerformed`** that changes the button's text

```
public void actionPerformed(ActionEvent ae)
{
        button1.setText("I've been pushed");
}
```

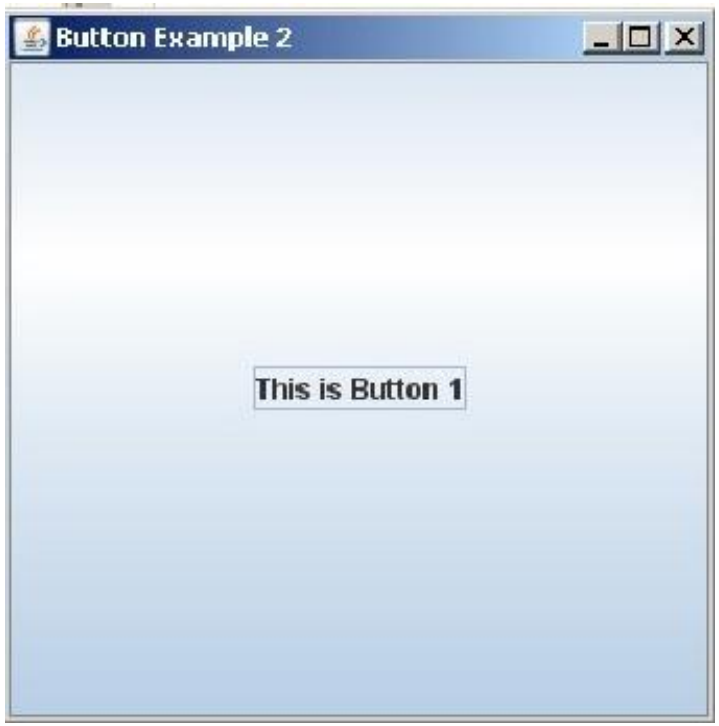
Registering as a Listener

- Now that we've written the **`actionPerformed`** method, we need to tell the button to call it when the button is pushed. To do that, we need to register our object as a listener for the button. To do so we call the button's **`addActionListener`** method

```
button1.addActionListener(this);
```

`ButtonExample2.java` has the complete application that shows the working GUI

Pushing the Button



Before

After

Other Common Components

- JLabel – an uneditable line of text
- JTextField – a single line of editable text
- JTextArea – a multi-line area of editable text
- JList – a list for item selection
- JCheckBox – a single on/off choice
- Etc, etc, etc

How are components arranged

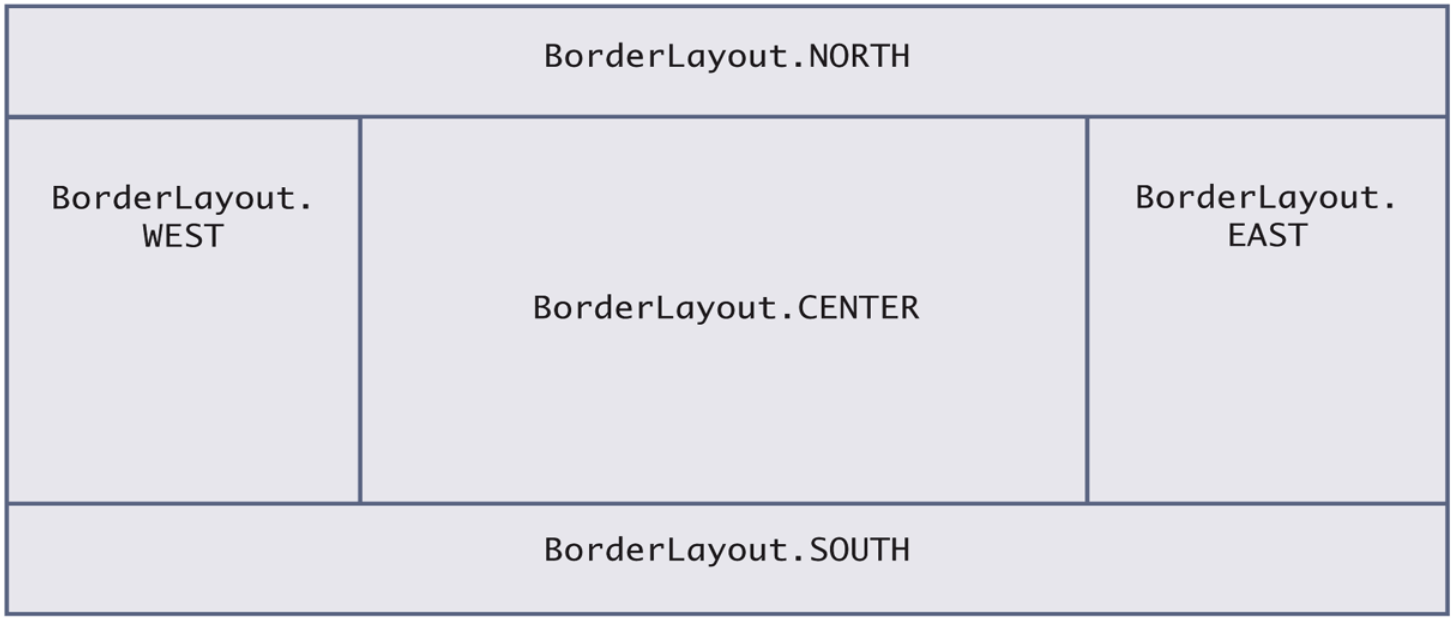
- Our example GUI has just one component, the button. How do put more than one component into a JFrame?
- By default, a JFrame has 5 regions to which components can be added. These regions are defined by the BorderLayout manager.

BorderLayout

BorderLayout is the default layout for a JFrame. We'll look at other layouts later.

When we add a component to a JFrame, we specify which region you want the component placed. Therefore components can be added in any order.

Display 17.8 **BorderLayout Regions**



Using the BorderLayout

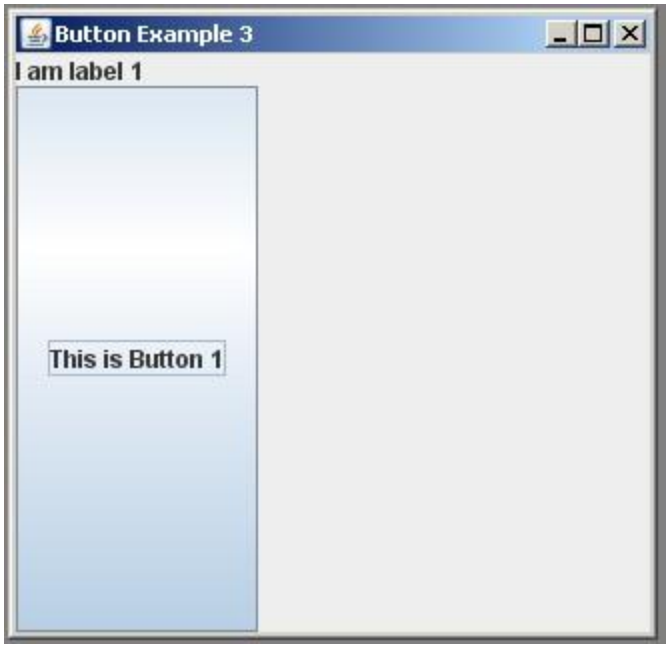
- This code snippet places a JButton in the WEST region and a JLabel in the NORTH region.

```
// create a button and add "this" object as a
listener
JButton button1 = new JButton("This is Button 1");
button1.addActionListener(this);
frame.add( BorderLayout.WEST, button1 );

// create and add the label
JLabel label1 = new JLabel("I am label 1");
frame.add( BorderLayout.NORTH, label1);
```

ButtonExample3

ButtonExample3.java contains the complete working application in which pushing the button changes the label's text.



Before

After

Another Button

- Let's add a 2nd button to our GUI that will change the label's text in a different way.
- Doesn't a 2nd button mean a 2nd event?
- Absolutely right!
- How do we get ActionEvents from two different buttons when each button does something different?

Two Buttons - Option 1

- Implement two **`actionPerformed( )`** methods in our class.
- Is this possible?
- If it is possible, how would each button know which of the two **`actionPerformed`**( ) methods to call?

Two Buttons – Option 2

- Register our listener object with both buttons.

```
button1.addActionListener( this );
button2.addActionListener( this );
```

- But then the **`actionPerformed( )`** method would have to figure out which button called it.
- Not very OO – having one event handler doing multiple things

Two Button – Option 3

- Create separate ActionListener classes so that each one can implement **`actionPerformed`**
  - This would eventually lead to many small classes
- But these separate classes won't have access to the frame, buttons, and label defined in our class.
- Wouldn't it be nice if you could have two listener classes that can access the instance variables inside of our class? Does this sound familiar?

Two Buttons – Option 4

- Implement separate listener classes as inner classes within the main GUI class.

```
// the main GUI class
public class ButtonExample4
{
   // inner listener class for button1
   private class Listener1 implements ActionListener
   {
       public void actionPerformed(ActionEvent ae)
       { label1.setText("Button 1 Pushed"); }
   }
   // inner listener class for button2
   private class Listener2 implements ActionListener
   {
       public void actionPerformed(ActionEvent ae)
       { label1.setText("Button 2 Pushed"); }
   }
   // the rest of ButtonExample4 class
```

Option 4 cont'd

// somewhere in a method of ButtonExample4

// register the inner classes with the buttons

```
button1.addActionListener( new Listener1( ) );
button2.addActionListener( new Listener2( ) );
```

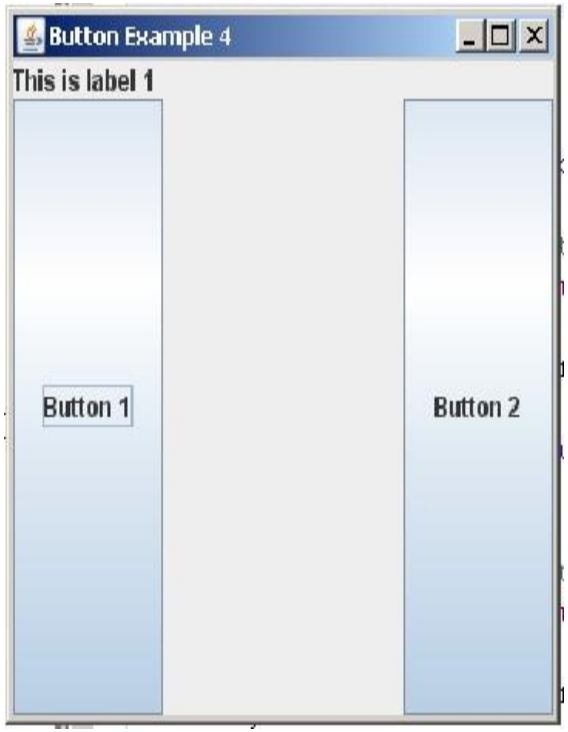
ButtonExample4.java has the complete working application. Screen shots appear on the next slide.

Option 4b: Anonymous Classes

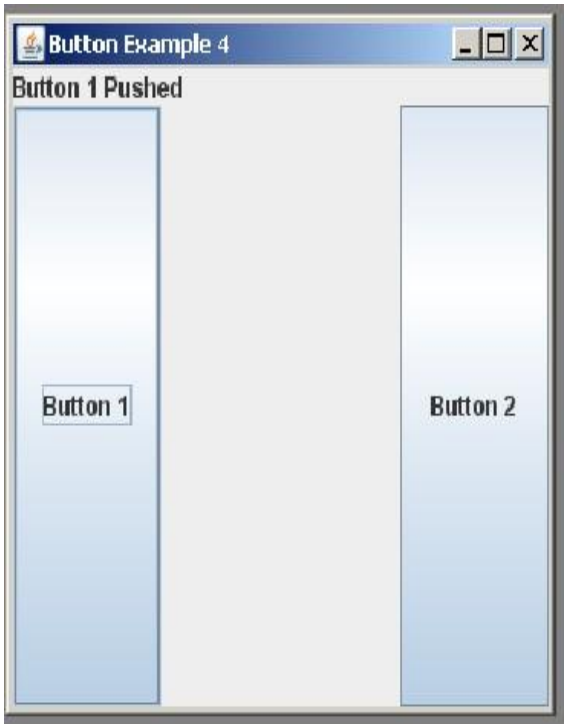
- Create listener classes as anonymous inner classes:

```
// the main GUI class
public class ButtonExample4b
{
    // … code to create GUI components
    button1.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            label1.setText("Button 1 Pushed");
        }
    }
    button2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            label1.setText("Button 2 Pushed");
        }
    }
    // the rest is as for ButtonExample4 class
}
```

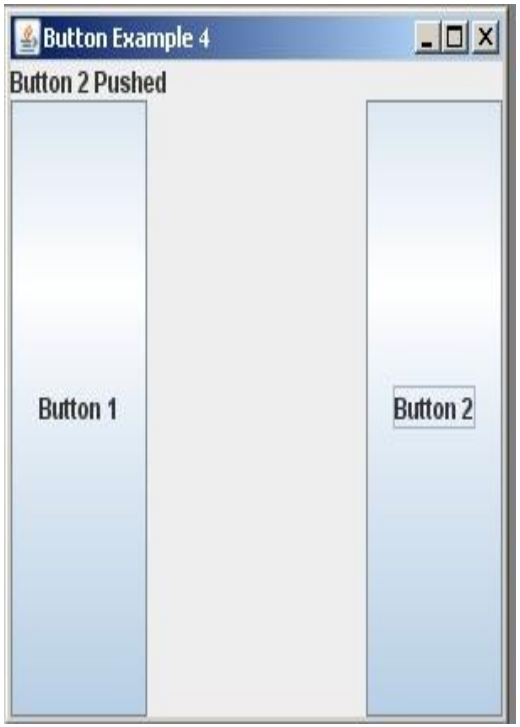
ButtonExample4



Initial Screen



After pushing Button 1



After pushing Button 2

Other Layout Mangers

- As we've seen, the default layout manager for a JFrame is the BorderLayout. But this is not the only layout manager available.
- Two other common layout managers are
  - FlowLayout
  - GridLayout

FlowLayout Manager

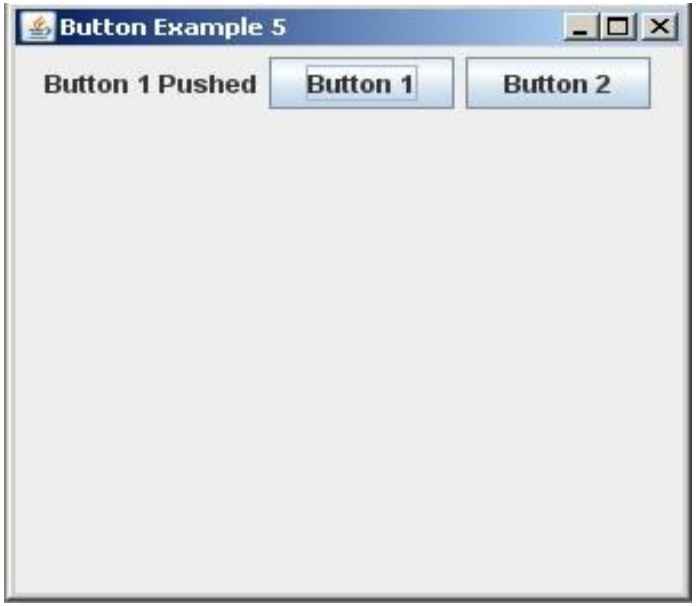
- The FlowLayout manager is the simplest. It arranges components one after the other, left to right and top to bottom in the order they are added to the JFrame.
- Components may be reconfigured if the JFrame is resized.
- This statement specifies the FlowLayout for the JFrame

```
frame.setLayout(new FlowLayout( ));
```

FlowLayout Example

- ButtonExample5.java has the complete application that creates the frame below using the FlowLayout manager.

Initial Screen



After pushing Button 1

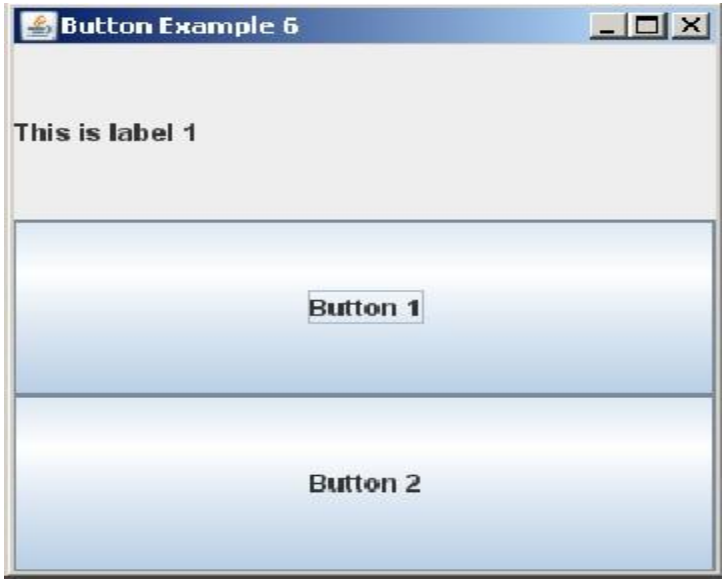
GridLayout Manager

- The GridLayout manager arranges components in a 2-dimensional grid with the number or rows and columns that you specify
- Components are placed in the grid cells left to right and top to bottom in the order in which they are added.
- All components are sized to fill the grid cell in which they are placed.
- This code specifies a GridLayout with 1 row and 3 columns which can be used to arrange components horizontally

```
frame.setLayout(new GridLayout(1,3));
```

GridLayout Example

- ButtonExample6.java contains the complete application which created the frame below using the GridLayout manager. Note that the components fill the grid cells



JPanel

- A GUI is often organized in a hierarchical fashion, with containers called *panels* inside other containers
- A panel is an object of the **`JPanel`** class that serves as a simple container
  - It is used to group smaller objects into a larger component (the panel)
  - One of the main functions of a **`JPanel`** object is to subdivide a **`JFrame`** or other container

Panels

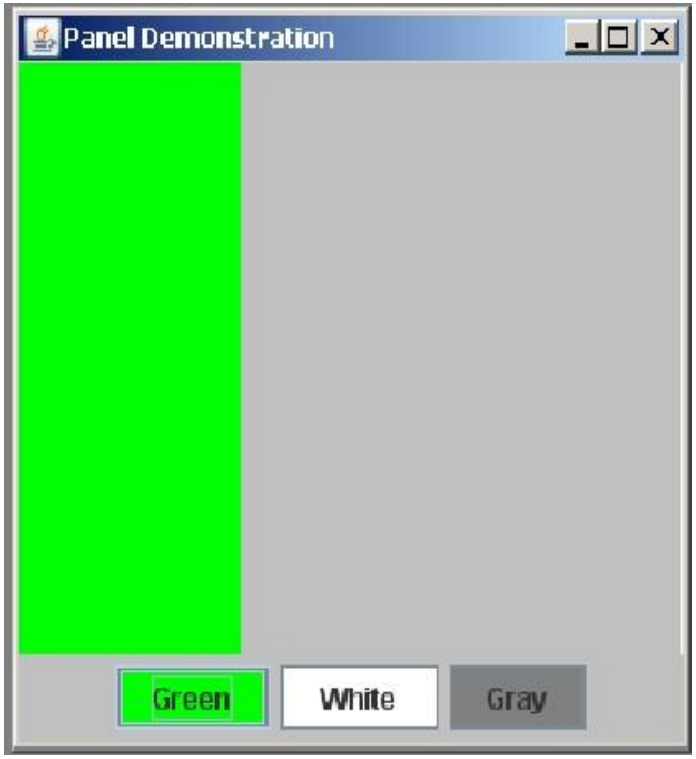
- Both a **`JFrame`** and each panel in a **`JFrame`** can use different layout managers
  - Additional panels can be added to each panel, and each panel can have its own layout manager
  - This enables almost any kind of overall layout to be used in a GUI
  - For example
    - Multiple components can be placed in one region of a BorderLayout by first adding the components to a JPanel, then adding the JPanel to the BorderLayout region.

JPanel Demo

- The PanelDemo code from the text (display 17.11, starting on page 957) shows the use of JPanels for subdividing a JFrame.
- PanelDemo.java contains the code from the text.

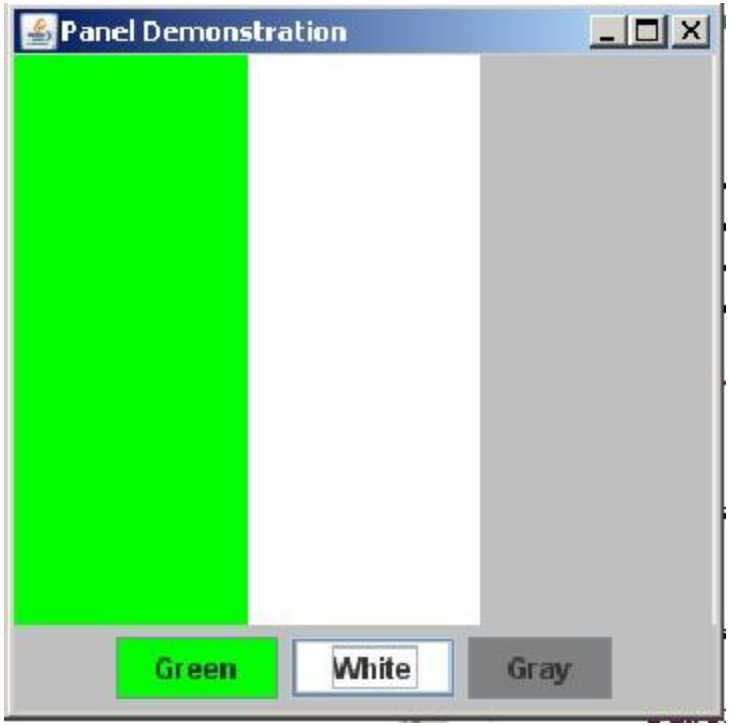
PanelDemo

Initial Screen

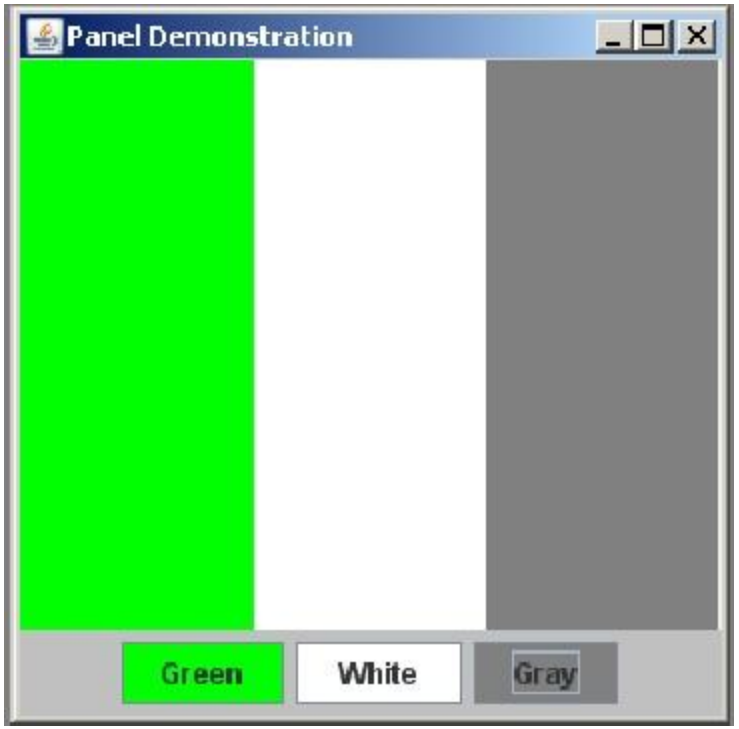


After pushing Green Button

PanelDemo (cont'd)



After pushing the White button



After pushing the Gray button

The `Container` Class

- Any class that is a descendent class of the class Container is considered to be a container class
  - The `Container` class is found in the `java.awt` package, not in the Swing library
- Any object that belongs to a class derived from the `Container` class (or its descendents) can have components added to it
- The classes `JFrame` and `JPanel` are descendent classes of the class `Container`
  - Therefore they and any of their descendents can serve as a container

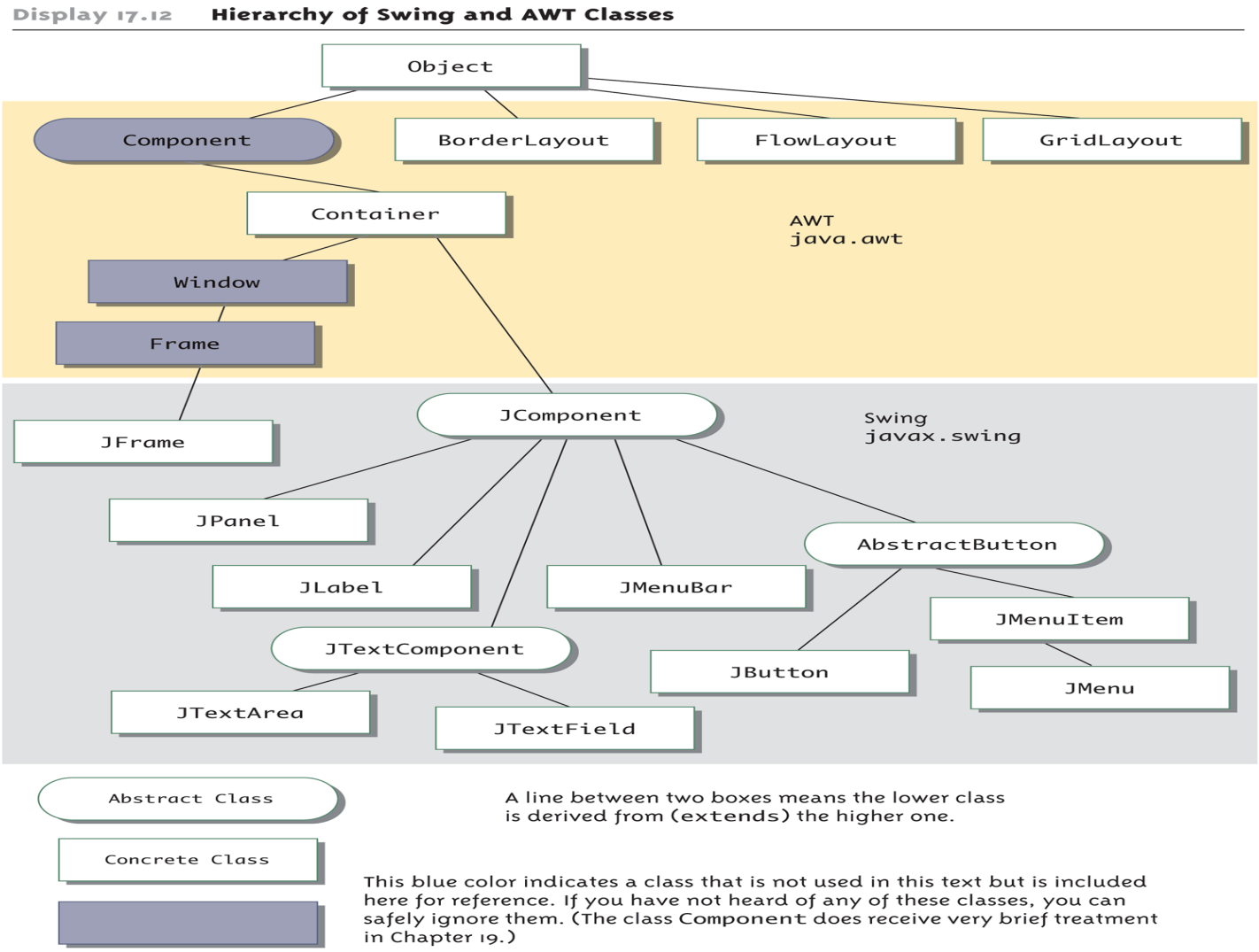
The `JComponent` Class

- Any descendent class of the class `JComponent` is called a *component class*
  - Any `JComponent` object or *component* can be added to any container class object
  - Because it is derived from the class `Container`, a `JComponent` can also be added to another `JComponent`

Objects in a Typical GUI

- Almost every GUI built using Swing container classes will be made up of three kinds of objects:
  1. The container itself, probably a panel or window-like object
  2. The components added to the container such as labels, buttons, and panels
  3. A layout manager to position the components inside the container

Hierarchy of Swing and AWT Classes



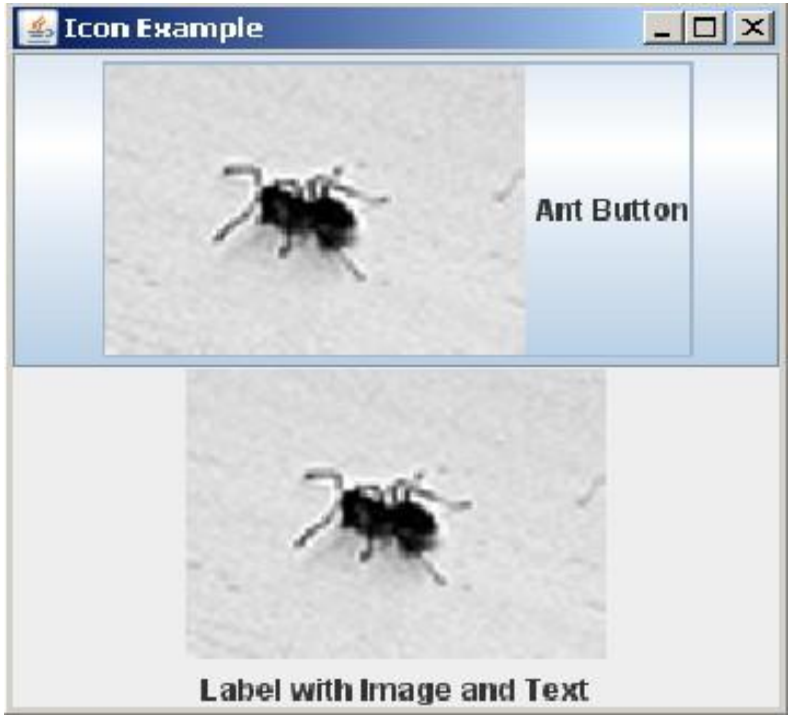
Icons

- An icon is a small picture
- Components such as JButtons and JLabels may be decorated with icons that are created from .jpg, .gif or .png files
- This code snippet creates an ImageIcon object then uses it to decorate JLabels

```
ImageIcon icon = new ImageIcon("ant.jpg");
JLabel label1 = new JLabel(icon);
JLabel label2 = new JLabel("Image & Text",
    icon,  JLabel.CENTER);
```

Icon Example

- IconExample.java contains the complete application that created the frame below



Basic Swing Graphics

- Swing allows applications to draw directly onto the surface of any component.
- This drawing process is known as "painting.
- Each component inherits the `paint( )` method from the `Component` class.
- `paint( )` is called by the system whenever a component needs to be displayed on the sreen

More paint( )

- paint( ) calls the methods **paintComponent, paintBorder, paintChildren** in that order.
- To paint the surface of a component
  - Create a subclass (extend) the component
  - Override the paintComponent( ) method

Overriding paintComponent

```
// This is part of the PaintDemo1 class
// PaintPanel extends the JPanel so that it can override
// paintComponent( ) to draw an  image and then overwrite
// a portion of the image with an oval

private class PaintPanel extends JPanel
{
   // called by the system when necessary
   public void paintComponent(Graphics g )
   {
        super.paintComponent(g);  // always the first thing
        ImageIcon antIcon = new ImageIcon("ant.jpg");
        Image antImage = antIcon.getImage( );
        g.drawImage(antImage, 3, 4, this);
        g.fillOval(3, 4, 50, 50);  // x, y, width, height
   }
}
// more of PaintDemo1 class follows
```

Requesting to be Painted

- An application can request that its components be repainted by calling the **`repaint( )`** method for the component.
- Because painting the screen is relatively slow and unimportant, repaint( ) only requests that the system calls paint( ) as soon as possible.

repaint( ) example

- This code snippet requests repainting a circle on a panel periodically as part of a simple animation.

```
public void launch( )
{
   frame.setVisible(true);
   for (int i = 0; i < 150; i++)
   {
        // change the circle's coordinates
        ++x; ++y;
        // force the panel to repaint itself
        paintPanel.repaint();
        try {  // wait 50ms
               Thread.sleep( 50 );
        }catch (Exception ex) {}
   }
}
```

Animation

SimpleAnimation.java contains the complete application which uses the code snippet from the previous slide. The animation moves a blue circle from the top left corner of the frame toward the bottom right corner. Three screenshots on the next slide show how the animation progress.

Animation (cont'd)