Java Swing

Introduction to Swing

- A *GUI (graphical user interface)* is a windowing system that interacts with the user
- The Java *AWT (Abstract Window Toolkit)* package is the original Java package for doing *GUIs*
- 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***

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 to another object know 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 may respond to the same events

Exception Objects

- *Where have we seen this "Even-Driven" programming before?*

- An exception object is an event
  - The throwing of an exception is an example of firing an event

- The listener for an exception object is the **`catch`** block that catches the event

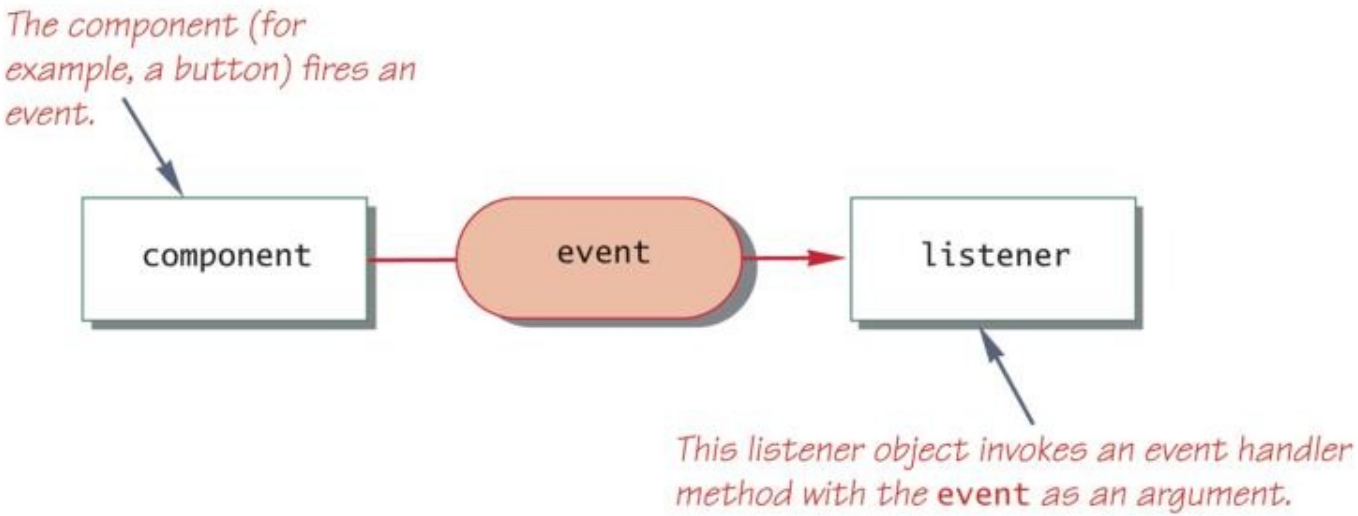
Event Handlers

- 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 (e.g., which button or menu will be selected by the user)

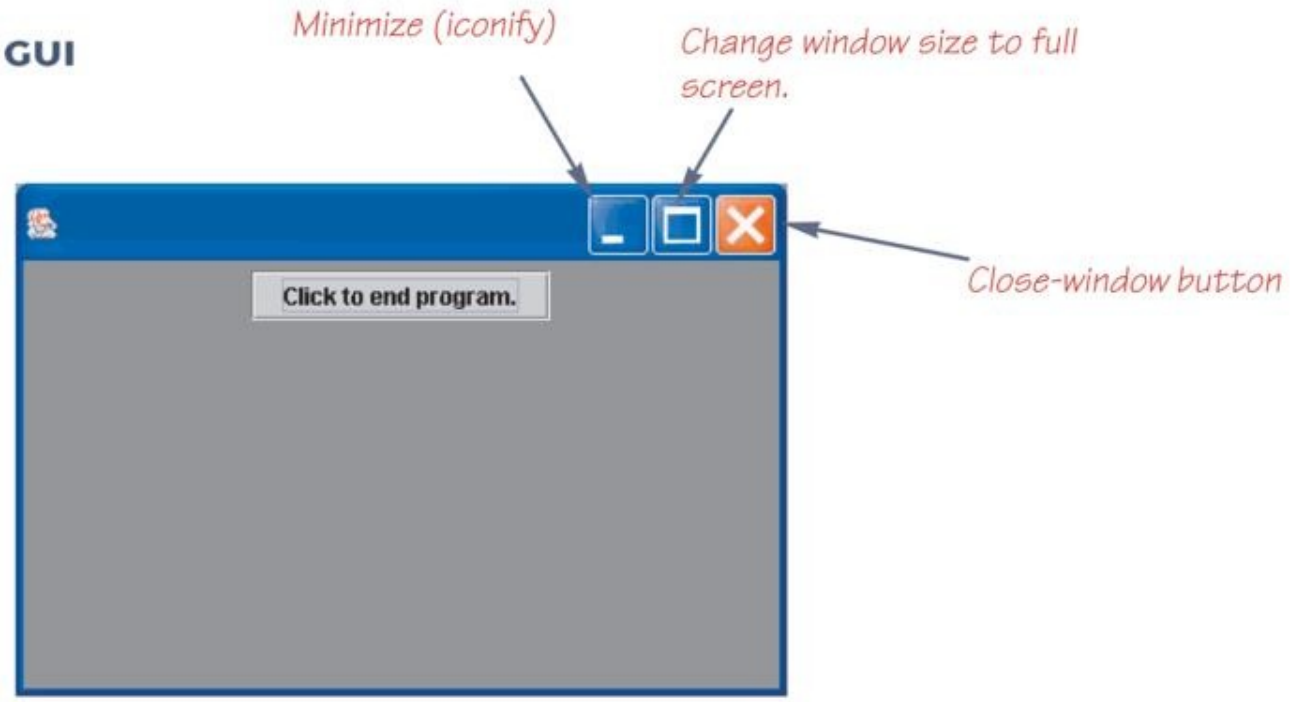
- In particular, ***methods are defined that will never be explicitly invoked in any program***
  - Instead, methods are invoked automatically when an event signals that the method needs to be called

A SIMPLE WINDOW

A First Swing Demonstration (Part 4 of 4)

Display 17.2 A First Swing Demonstration Program

RESULTING GUI



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

```
JFrame firstWindow = new JFrame();
```

- A **`JFrame`** can have components added to it, such as buttons, menus, and text labels
  - These components can be programmed for action

```
firstWindow.add(endButton);
```

  - It can be made visible using the **`setVisible`** method

```
firstWindow.setVisible(true);
```

A First Swing Demonstration (Part 1 of 4)

Display 17.2 A First Swing Demonstration Program

```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;

3  public class FirstSwingDemo
4  {
5      public static final int WIDTH = 300;
6      public static final int HEIGHT = 200;

7      public static void main(String[] args)
8      {
9          JFrame firstWindow = new JFrame();
10         firstWindow.setSize(WIDTH, HEIGHT);
```

This program is not typical of the style we will use in Swing programs.

(continued)

Create a JFrame object

A First Swing Demonstration (Part 1 of 4)

Display 17.2 A First Swing Demonstration Program

```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;

3  public class FirstSwingDemo
4  {
5      public static final int WIDTH = 300;
6      public static final int HEIGHT = 200;

7      public static void main(String[] args)
8      {
9          JFrame firstWindow = new JFrame();
10         firstWindow.setSize(WIDTH, HEIGHT);
```

This program is not typical of the style we will use in Swing programs.

(continued)

Set window size with numbers of pixels

Pixels and the Relationship between Resolution and Size

- A *pixel* is the smallest unit of space on a screen
  - Both the size and position of Swing objects are measured in pixels
  - The more pixels on a screen, the greater the screen resolution
- A high-resolution screen of fixed size has many pixels
  - Therefore, each one is very small
- A low-resolution screen of fixed size has fewer pixels
  - Therefore, each one is much larger
- Therefore, a two-pixel figure on a low-resolution screen will look larger than a two-pixel figure on a high-resolution screen

A First Swing Demonstration (Part 2 of 4)

Display 17.2 A First Swing Demonstration Program

```
11          firstWindow.setDefaultCloseOperation(
12                                  JFrame.DO_NOTHING_ON_CLOSE);

13          JButton endButton = new JButton("Click to end program.");
14          EndingListener buttonEar = new EndingListener();
15          endButton.addActionListener(buttonEar);
16          firstWindow.add(endButton);

17          firstWindow.setVisible(true);
18      }
19  }
```

This is the file `FirstSwingDemo.java`.

(continued)

Create a button object with text

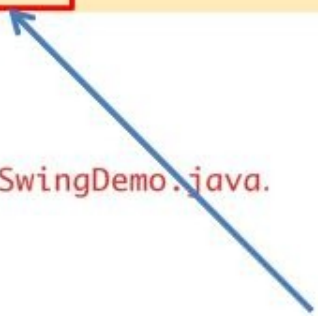
A First Swing Demonstration (Part 2 of 4)

Display 17.2 A First Swing Demonstration Program

```
11          firstWindow.setDefaultCloseOperation(
12                                  JFrame.DO_NOTHING_ON_CLOSE);

13          JButton endButton = new JButton("Click to end program.");
14          EndingListener buttonEar = new EndingListener();
15          endButton.addActionListener(buttonEar);
16          firstWindow.add(endButton);

17          firstWindow.setVisible(true);
18      }
19  }
```

This is the file `FirstSwingDemo.java`.

(continued)



Add the button to the JFrame object

Buttons

- 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 endButton = new
            JButton("Click to end program.");
firstWindow.add(endButton);
```

A First Swing Demonstration (Part 2 of 4)

Display 17.2 A First Swing Demonstration Program

```
11          firstWindow.setDefaultCloseOperation(
12                                  JFrame.DO_NOTHING_ON_CLOSE);

13          JButton endButton = new JButton("Click to end program.");
14          EndingListener buttonEar = new EndingListener();
15          endButton.addActionListener(buttonEar);
16          firstWindow.add(endButton);

17          firstWindow.setVisible(true);
18      }
19  }
```

This is the file `FirstSwingDemo.java`.

(continued)

Associate an Even Lister

A First Swing Demonstration (Part 3 of 4)

Display 17.2 A First Swing Demonstration Program

This is the file `EndingListener.java`.

```
1  import java.awt.event.ActionListener;
2  import java.awt.event.ActionEvent;

3  public class EndingListener implements ActionListener
4  {
5      public void actionPerformed(ActionEvent e)
6      {
7          System.exit(0);
8      }
9  }
```

(continued)

Define Even Listener

A First Swing Demonstration (Part 2 of 4)

Display 17.2 A First Swing Demonstration Program

```
11          firstWindow.setDefaultCloseOperation(
12                                  JFrame.DO_NOTHING_ON_CLOSE);

13          JButton endButton = new JButton("Click to end program.");
14          EndingListener buttonEar = new EndingListener();
15          endButton.addActionListener(buttonEar);
16          firstWindow.add(endButton);

17          firstWindow.setVisible(true);
18      }
19  }
```

This is the file `FirstSwingDemo.java`.

(continued)

Make the JFrame object visible

Pitfall: Forgetting to Program the Close-Window Button

- The following lines from the **`FirstSwingDemo`** program ensure that when the user clicks the *close-window button*, nothing happens

```
firstWindow.setDefaultCloseOperation(
                                JFrame.DO_NOTHING_ON_CLOSE);
```

- If this were not set, the default action would be **`JFrame.HIDE_ON_CLOSE`**
  - This would make the window invisible and inaccessible, but would not end the program
  - Therefore, given this scenario, there would be no way to click the "Click to end program" button
- Note that the close-window and other two accompanying buttons are part of the **`JFrame`** object, and not separate buttons

Change a few places and see what will happen?

Action Listeners and Action Events

- Clicking a button fires an event
- The event object is "sent" to another object called a listener
  - This means that a method in the listener object is invoked automatically
  - Furthermore, it is invoked with the event object as its argument
- In order to set up this relationship, a GUI program must do two things
  1. It must specify, for each button, what objects are its listeners, i.e., it must register the listeners
  2. It must define the methods that will be invoked automatically when the event is sent to the listener

A First Swing Demonstration (Part 3 of 4)

Display 17.2 A First Swing Demonstration Program

This is the file `EndingListener.java`.

```
1  import java.awt.event.ActionListener;
2  import java.awt.event.ActionEvent;

3  public class EndingListener implements ActionListener
4  {
5      public void actionPerformed(ActionEvent e)
6      {
7          System.exit(0);
8      }
9  }
```

(continued)

Define Even Listener

Action Listeners and Action Events

```
EndingListener buttonEar = new
                           EndingListener();
endButton.addActionListener(buttonEar);
```

- Above, a listener object named **`buttonEar`** is created and registered as a listener for the button named **`endButton`**
  - Note that a button fires events known as *action events*, which are handled by listeners known as *action listeners*

Action Listeners and Action Events

- Different kinds of components require different kinds of listener classes to handle the events they fire
- An action listener is an object whose class implements the **`ActionListener`** interface
  - The **`ActionListener`** interface has one method heading that must be implemented

```
public void actionPerformed(ActionEvent e)
```

Action Listeners and Action Events

```
public void actionPerformed(ActionEvent e)
{
   System.exit(0);
}
```

- The **EndingListener** class defines its **actionPerformed** method as above
  - When the user clicks the **endButton**, an action event is sent to the action listener for that button
  - The **EndingListener** object **buttonEar** is the action listener for **endButton**
  - The action listener **buttonEar** receives the action event as the parameter **e** to its **actionPerformed** method, which is automatically invoked
  - Note that **e** must be received, even if it is not used

Pitfall: Changing the Heading for `actionPerformed`

- When the **`actionPerformed`** method is implemented in an action listener, its header must be the one specified in the **`ActionListener`** interface
  - It is already determined, and may not be changed
  - Not even a throws clause may be added

```
public void actionPerformed(ActionEvent e)
```

- The only thing that can be changed is the name of the parameter, since it is just a placeholder
  - Whether it is called **`e`** or something else does not matter, as long as it is used consistently within the body of the method

A Better Version of the Previous indow

The Normal Way to Define a **`JFrame`** (Part 1 of 4)

Display 17.4 **The Normal Way to Define a `JFrame`**

```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;


3  public class FirstWindow extends JFrame
4  {
5      public static final int WIDTH = 300;
6      public static final int HEIGHT = 200;

7      public FirstWindow()
8      {
9          super();
10         setSize(WIDTH, HEIGHT);

11         setTitle("First Window Class");
```

(continued)

The Normal Way to Define a `JFrame` (Part 2 of 4)

Display 17.4 The Normal Way to Define a `JFrame`

```
12            setDefaultCloseOperation(
13                                    JFrame.DO_NOTHING_ON_CLOSE);

14            JButton endButton = new JButton("Click to end program.");
15            endButton.addActionListener(new EndingListener());
16            add(endButton);
17        }
18    }
```

The class `EndingListener` *is defined in Display 17.2.*

This is the file `FirstWindow.java`.

(continued)

The Normal Way to Define a **`JFrame`** (Part 3 of 4)

Display 17.4 The Normal Way to Define a `JFrame`

This is the file `DemoWindow.java`.

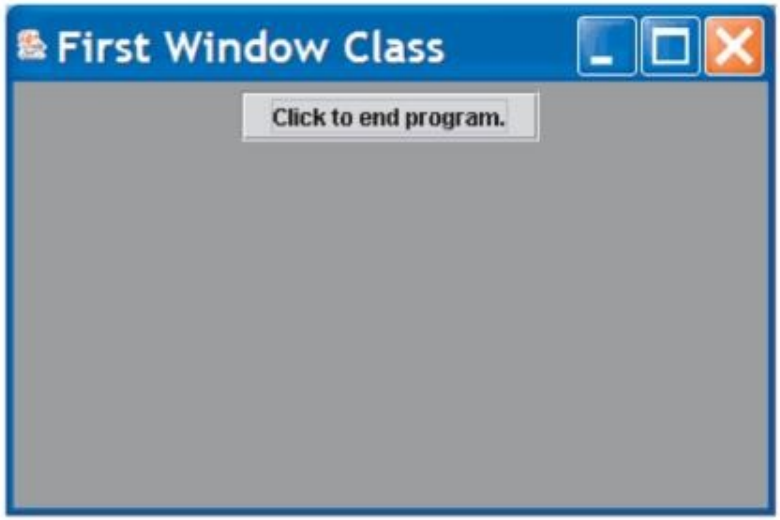
```
1  public class DemoWindow
2  {
3      public static void main(String[] args)
4      {
5          FirstWindow w = new FirstWindow();
6          w.setVisible(true);
7      }
8  }
```

(continued)

The Normal Way to Define a `JFrame` (Part 4 of 4)

Display 17.4 **The Normal Way to Define a `JFrame`**

RESULTING GUI



What Can be Summarized?

Derive a new container class from, say, JFrame

In the derived class
Add the interface objects, such as buttons and others
Associate a listener object for each interface object

Define listener classes to handle possible events fired by the interface objects added in the window

In the main function
Create the object of the derived window class
Define a constructor that sets up the title and size of the window
Launch the interface by setting it as visible

Labels

- A *label* is an object of the class **`JLabel`**
  - Text can be added to a **`JFrame`** using a label
  - The text for the label is given as an argument when the **`JLabel`** is created
  - The label can then be added to a **`JFrame`**

```
JLabel greeting = new JLabel("Hello");
add(greeting);
```

Color

- In Java, a *color* is an object of the class **`Color`**
  - The class **`Color`** is found in the **`java.awt`** package
  - There are constants in the **`Color`** class that represent a number of basic colors

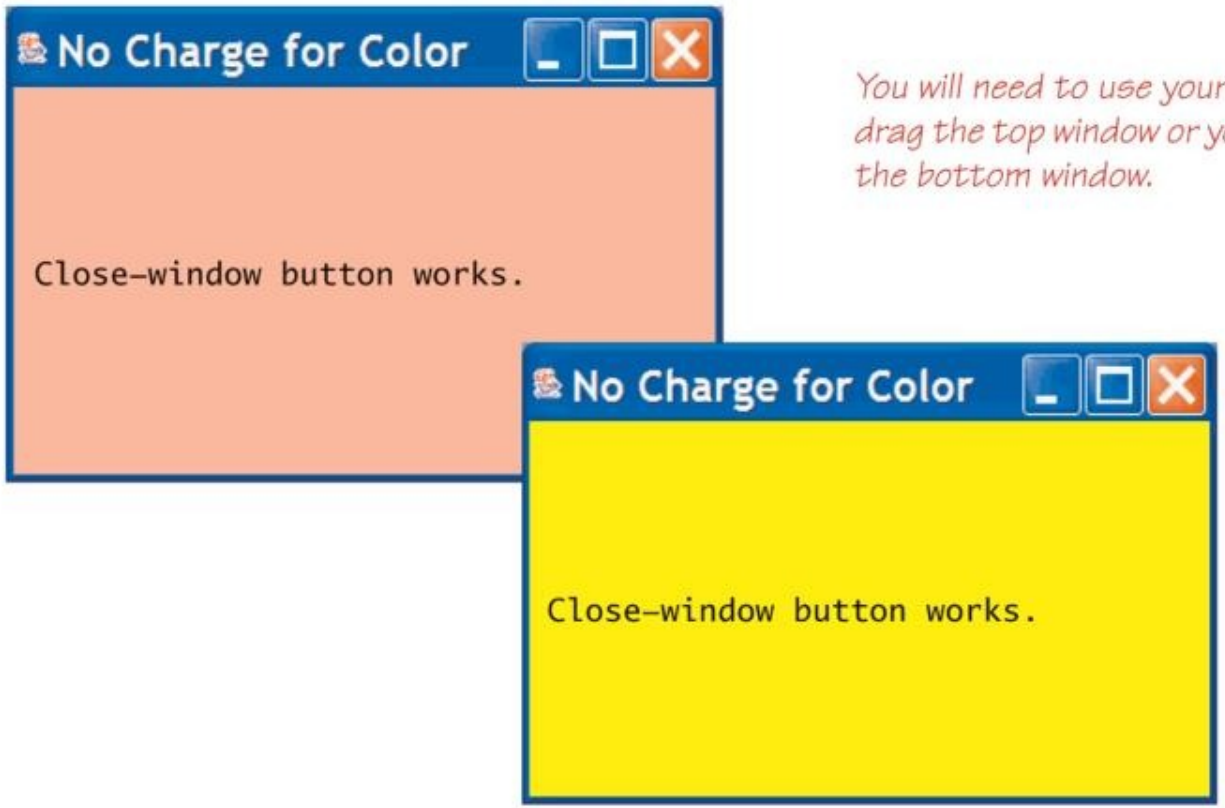
- **`JFrame`** can not be colored directly
  - Instead, a program must color something called the ***content pane*** of the **`JFrame`**
  - Since the content pane is the "**inside**" of a **`JFrame`**, coloring the content pane has the effect of coloring the inside of the **`JFrame`**
  - Therefore, the ***background color*** of a **`JFrame`** can be set using the following code:

```
getContentPane().setBackground(Color);
```

A `JFrame` with Color (Part 4 of 4)

Display 17.6 **A `JFrame` with Color**

RESULTING GUI



You will need to use your mouse to drag the top window or you will not see the bottom window.

The Color Constants

Display 17.5 The Color Constants

`Color.BLACK`	`Color.MAGENTA`
`Color.BLUE`	`Color.ORANGE`
`Color.CYAN`	`Color.PINK`
`Color.DARK_GRAY`	`Color.RED`
`Color.GRAY`	`Color.WHITE`
`Color.GREEN`	`Color.YELLOW`
`Color.LIGHT_GRAY`	

The class `Color` is in the `java.awt` package.

The color can be customized…

A **JFrame** with Color (Part 1 of 4)

Display 17.6 **A JFrame with Color**

```
1   import javax.swing.JFrame;
2   import javax.swing.JLabel;
3   import java.awt.Color;

4   public class ColoredWindow extends JFrame
5   {
6       public static final int WIDTH = 300;
7       public static final int HEIGHT = 200;

8       public ColoredWindow(Color theColor)
9       {
10          super("No Charge for Color");
11          setSize(WIDTH, HEIGHT);
12          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

(continued)

A **JFrame** with Color (Part 2 of 4)

Display 17.6 **A JFrame with Color**

```
13          getContentPane().setBackground(theColor);

14          JLabel aLabel = new JLabel("Close-window button works.");
15          add(aLabel);
16      }

17      public ColoredWindow()
18      {
19          this(Color.PINK);
20      }
21  }
```

This is an invocation of the other constructor.

This is the file `ColoredWindow.java`.

(continued)

A **JFrame** with Color (Part 3 of 4)

Display 17.6 **A JFrame with Color**

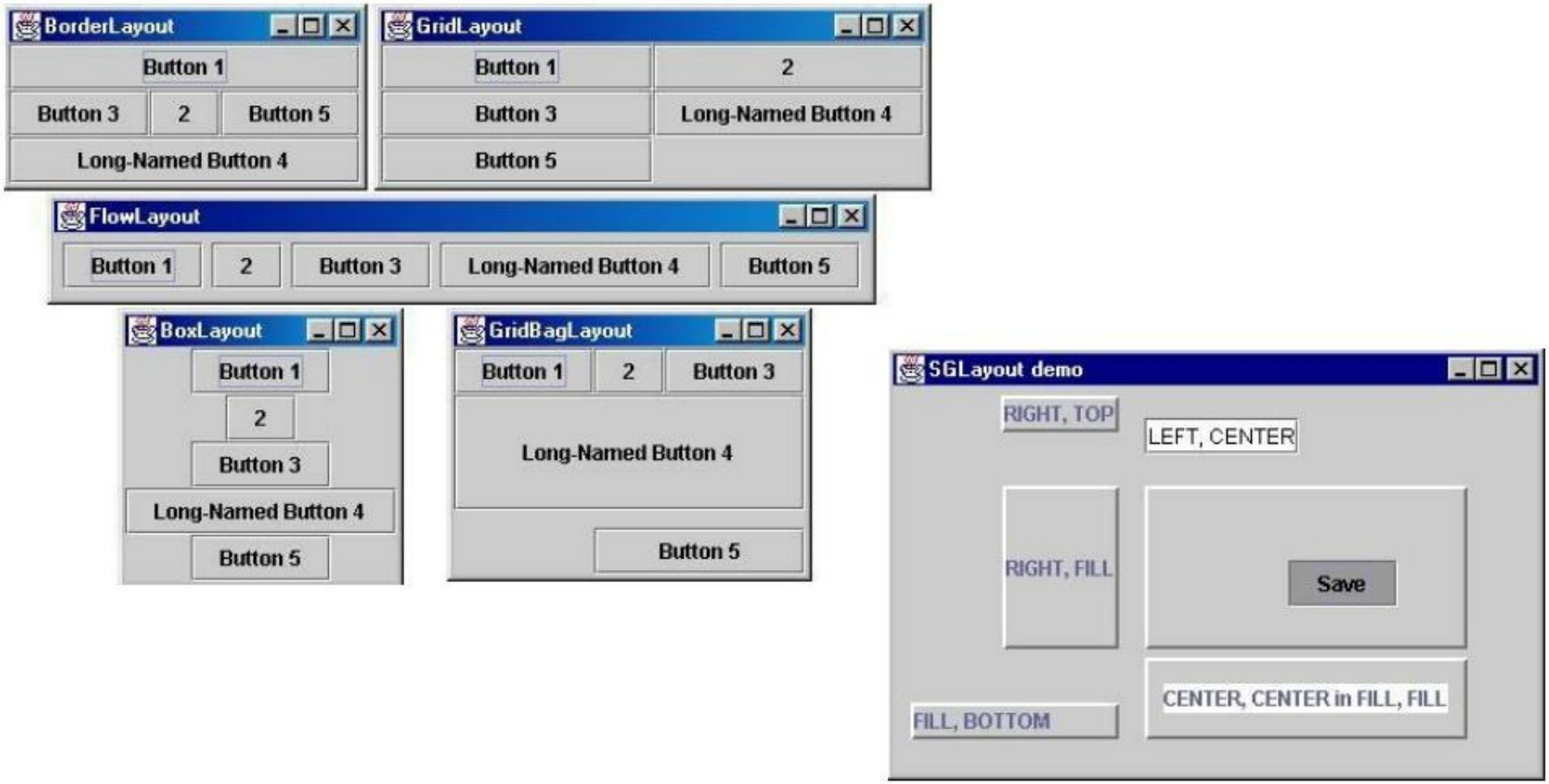
This is the file `ColoredWindow.java`.

```
1  import java.awt.Color;

2  public class DemoColoredWindow
3  {
4      public static void main(String[] args)
5      {
6          ColoredWindow w1 = new ColoredWindow();
7          w1.setVisible(true);

8          ColoredWindow w2 = new ColoredWindow(Color.YELLOW);
9          w2.setVisible(true);
10     }
11 }
```

(continued)

LAYOUT MANAGER

Layout Examples



Containers and Layout Managers

- Multiple components can be added to the content pane of a **`JFrame`** using the **`add`** method
  - However, the **`add`** method does not specify how these components are to be arranged

- To describe how multiple components are to be arranged, a *layout manager (**awt**)* is used
  - There are a number of layout manager classes such as **`BorderLayout`**, **`FlowLayout`**, and **`GridLayout`**
  - If a layout manager is not specified, a default layout manager is used (e.g., only one widget on the interface)

Flow Layout Managers

- The **`FlowLayout`** manager is the simplest layout manager

```
setLayout(new FlowLayout());
```

  - It arranges components one after the other, going from left to right
  - Components are arranged in the order in which they are added
- Since a location is not specified, the **`add`** method has only one argument when using the **`FlowLayoutManager`**

```
add.(label1);
```

Border Layout Managers

- A **`BorderLayout`** manager places the components that are added to a **`JFrame`** object into five regions
  - These regions are: **`BorderLayout.NORTH`**, **`BorderLayout.SOUTH`**, **`BorderLayout.EAST`**, **`BorderLayout.WEST`**, and **`BorderLayout.Center`**
- A **`BorderLayout`** manager is added to a **`JFrame`** using the **`setLayout`** method
  - For example:

```
setLayout(new BorderLayout());
```

The **BorderLayout** Manager (Part 1 of 4)

Display 17.7 **The BorderLayout Manager**

```
1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.BorderLayout;

4  public class BorderLayoutJFrame extends JFrame
5  {
6      public static final int WIDTH = 500;
7      public static final int HEIGHT = 400;

8      public BorderLayoutJFrame()
9      {
10         super("BorderLayout Demonstration");
11         setSize(WIDTH, HEIGHT);
12         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

(continued)

The **BorderLayout** Manager (Part 2 of 4)

Display 17.7 **The BorderLayout Manager**

```
13          setLayout(new BorderLayout());

14          JLabel label1 = new JLabel("First label");
15          add(label1, BorderLayout.NORTH);

16          JLabel label2 = new JLabel("Second label");
17          add(label2, BorderLayout.SOUTH);

18          JLabel label3 = new JLabel("Third label");
19          add(label3, BorderLayout.CENTER);
20      }
21  }
```

This is the file `BorderLayoutJFrame.java`.

(continued)

The **BorderLayout** Manager (Part 3 of 4)

Display 17.7 **The BorderLayout Manager**

This is the file BorderLayoutDemo.java.

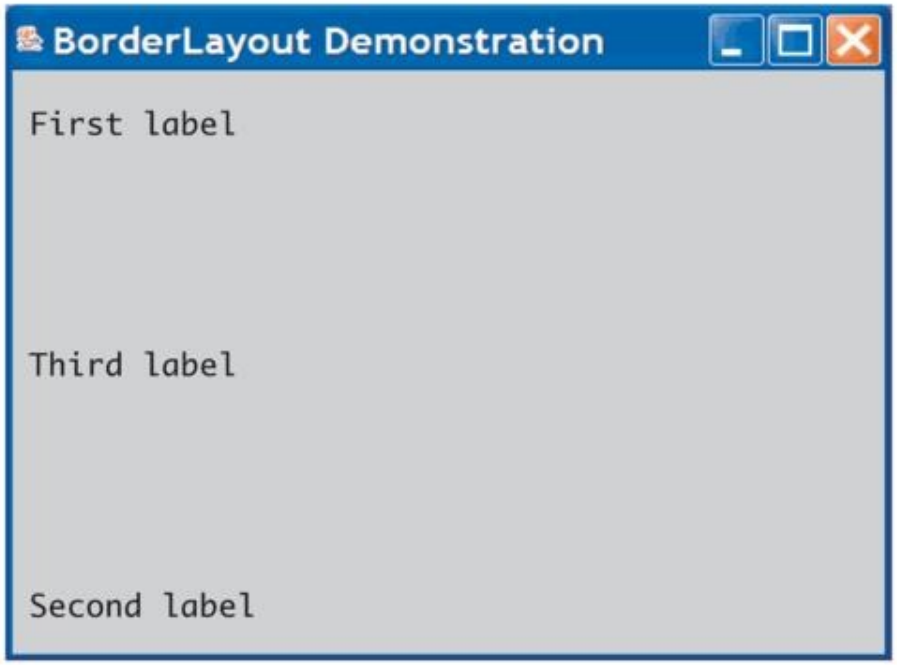
```
1  public class BorderLayoutDemo
2  {
3      public static void main(String[] args)
4      {
5          BorderLayoutJFrame gui = new BorderLayoutJFrame();
6          gui.setVisible(true);
7      }
8  }
```

(continued)

The `BorderLayout` Manager (Part 4 of 4)

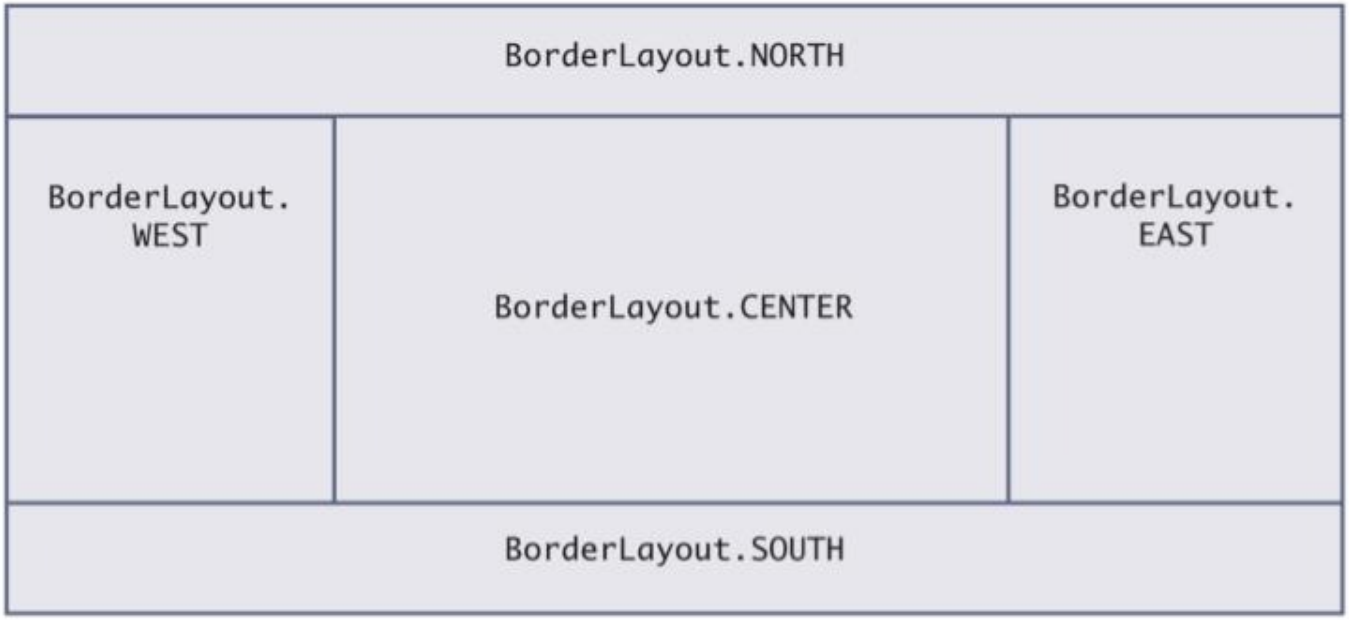
Display 17.7 **The BorderLayout Manager**

RESULTING GUI



BorderLayout Regions

Display 17.8 **BorderLayout Regions**



Border Layout Managers

- The previous diagram shows the arrangement of the five border layout regions
  - Note: None of the lines in the diagram are normally visible
- When using a **`BorderLayout`** manager, the location of the component being added is given as a second argument to the **`add`** method

```
add(label1, BorderLayout.NORTH);
```

  - Components can be added in any order since their location is specified

Grid Layout Managers

- A **`GridLayout`** manager arranges components in a two-dimensional grid with some number of rows and columns

```
setLayout(new GridLayout(rows, columns));
```

  - Each entry is the same size
  - The two numbers given as arguments specify the number of rows and columns
  - Each component is stretched so that it completely fills its grid position

  - Note: None of the lines in the diagram are normally visible

Grid Layout Managers

- When using the **`GridLayout`** class, the method **`add`** has only one argument

```
add(label1);
```

  - Items are placed in the grid from left to right
  - The top row is filled first, then the second, and so forth
  - Grid positions may not be skipped
- Note the use of a **`main`** method in the GUI class itself in the following example
  - This is often a convenient way of demonstrating a class

The **`GridLayout`** Manager (Part 1 of 4)

Display 17.9 **The GridLayout Manager**

```
1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.GridLayout;

4  public class GridLayoutJFrame extends JFrame
5  {
6      public static final int WIDTH = 500;
7      public static final int HEIGHT = 400;

8      public static void main(String[] args)
9      {
10         GridLayoutJFrame gui = new GridLayoutJFrame(2, 3);
11         gui.setVisible(true);
12     }
```

(continued)

The **GridLayout** Manager (Part 2 of 4)

Display 17.9 **The GridLayout Manager**

```
13      public GridLayoutJFrame(int rows, int columns )
14      {
15          super();
16          setSize(WIDTH, HEIGHT);
17          setTitle("GridLayout Demonstration");
18          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19          setLayout(new GridLayout(rows, columns ));

20          JLabel label1 = new JLabel("First label");
21          add(label1);
```

(continued)

The **`GridLayout`** Manager (Part 3 of 4)

Display 17.9 The GridLayout Manager

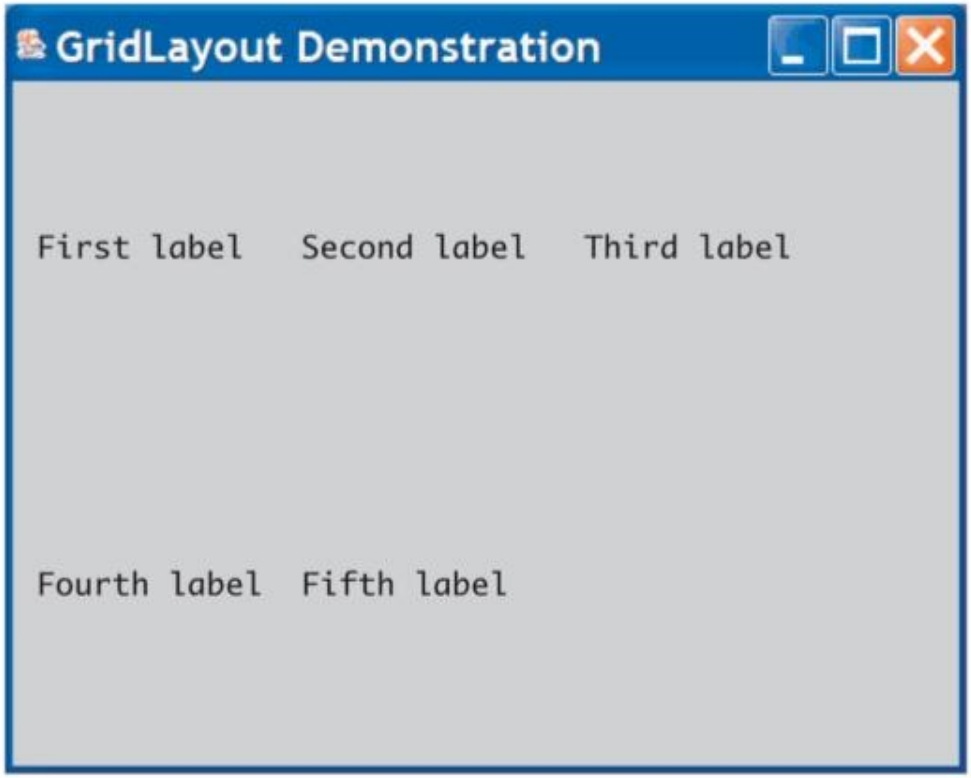
```
22            JLabel label2 = new JLabel("Second label");
23            add(label2);

24            JLabel label3 = new JLabel("Third label");
25            add(label3);

26            JLabel label4 = new JLabel("Fourth label");
27            add(label4);

28            JLabel label5 = new JLabel("Fifth label");
29            add(label5);
30        }
31    }
```

(continued)

The `GridLayout` Manager (Part 4 of 4)

Display 17.9 **The GridLayout Manager**

RESULTING GUI



Some Layout Managers

Display 17.10 Some Layout Managers

LAYOUT MANAGER	DESCRIPTION
These layout manager classes are in the `java.awt` package.	
`FlowLayout`	Displays components from left to right in the order in which they are added to the container.
`BorderLayout`	Displays the components in five areas: north, south, east, west, and center. You specify the area a component goes into in a second argument of the add method.
`GridLayout`	Lays out components in a grid, with each component stretched to fill its box in the grid.

What Can be Summarized?

Derived a new container class from, say, JFrame

In the derived class

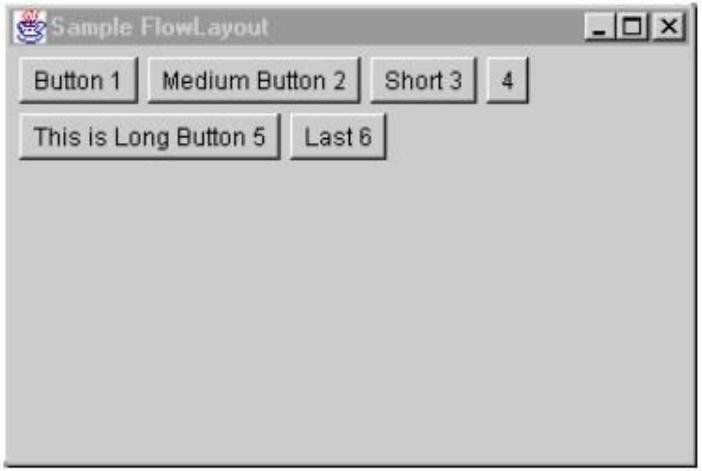
- Define a constructor that sets up the title and size of the window
- **Set up the proper layout of the widgets**
- Add the interface objects, such as buttons and others
- Remember to associate a listener object for each interface object

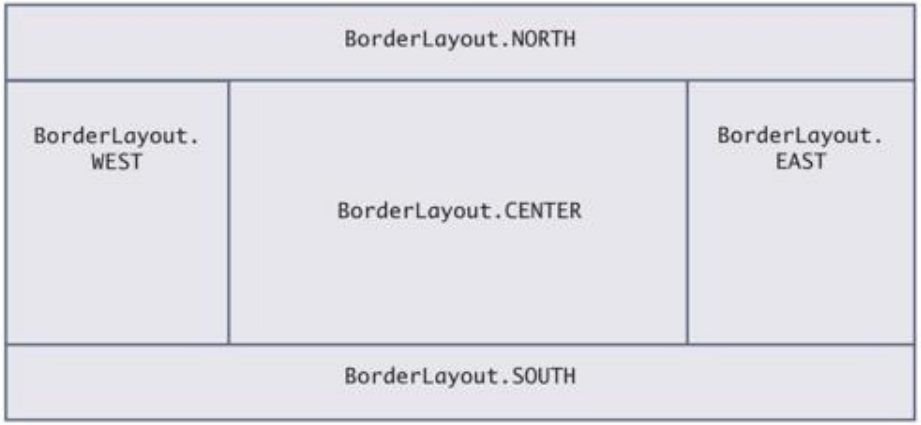
Define listener classes to handle possible events fired by the interface objects added in the window

In the main function

- Create the object of the derived window class
- Launch the interface by setting it as visible (you can do it in the constructor of the customized Frame as well)

Layout Summary





Action Listeners and Action Events

- Different kinds of components require different kinds of listener classes to **handle** the events they fire

- An action listener is an object whose class implements the **`ActionListener`** interface
  - The **`ActionListener`** interface has one method heading that must be implemented

```
public void actionPerformed(ActionEvent e)
{ //TODO
}
```

Action Listeners and Action Events

- To use the customized action listener class, you need to create a listener object first, e.g.,

```
EndingListener buttonEar = new EndingListener();
```

- Then, associate this listener with the specific component object, e.g.,

```
endButton.addActionListener(buttonEar);
```

The `setActionCommand` Method

- When a user clicks a button or menu item, an event is fired that normally goes to one or more action listeners
  - The action event becomes an argument to an `actionPerformed` method
  - *This action event includes a `String` instance variable called the action command for the button or menu item*
  - The default value for this string is the string written on the button or the menu item
  - This string can be retrieved with the `getActionCommand` method

```
e.getActionCommand()
```

The `setActionCommand` Method

- The `setActionCommand` method can be used to change the action command for a component
  - **This is especially useful when two or more buttons or menu items have the same default action command strings**

```
JButton nextButton = new JButton("Next");
nextButton.setActionCommand("Next Button");

JMenuItem choose = new JMenuItem("Next");
choose.setActionCommand("Next Menu Item");
```

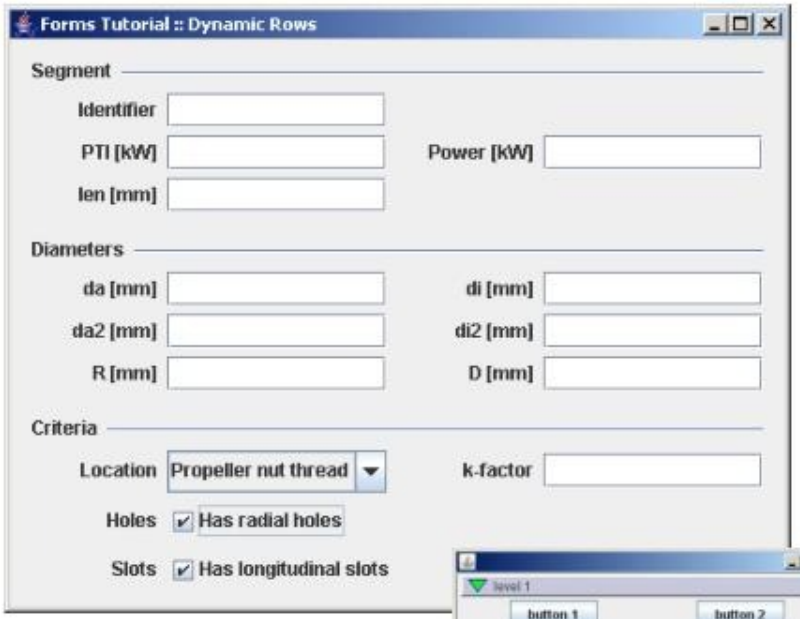
Listeners as Inner Classes

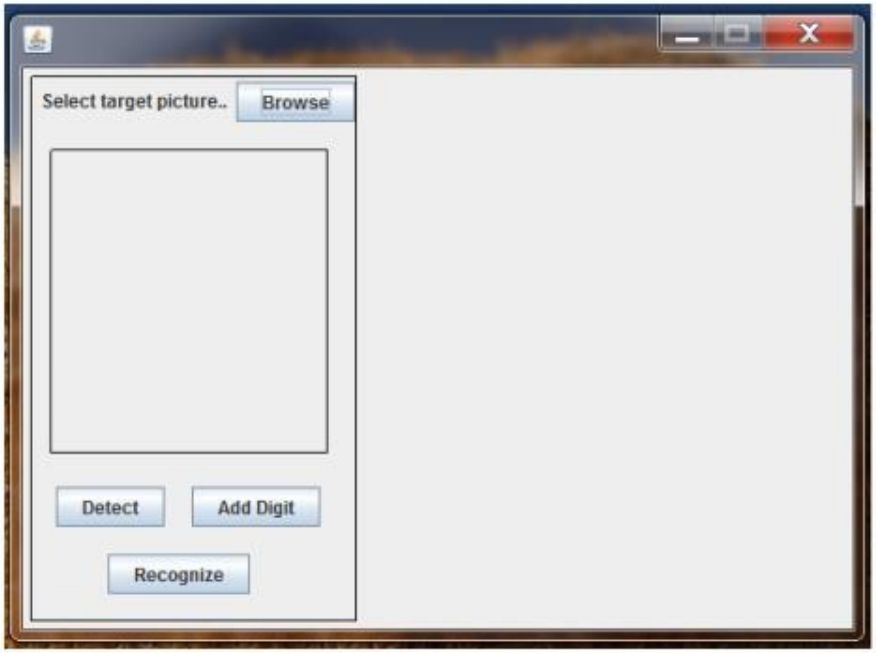
- Often, instead of having one action listener object deal with all the action events in a GUI, a separate **`ActionListener`** class is created for each button or menu item
  - Each button or menu item has its own unique action listener
  - There is then no need for a multiway if-else statement

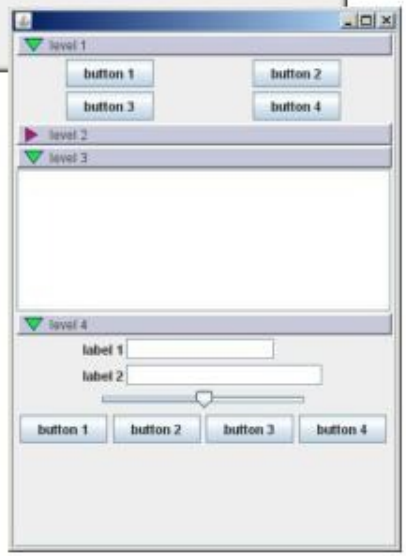
- **When this approach is used, each class is usually made a private inner class**

PANNEL

Panel Examples







Panels

- **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

```
setLayout(new BorderLayout());
JPanel somePanel = new JPanel();
somePanel.setLayout(new FlowLayout());
```

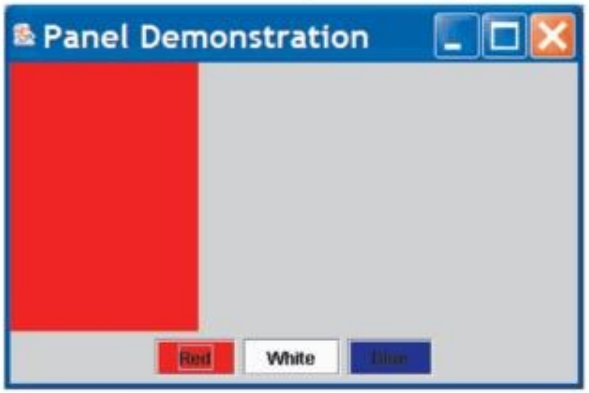
- Note in the following example that panel and button objects are given color using the **`setBackground`** method without invoking **`getContentPane`**
  - The **`getContentPane`** method is only used when adding color to a **`JFrame`**

Using Panels (Part 7 of 8)

Display 17.11 **Using Panels**

RESULTING GUI (When first run)

RESULTING GUI (After clicking Red button)

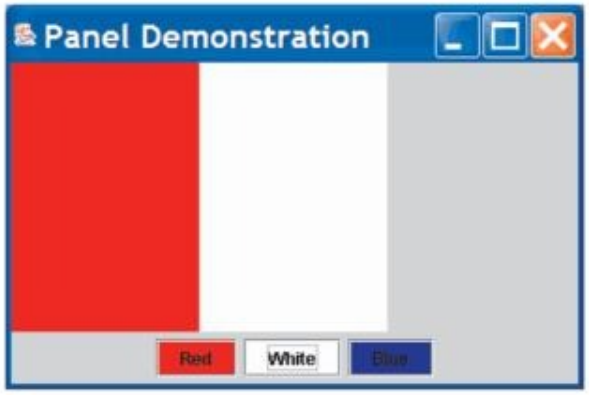


(continued)

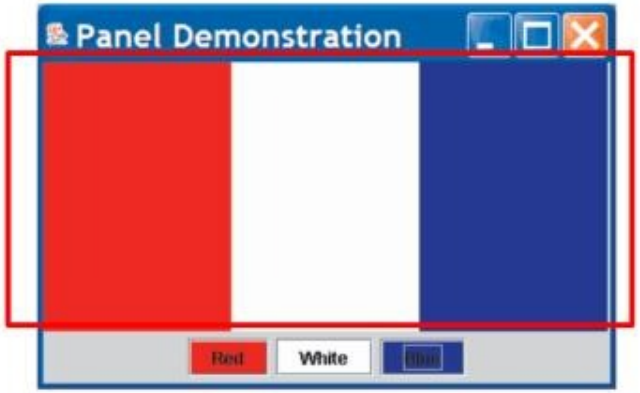
Using Panels (Part 8 of 8)

Display 17.11 **Using Panels**

RESULTING GUI (After clicking White button)



RESULTING GUI (After clicking Blue button)



Using Panels (Part 1 of 8)

Display 17.11 Using Panels

```
1   import javax.swing.JFrame;
2   import javax.swing.JPanel;
3   import java.awt.BorderLayout;
4   import java.awt.GridLayout;
5   import java.awt.FlowLayout;
6   import java.awt.Color;
7   import javax.swing.JButton;
8   import java.awt.event.ActionListener;
9   import java.awt.event.ActionEvent;

10  public class PanelDemo extends JFrame implements ActionListener
11  {
12      public static final int WIDTH = 300;
13      public static final int HEIGHT = 200;
```

In addition to being the GUI class, the class `PanelDemo` *is the action listener class. An object of the class* `PanelDemo` *is the action listener for the buttons in that object.*

(continued)

Using Panels (Part 2 of 8)

Display 17.11 Using Panels

```
14      private JPanel redPanel;
15      private JPanel whitePanel;
16      private JPanel bluePanel;

17      public static void main(String[] args)
18      {
19          PanelDemo gui = new PanelDemo();
20          gui.setVisible(true);
21      }

22      public PanelDemo()
23      {
24          super("Panel Demonstration");
25          setSize(WIDTH, HEIGHT);
26          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27          setLayout(new BorderLayout());
```

We made these instance variables because we want to refer to them in both the constructor and the method `actionPerformed`.

(continued)

Using Panels (Part 3 of 8)

Display 17.11 Using Panels

```
28          JPanel biggerPanel = new JPanel();
29          biggerPanel.setLayout(new GridLayout(1, 3));

30          redPanel = new JPanel();
31          redPanel.setBackground(Color.LIGHT_GRAY);
32          biggerPanel.add(redPanel);

33          whitePanel = new JPanel();
34          whitePanel.setBackground(Color.LIGHT_GRAY);
35          biggerPanel.add(whitePanel);
```

(continued)

Using Panels (Part 4 of 8)

Display 17.11 Using Panels

```
36          bluePanel = new JPanel();
37          bluePanel.setBackground(Color.LIGHT_GRAY);
38          biggerPanel.add(bluePanel);

39          add(biggerPanel, BorderLayout.CENTER);

40          JPanel buttonPanel = new JPanel();
41          buttonPanel.setBackground(Color.LIGHT_GRAY);
42          buttonPanel.setLayout(new FlowLayout());

43          JButton redButton = new JButton("Red");
44          redButton.setBackground(Color.RED);
45          redButton.addActionListener(this);
46          buttonPanel.add(redButton);
```

An object of the class `PanelDemo` *is the action listener for the buttons in that object.*

(continued)

Using Panels (Part 5 of 8)

Display 17.11 Using Panels

```
47          JButton whiteButton = new JButton("White");
48          whiteButton.setBackground(Color.WHITE);
49          whiteButton.addActionListener(this);
50          buttonPanel.add(whiteButton);

51          JButton blueButton = new JButton("Blue");
52          blueButton.setBackground(Color.BLUE);
53          blueButton.addActionListener(this);
54          buttonPanel.add(blueButton);

55          add(buttonPanel, BorderLayout.SOUTH);
56      }
```

(continued)

Using Panels (Part 6 of 8)

Display 17.11 Using Panels

```
57      public void actionPerformed(ActionEvent e)
58      {
59          String buttonString = e.getActionCommand();

60          if (buttonString.equals("Red"))
61              redPanel.setBackground(Color.RED);
62          else if (buttonString.equals("White"))
63              whitePanel.setBackground(Color.WHITE);
64          else if (buttonString.equals("Blue"))
65              bluePanel.setBackground(Color.BLUE);
66          else
67              System.out.println("Unexpected error.");
68      }
69  }
```

(continued)

The panels with borderlines example

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

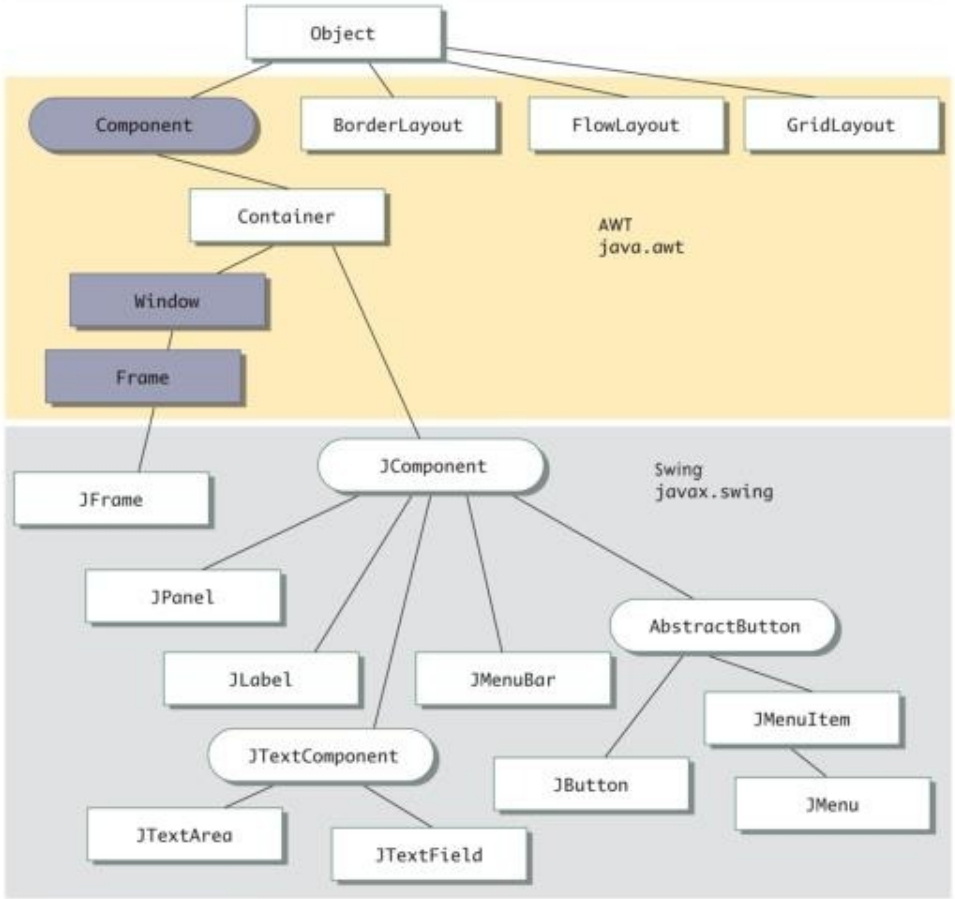
- Any descendent class of the class `JComponent` is called a *component class*
  - Any `JComponent` object or *component* can be added to any container class object
  - **Because it is derived from the class `Container`, a `JComponent` can also be added to another `JComponent`**

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

Display 17.12 **Hierarchy of Swing and AWT Classes**



Abstract Class

Concrete Class

A line between two boxes means the lower class is derived from (`extends`) the higher one.

This blue color indicates a class that is not used in this text but is included here for reference. If you have not heard of any of these classes, you can safely ignore them. (The class `Component` does receive very brief treatment in Chapter 19.)

Tip: **Code a GUI's Look and Actions Separately**

- The task of designing a Swing GUI can be divided into two main subtasks:
  1. Designing and coding the appearance of the GUI on the screen
  2. Designing and coding the actions performed in response to user actions

- In particular, it is useful to implement the **`actionPerformed`** method as a *stub*, until the GUI looks the way it should

```
public void actionPerformed(ActionEvent e)
{}
```

What Can be Summarized?

Derived a new container class from, say, JFrame

In the derived class

- Define a constructor that sets up the title and size of the window
- **Set up the proper lay out of the outer container**
- **Create inner containers**
- **Set up the proper lay out of each inner containers**
- Add the interface objects, such as buttons and others, to the corresponding containers
- Remember to associate a listener object for each interface object
- **Add the containers to the Frame object in order**

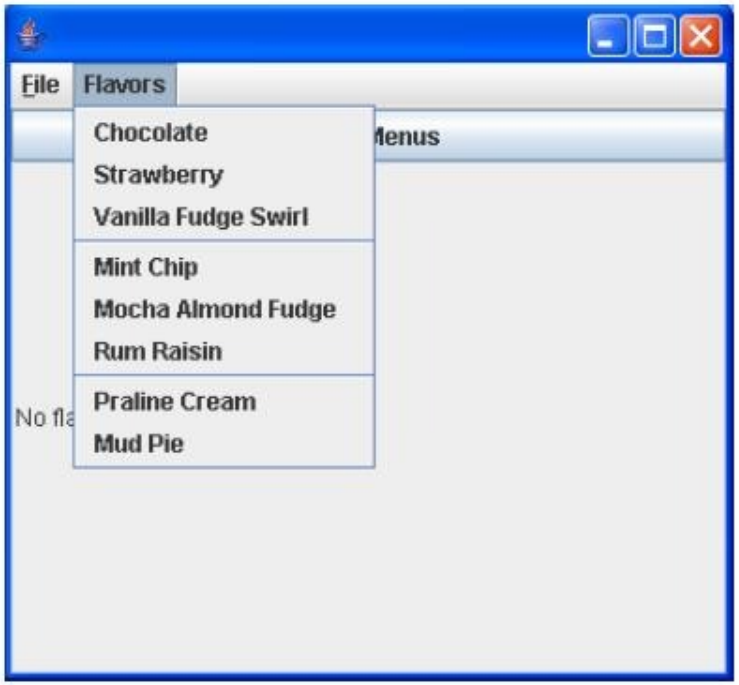
Define listener classes to handle possible events fired by the interface objects added in the window

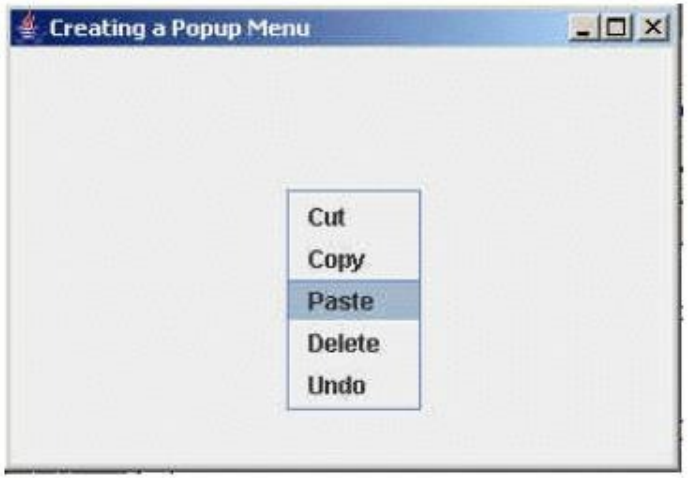
In the main function

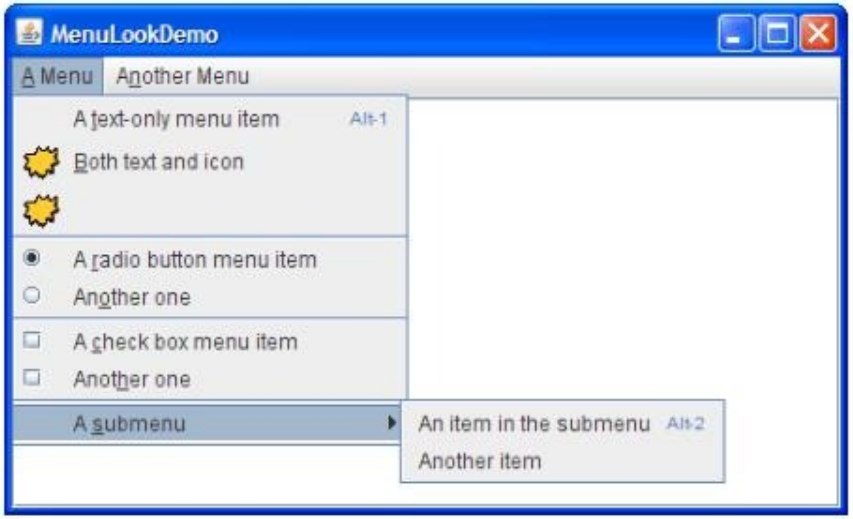
- Create the object of the derived window class
- Launch the interface by setting it as visible

MENU

Menu Examples







Menu Bars, Menus, and Menu Items

- A *menu* is an object of the class **`JMenu`**
- A choice on a menu is called a ***menu item***, and is an object of the class **`JMenuItem`**
  - A menu can contain any number of menu items
  - **A menu item is identified by the string that labels it**, and is displayed in the order to which it was added to the menu
- The **`add`** method is used to add a menu item to a menu in the same way that a component is added to a container object

Menu Bars, Menus, and Menu Items

- The following creates a new menu, and then adds a menu item to it

```
JMenu diner = new
        JMenu("Daily Specials");
JMenuItem lunch = new
        JMenuItem("Lunch Specials");
lunch.addActionListener(this);
diner.add(lunch);
```

  - Note that the `this` parameter has been registered as an action listener for the menu item

Nested Menus

- The class **`JMenu`** is a descendent of the **`JMenuItem`** class
  - **Every `JMenu` can be a menu item in another menu**
  - Therefore, menus can be nested
- Menus can be added to other menus in the same way as menu items

Menu Bars and `JFrame`

- **A *menu bar* is a container for menus, typically placed near the top of a windowing interface**

- The `add` method is used to add a menu to a menu bar in the same way that menu items are added to a menu

```
JMenuBar bar = new JMenuBar();
bar.add(diner);
```

- The menu bar can be added to a `JFrame` in two different ways
  1. Using the `setJMenuBar` method

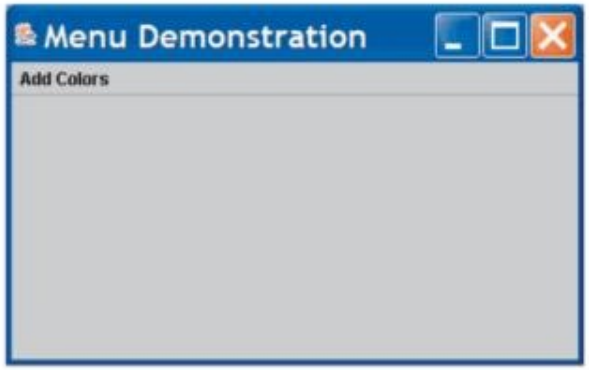
```
setJMenuBar(bar);
```

  2. Using the `add` method – which can be used to add a menu bar to a `JFrame` or any other container

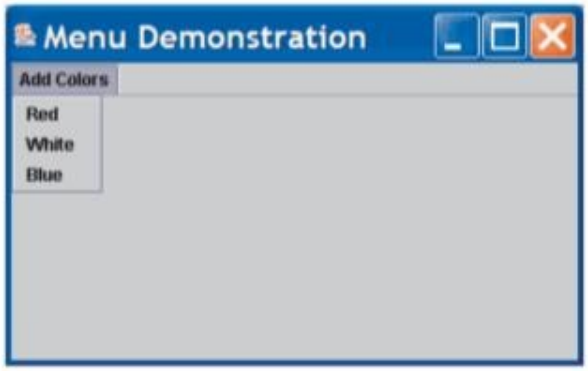
A GUI with a Menu (Part 7 of 8)

Display 17.14 **A GUI with a Menu**

RESULTING GUI



RESULTING GUI (after clicking Add Colors in the menu bar)

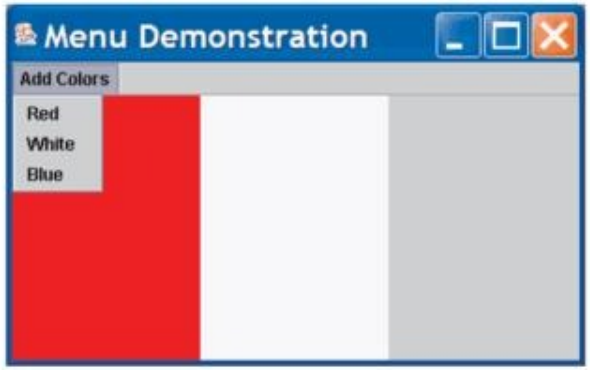


(continued)

A GUI with a Menu (Part 8 of 8)

Display 17.14 **A GUI with a Menu**

RESULTING GUI (after choosing Red and White on the menu)



RESULTING GUI (after choosing all the colors on the menu)

A GUI with a Menu (Part 1 of 8)

Display 17.14 A GUI with a Menu

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import java.awt.GridLayout;
4  import java.awt.Color;
5  import javax.swing.JMenu;
6  import javax.swing.JMenuItem;
7  import javax.swing.JMenuBar;
8  import java.awt.event.ActionListener;
9  import java.awt.event.ActionEvent;
```

(continued)

A GUI with a Menu (Part 2 of 8)

Display 17.14 A GUI with a Menu

```
10  public class MenuDemo extends JFrame implements ActionListener
11  {
12      public static final int WIDTH = 300;
13      public static final int HEIGHT = 200;

14      private JPanel redPanel;
15      private JPanel whitePanel;
16      private JPanel bluePanel;

17      public static void main(String[] args)
18      {
19          MenuDemo gui = new MenuDemo();
20          gui.setVisible(true);
21      }
```

(continued)

A GUI with a Menu (Part 3 of 8)

Display 17.14 A GUI with a Menu

```
22      public MenuDemo()
23      {
24          super("Menu Demonstration");
25          setSize(WIDTH, HEIGHT);
26          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27          setLayout(new GridLayout(1, 3));

28          redPanel = new JPanel();
29          redPanel.setBackground(Color.LIGHT_GRAY);
30          add(redPanel);

31          whitePanel = new JPanel();
32          whitePanel.setBackground(Color.LIGHT_GRAY);
33          add(whitePanel);
```

(continued)

A GUI with a Menu (Part 4 of 8)

Display 17.14 **A GUI with a Menu**

```
34          bluePanel = new JPanel();
35          bluePanel.setBackground(Color.LIGHT_GRAY);
36          add(bluePanel);

37          JMenu colorMenu = new JMenu("Add Colors");

38          JMenuItem redChoice = new JMenuItem("Red");
39          redChoice.addActionListener(this);
40          colorMenu.add(redChoice);

41          JMenuItem whiteChoice = new JMenuItem("White");
42          whiteChoice.addActionListener(this);
43          colorMenu.add(whiteChoice);
```

(continued)

A GUI with a Menu (Part 5 of 8)

Display 17.14 A GUI with a Menu

```
44          JMenuItem blueChoice = new JMenuItem("Blue");
45          blueChoice.addActionListener(this);
46          colorMenu.add(blueChoice);

47          JMenuBar bar = new JMenuBar();
48          bar.add(colorMenu);
49          setJMenuBar(bar);
50      }
```

The definition of `actionPerformed` *is identical to the definition given in Display 17.11 for a similar GUI using buttons instead of menu items.*

(continued)

A GUI with a Menu (Part 6 of 8)

Display 17.14 A GUI with a Menu

```
51      public void actionPerformed(ActionEvent e)
52      {
53          String buttonString = e.getActionCommand();

54          if (buttonString.equals("Red"))
55               redPanel.setBackground(Color.RED);
56          else if (buttonString.equals("White"))
57              whitePanel.setBackground(Color.WHITE);
58          else if (buttonString.equals("Blue"))
59              bluePanel.setBackground(Color.BLUE);
60          else
61              System.out.println("Unexpected error.");
62      }
63  }
```

(continued)