GUI with Swing – Part I

CIS*2430 (Fall 2021)

Introduction to GUIs

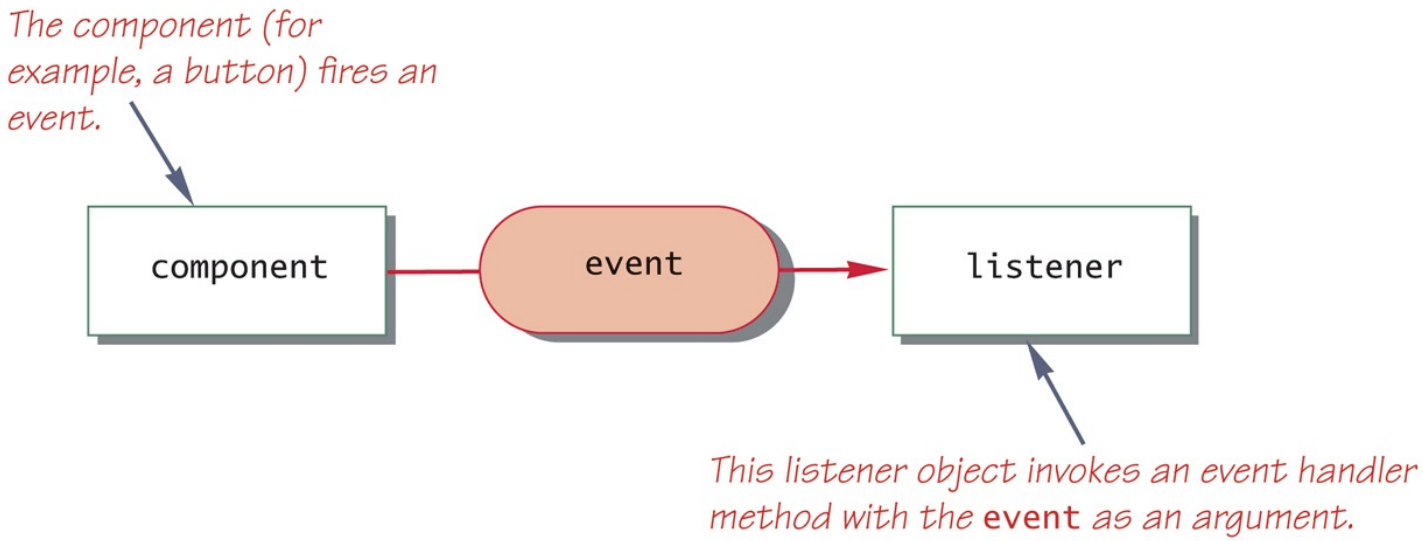
- AWT (Abstract Window Toolkit) is the original Java package for doing GUI (Graphical User Interface).

- 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 doing GUIs.

- GUIs are designed using a form of programming known as event-driven programming.

Event-Driven Programming

Display 17.1 Event Firing and an Event Listener



Event-Driven Programming

- In event-driven programming, objects are created that can fire events, and listener objects are created that react to the events.

- The program can no longer determines what happens next: the events from a user determine the order of actions.

- Some methods defined may never be explicitly invoked by some users.

First Swing Demo (1/4)

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)

First Swing Demo (2/4)

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)

First Swing Demo (3/4)

Display 17.2 A First Swing Demonstration Program

This is the file `EndingListener.java`.

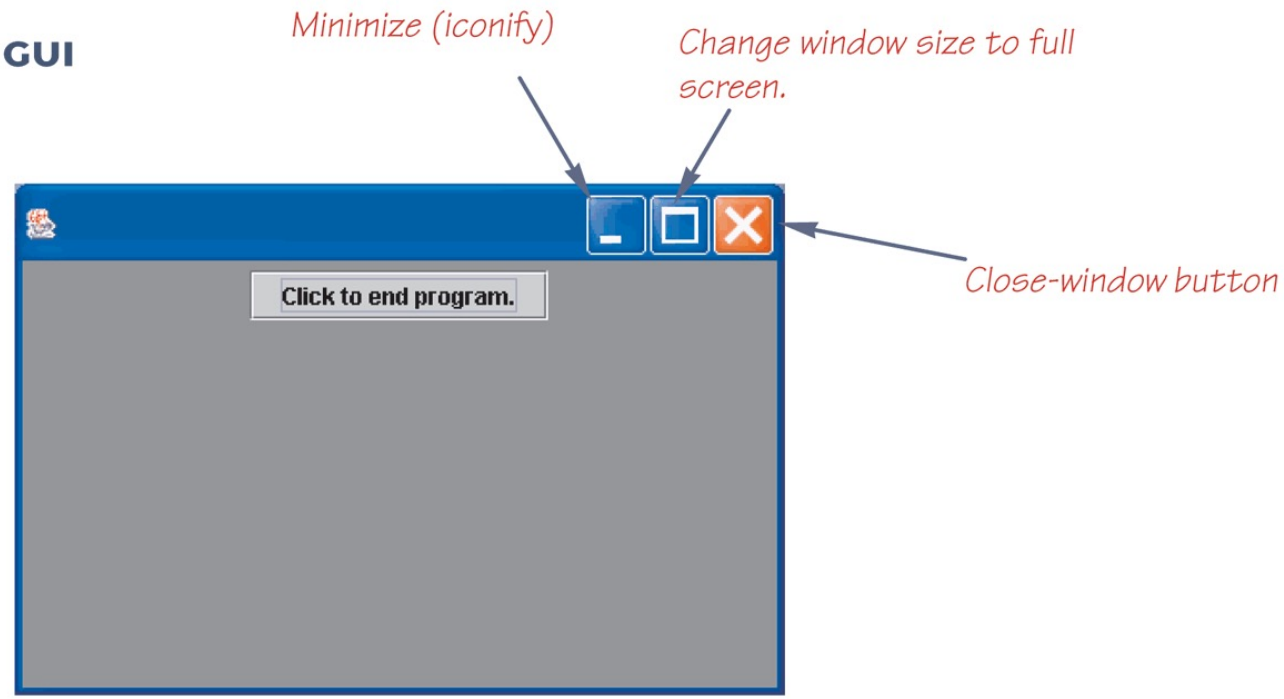
```
1  import java.awt.event.ActionListener;
2  import java.awt.event.ActionEvent;

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

(continued)

First Swing Demo (4/4)

Display 17.2 A First Swing Demonstration Program

RESULTING GUI



Some Methods for JFrame (1/3)

Display 17.3 Some Methods in the Class `JFrame`

The class `JFrame` is in the `javax.swing` package.

```
public JFrame()
```

Constructor that creates an object of the class `JFrame`.

```
public JFrame(String title)
```

Constructor that creates an object of the class `JFrame` with the title given as the argument.

(continued)

Some Methods for JFrame (2/3)

Display 17.3 Some Methods in the Class `JFrame`

```
public void setDefaultCloseOperation(int operation)
```

Sets the action that will happen by default when the user clicks the close-window button. The argument should be one of the following defined constants:

`JFrame.DO_NOTHING_ON_CLOSE`: Do nothing. The `JFrame` does nothing, but if there are any registered window listeners, they are invoked. (Window listeners are explained in Chapter 19.)

`JFrame.HIDE_ON_CLOSE`: Hide the frame after invoking any registered `WindowListener` objects.

`JFrame.DISPOSE_ON_CLOSE`: Hide and *dispose* the frame after invoking any registered window listeners. When a window is **disposed** it is eliminated but the program does not end. To end the program, you use the next constant as an argument to `setDefaultCloseOperation`.

`JFrame.EXIT_ON_CLOSE`: Exit the application using the `System exit` method. (Do not use this for frames in applets. Applets are discussed in Chapter 18.)

If no action is specified using the method `setDefaultCloseOperation`, then the default action taken is `JFrame.HIDE_ON_CLOSE`.

Throws an `IllegalArgumentException` if the argument is not one of the values listed above.[2]

Throws a `SecurityException` if the argument is `JFrame.EXIT_ON_CLOSE` and the Security Manager will not allow the caller to invoke `System.exit`. (You are not likely to encounter this case.)

```
public void setSize(int width, int height)
```

Sets the size of the calling frame so that it has the `width` and `height` specified. Pixels are the units of length used.

(continued)

Some Methods for JFrame (3/3)

Display 17.3 Some Methods in the Class `JFrame`

```
public void setTitle(String title)
```

Sets the title for this frame to the argument string.

```
public void add(Component componentAdded)
```

Adds a component to the `JFrame`.

```
public void setLayout(LayoutManager manager)
```

Sets the layout manager. Layout managers are discussed later in this chapter.

```
public void setJMenuBar(JMenuBar menubar)
```

Sets the menubar for the calling frame. (Menus and menu bars are discussed later in this chapter.)

```
public void dispose()
```

Eliminates the calling frame and all its subcomponents. Any memory they use is released for reuse. If there are items left (items other than the calling frame and its subcomponents), then this does not end the program. (The method `dispose` is discussed in Chapter 19.)

Pixels and Resolutions

- A *pixel* is the smallest unit of space on a screen:
  - Both the size and position of Swing objects are measured in pixels.
  - A high-resolution screen of fixed size has many pixels.
  - A low-resolution screen of fixed size has fewer pixels.

- Therefore, a two-pixel figure on a low-resolution screen will look larger than a two-pixel figure on a high-resolution screen.

Events and Listeners

- A *button* object is created from the class **JButton** and can be added to a **Jframe**.
- Clicking a button fires an event, which is sent to a listener, which automatically invokes an event handler using the event as the argument.
- In order to set up this relationship, a GUI program must do two things:
  - It must specify, for each button, what objects are its listeners, i.e., it must register the listeners.
  - It must define the methods that will be invoked automatically when the event is sent to the listener.

Events and Listeners

- 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:

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

  - Note that **e** must be received, even if it is not used.

Pitfall 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.

Tip: Ending a Swing Program

- GUI programs are often based on a kind of infinite loop.
- If the user never asks the windowing system to go away, it will never go away.
- In order to end a GUI program, **System.exit** must be used when the user asks to end the program:
  - It must be explicitly invoked or included in some library code that is executed.
  - Otherwise, a Swing program will not end after it has executed all the code in the program.

Normal Way for JFrame (1/4)

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)

Normal Way for JFrame (2/4)

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)

Normal Way for JFrame (3/4)

Display 17.4 The Normal Way to Define a JFrame

This is the file DemoWindow.java.

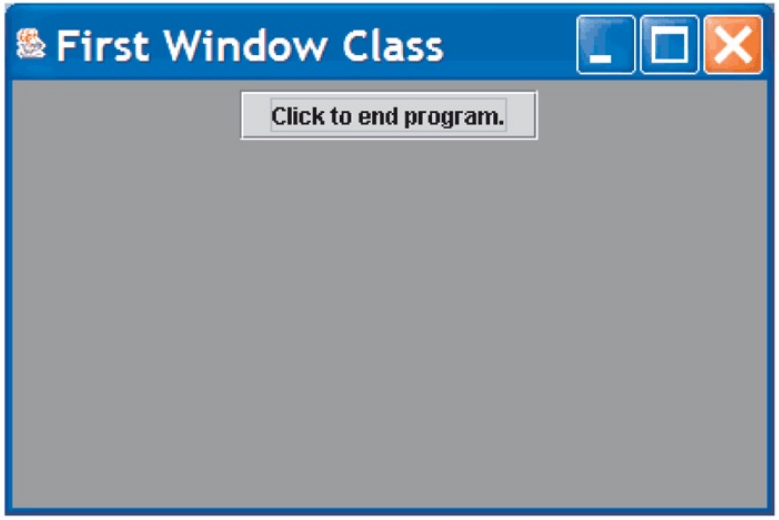
```
1  public class DemoWindow
2  {
3      public static void main(String[] args)
4      {
5          FirstWindow w = new FirstWindow();
6          w.setVisible(true);
7      }
8  }
```

(continued)

Normal Way for JFrame (4/4)

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

RESULTING GUI



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:**
  - There are constants in the **Color** class that represent many basic colors.

- A **JFrame** cannot be colored directly:
  - Instead, a program must color something called the content pane of the **JFrame.**
  - The background color of a **JFrame** can be set using the following code:

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

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.

JFrame with Color (1/4)

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)

JFrame with Color (2/4)

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)

JFrame with Color (3/4)

Display 17.6 A JFrame with Color

This is the file `ColoredWindow.java`.

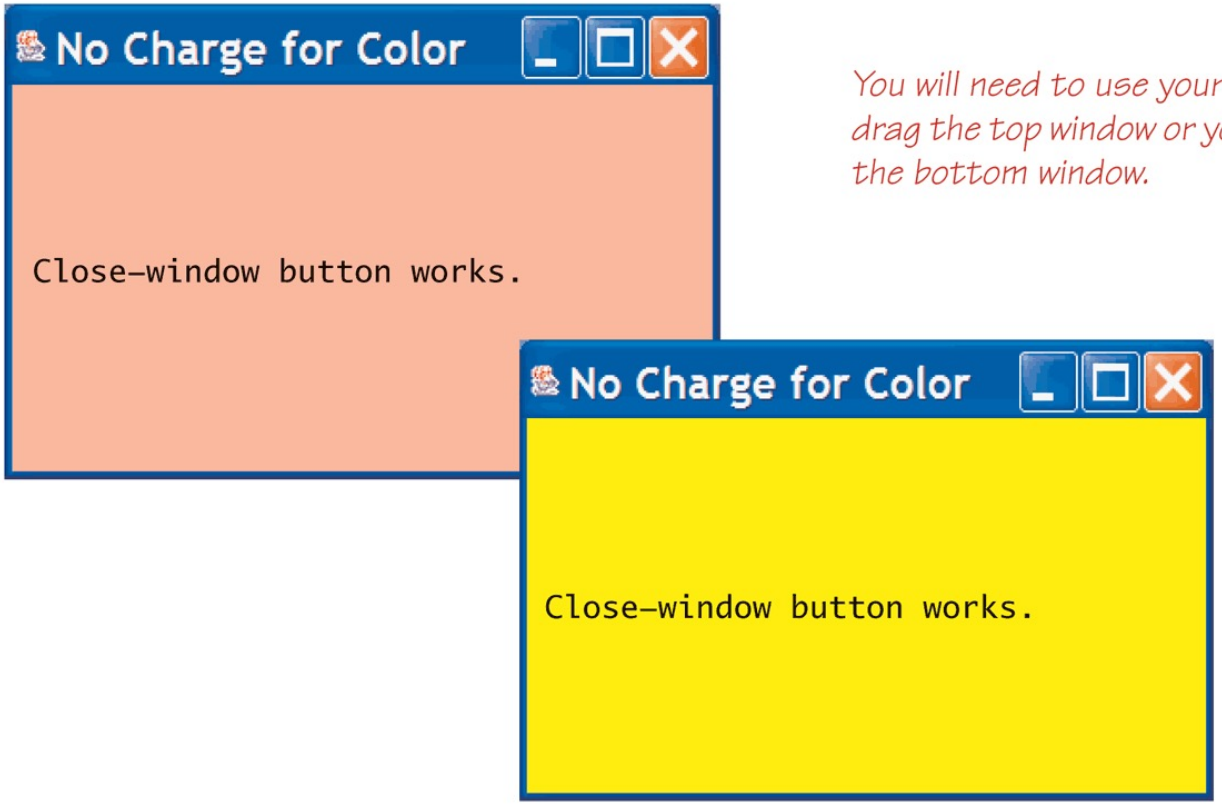
```
1  import java.awt.Color;

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)

JFrame with Color (4/4)

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.

Containers and Layout Managers

- Multiple components can be added to the content pane of a **JFrame,** but the **add** method doesn't specify how these components are to be arranged.

- To describe how multiple components are to be arranged, a *layout manager* 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.

BorderLayout Manager

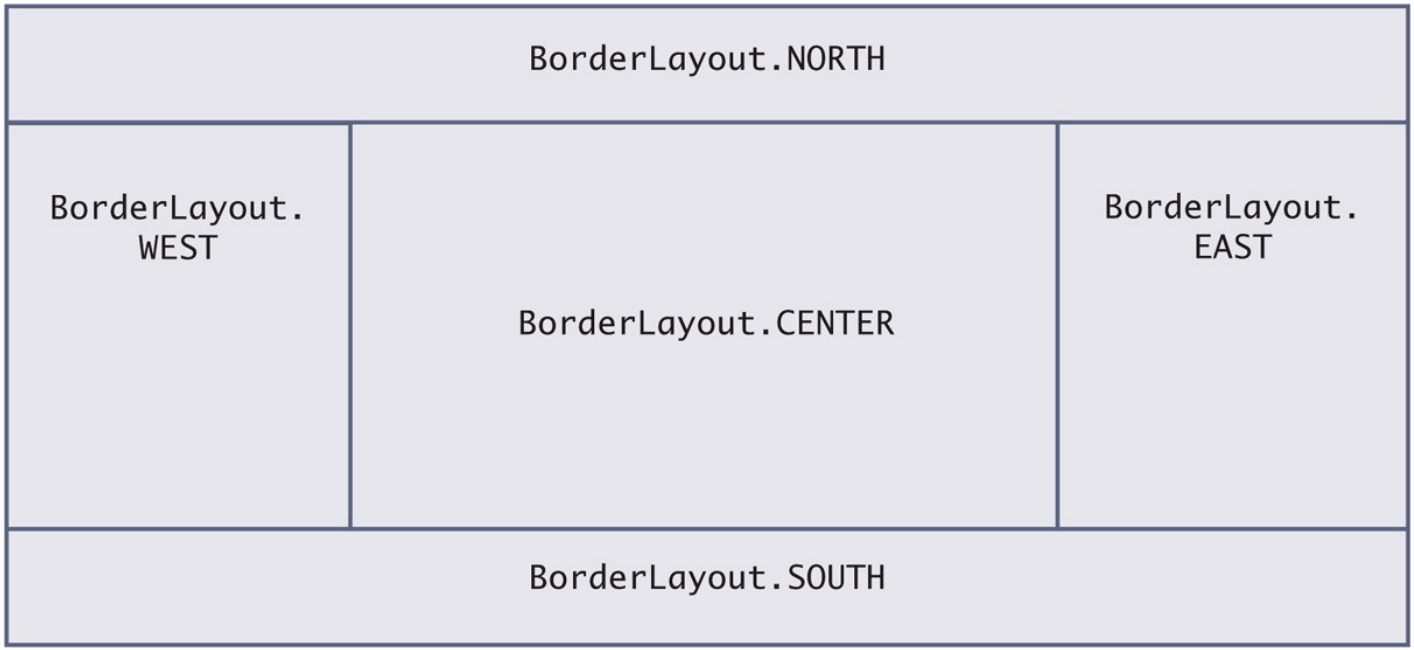
- A **BorderLayout** manager places the components to a **JFrame** object partitioned into five regions: **BorderLayout.NORTH**, **BorderLayout.SOUTH**, **BorderLayout.EAST**, **BorderLayout.WEST**, and **BorderLayout.Center**.

- A **BorderLayout** manager is added to a **JFrame** using the **setLayout** method

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

BorderLayout Regions

Display 17.8 **BorderLayout Regions**



– Note that none of the lines in the diagram are normally visible

BorderLayout Manager (1/4)

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)

BorderLayout Manager (2/4)

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)

BorderLayout Manager (3/4)

Display 17.7 The BorderLayout Manager

This is the file BorderLayoutDemo.java.

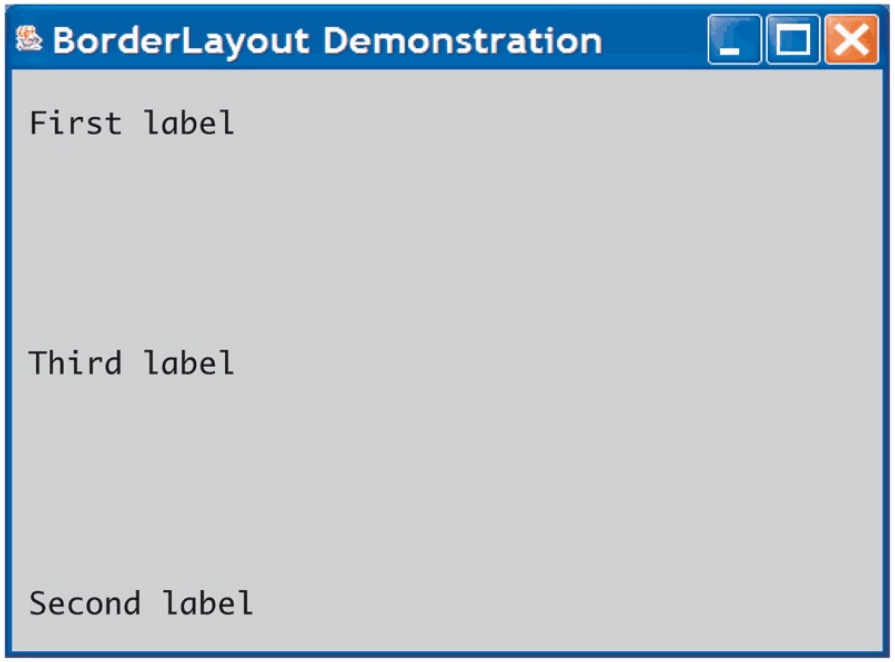
```
1  public class BorderLayoutDemo
2  {
3      public static void main(String[] args)
4      {
5          BorderLayoutJFrame gui = new BorderLayoutJFrame();
6          gui.setVisible(true);
7      }
8  }
```

(continued)

BorderLayout Manager (4/4)

Display 17.7 **The BorderLayout Manager**

RESULTING GUI



`FlowLayout` Manager

- 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 only takes one argument.

`GirdLayout` Manager

- A **GridLayout** manager arranges components in a two-dimensional grid:

  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.

GirdLayout Manager

- 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.

GirdLayout Manager (1/4)

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)

GirdLayout Manager (2/4)

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)

GirdLayout Manager (3/4)

Display 17.9 The GridLayout Manager

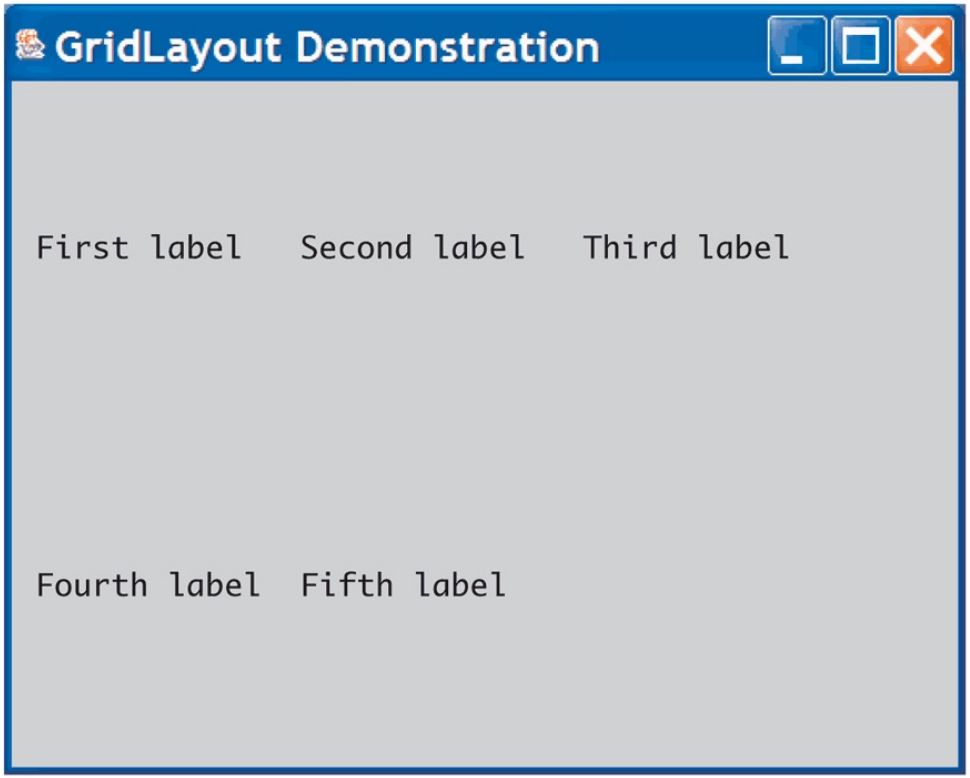
```
22          JLabel label2 = new JLabel("Second label");
23          add(label2);

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)

GridLayout Manager (4/4)

Display 17.9 **The GridLayout Manager**

RESULTING GUI



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 containers.

Panels

- Both a **JFrame** and any panels in it can use different layout managers:

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

- We can color a panel using the **setBackground** method without invoking **getContentPane:**
  - The **getContentPane** method is only used when adding color to a **Jframe.**

Using Panels (1/8)

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 (2/8)

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 (3/8)

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 (4/8)

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 (5/8)

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 (6/8)

Display 17.11 Using Panels

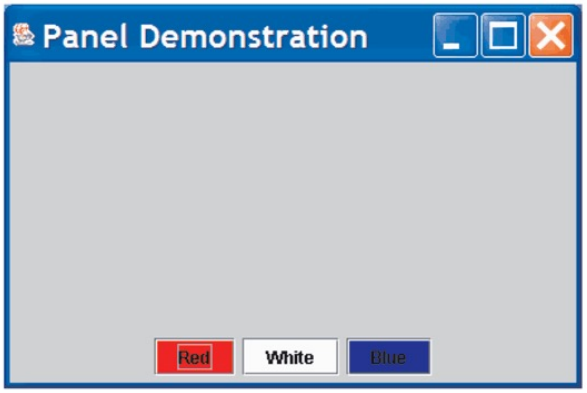
```
57      public void actionPerformed(ActionEvent e)
58      {
59          String buttonString = e.getActionCommand();

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)

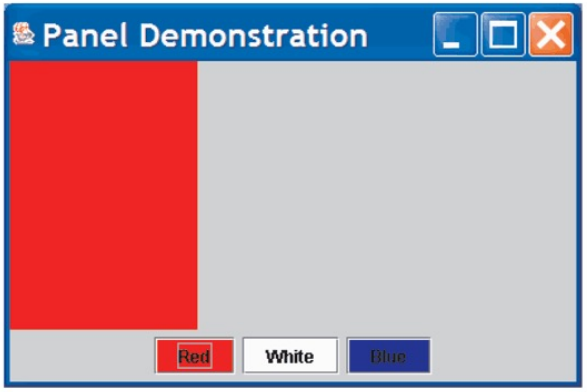
Using Panels (7/8)

Display 17.11 **Using Panels**

RESULTING GUI (When first run)



RESULTING GUI (After clicking Red button)

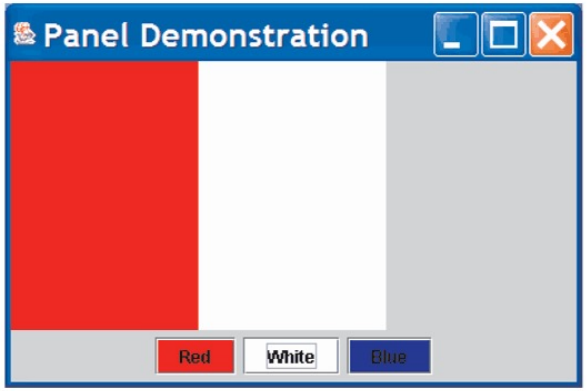


(continued)

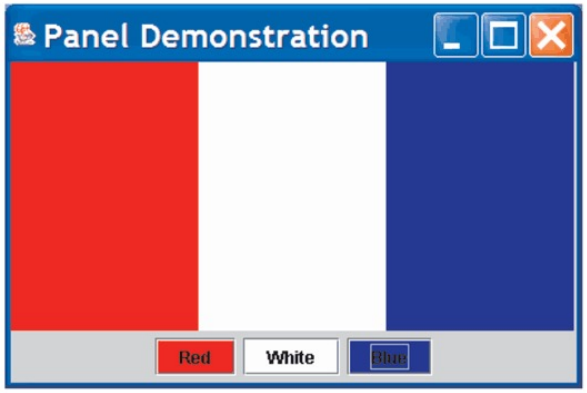
Using Panels (8/8)

Display 17.11 **Using Panels**

RESULTING GUI (After clicking White button)



RESULTING GUI (After clicking Blue button)



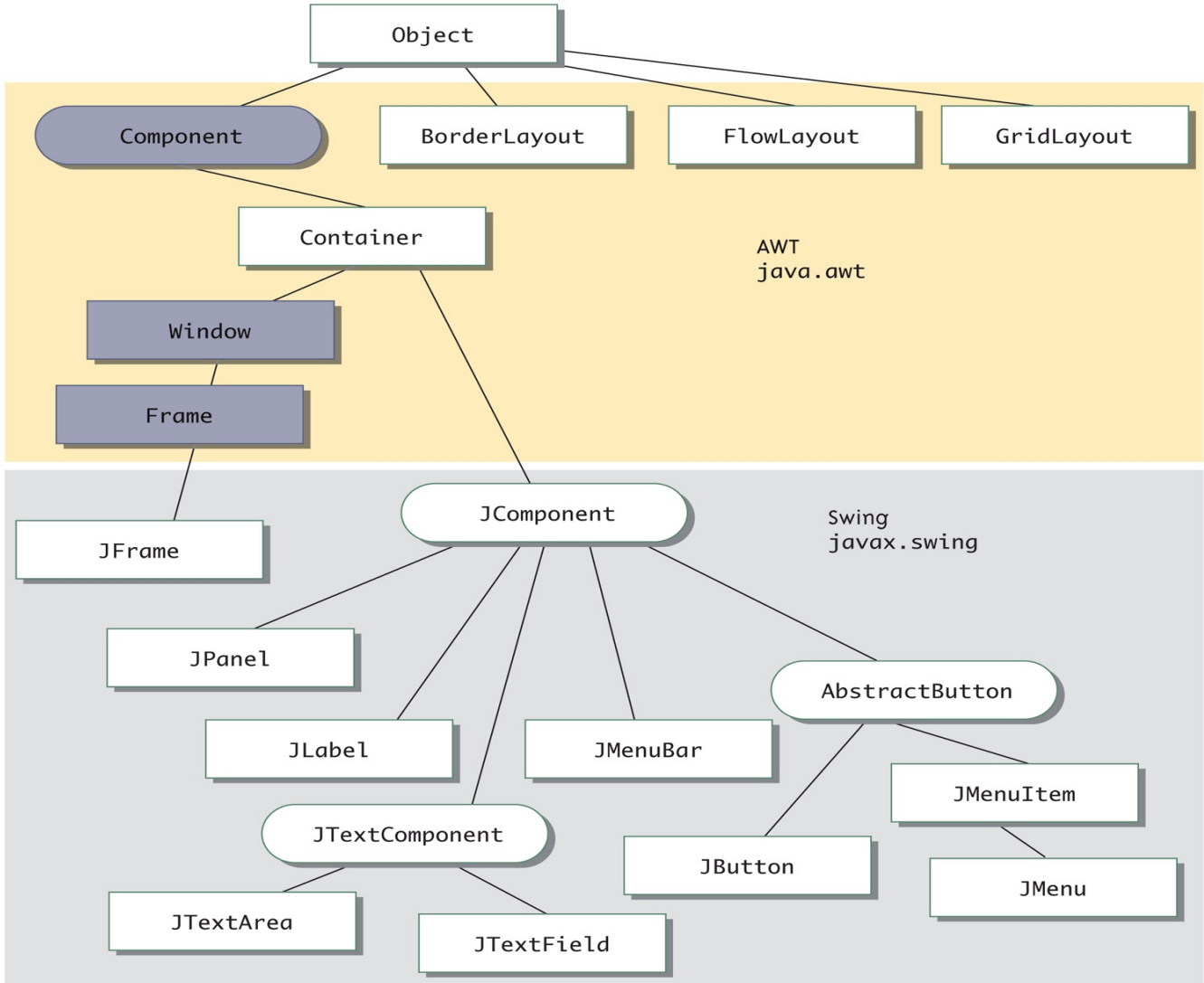
The `Container` Class

- Any class that is a descendent of the class Container is considered as 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 containers.

The JComponet 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.**

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.)

Code Look and Actions Separately

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

- It's useful to treat **actionPerformed** method as a *stub*, until the GUI looks good enough:

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

Text Fields

- A *text field* is an object of the class **JTextField:**
  - It is displayed as a field that allows the user to enter a single line of text:

```
private JTextField name;
. . .
name = new JTextField(NUMBER_OF_CHAR);
```

  - In the text field above, at least **NUMBER_OF_CHAR** characters can be visible.

Text Fields

- There is also a constructor with one additional **String** parameter for displaying an initial **String** in the text field:

```
JTextField name = new JTextField("Enter name here.", 30);
```

- A Swing GUI can read the text in a text field using the **getText** method:

```
String inputString = name.getText();
```

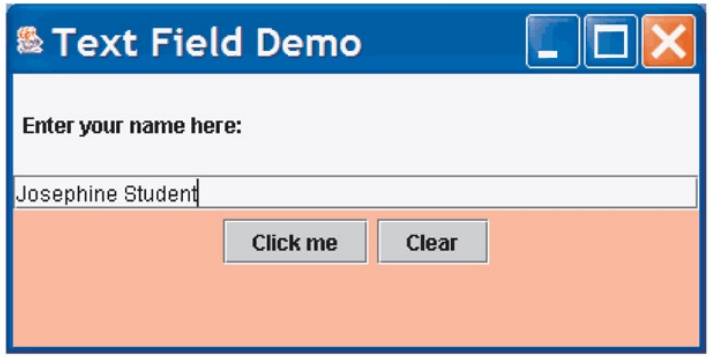
- The method **setText** can be used to display a new text string in a text field:

```
name.setText("This is some output");
```

Text Field Example (1/2)

Display 17.17 **A Text Field**

RESULTING GUI (When program is started and a name entered)

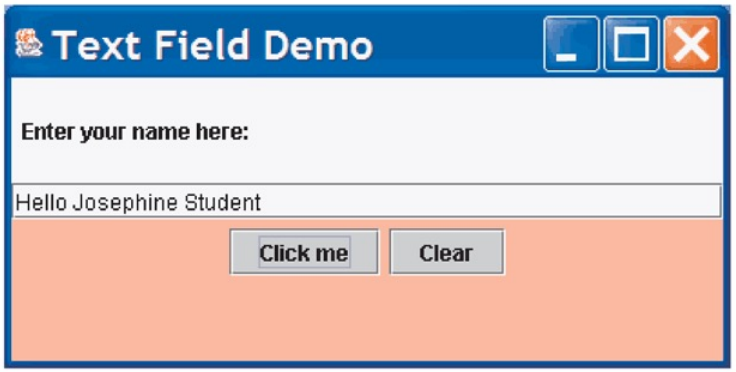


(continued)

Text Field Example (2/2)

Display 17.17 **A Text Field**

RESULTING GUI (After clicking the "Click me" button)



Text Areas

- A *text area* is an object of the class **JTextArea:**
  - It is the same as a text field, except that it allows multiple lines
  - Two parameters to the **JTextArea** constructor specify the minimum number of lines, and the minimum number of characters per line that are guaranteed to be visible

```
JTextArea theText = new JTextArea(5,20);
```

  - Another constructor has one addition **String** parameter for the string initially displayed in the text area

```
JTextArea theText = new JTextArea(
    "Enter\ntext here." 5, 20);
```

Text Areas

- The line-wrapping policy for a **JTextArea** can be set using the method **setLineWrap:**
  - The method takes one **boolean** type argument.
  - If the argument is **true**, then any additional characters at the end of a line will appear on the following line of the text area.
  - If the argument is **false**, the extra characters will remain on the same line and not be visible:

```
theText.setLineWrap(true);
```

Text Fields and Text Areas

- A **JTextField** or **JTextArea** can be set so that it can not be changed by the user:

```
theText.setEditable(false);
```

  - This will set **theText** so that it can only be edited by the GUI program, not the user
  - To reverse this, use true instead (this is the default)

```
theText.setEditable(true);
```

Tip: Label a Text Field

- In order to label one or more text fields:
  - Use an object of the class **Jlabel.**
  - Place the text field(s) and label(s) in a **JPanel.**
  - Treat the **JPanel** as a single component.

Tip: Input/Output Numbers

- When attempting to input numbers from any Swing GUI, input text must be converted to numbers:
  - If the user enters the number **42** in a **JTextField**, the program receives the string **"42"** and must convert it to the integer **42.**

- The same thing is true when attempting to output a number:
  - In order to output the number **42**, it must first be converted to the string **"42".**

Combo Boxes

- A combo box lets the user to choose one of the several choices:

```
String[] petStrings = { "Bird", "Cat", "Dog", "Rabbit", "Pig" };
JComboBox petList = new JComboBox(petStrings);
petList.setSelectedIndex(4);
petList.addActionListener(this);
```

Using Combo Boxes (1/3)

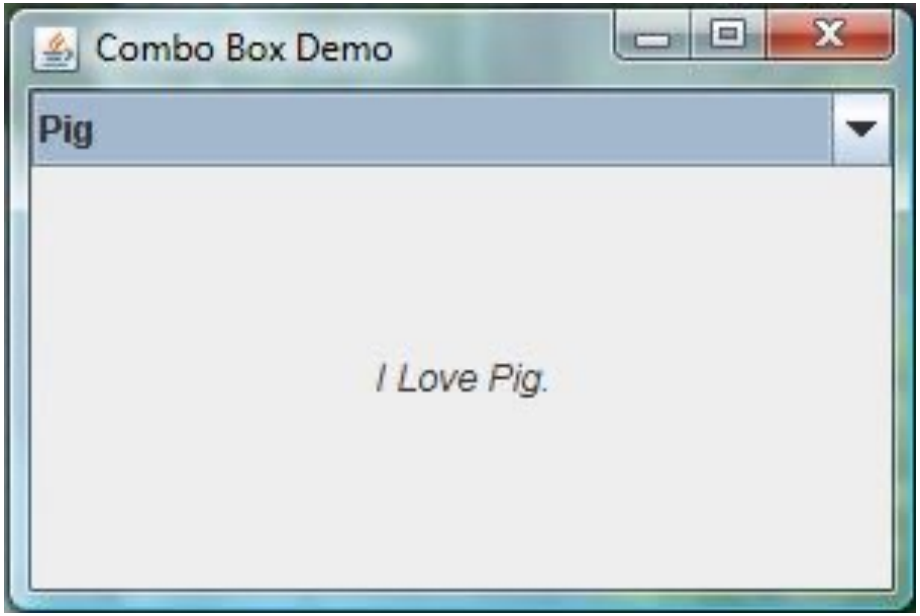
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ComboBoxDemo extends Jframe implements
    ActionListener {

    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;

    JLabel message;

    public static void main(String[] args) {
        ComboBoxDemo gui = new ComboBoxDemo();
        gui.setVisible(true);
    }
```

Using Combo Boxes (2/3)

```
public ComboBoxDemo() {
    super("Combo Box Demo");
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new BorderLayout());

    String[] petStrings = { "Bird", "Cat", "Dog", "Rabbit", "Pig" };
    JComboBox petList = new JComboBox(petStrings);
    petList.setSelectedIndex(4);
    petList.addActionListener(this);

    message = new JLabel("I Love " + petStrings[4] + ".");
    message.setFont(message.getFont().deriveFont(Font.ITALIC));
    message.setHorizontalAlignment(JLabel.CENTER);

    add(petList, BorderLayout.NORTH);
    add(message, BorderLayout.CENTER);
}
```

Using Combo Boxes (3/3)

```
    public void actionPerformed(ActionEvent e) {
        JComboBox cb = (JComboBox)e.getSource();
        String petName = (String)cb.getSelectedItem();
        message.setText("I Love " + petName + ".");
    }
}
```



Scroll Bars

- When a text area is created, the number of lines that are visible and the number of characters per line are specified as follows:

```
JTextArea memoDisplay = new JTextArea(15, 30);
```

- However, it would often be better not to have to set a firm limit on the number of lines or the number of characters per line:
  - This can be done by using *scroll bars* with the text area.

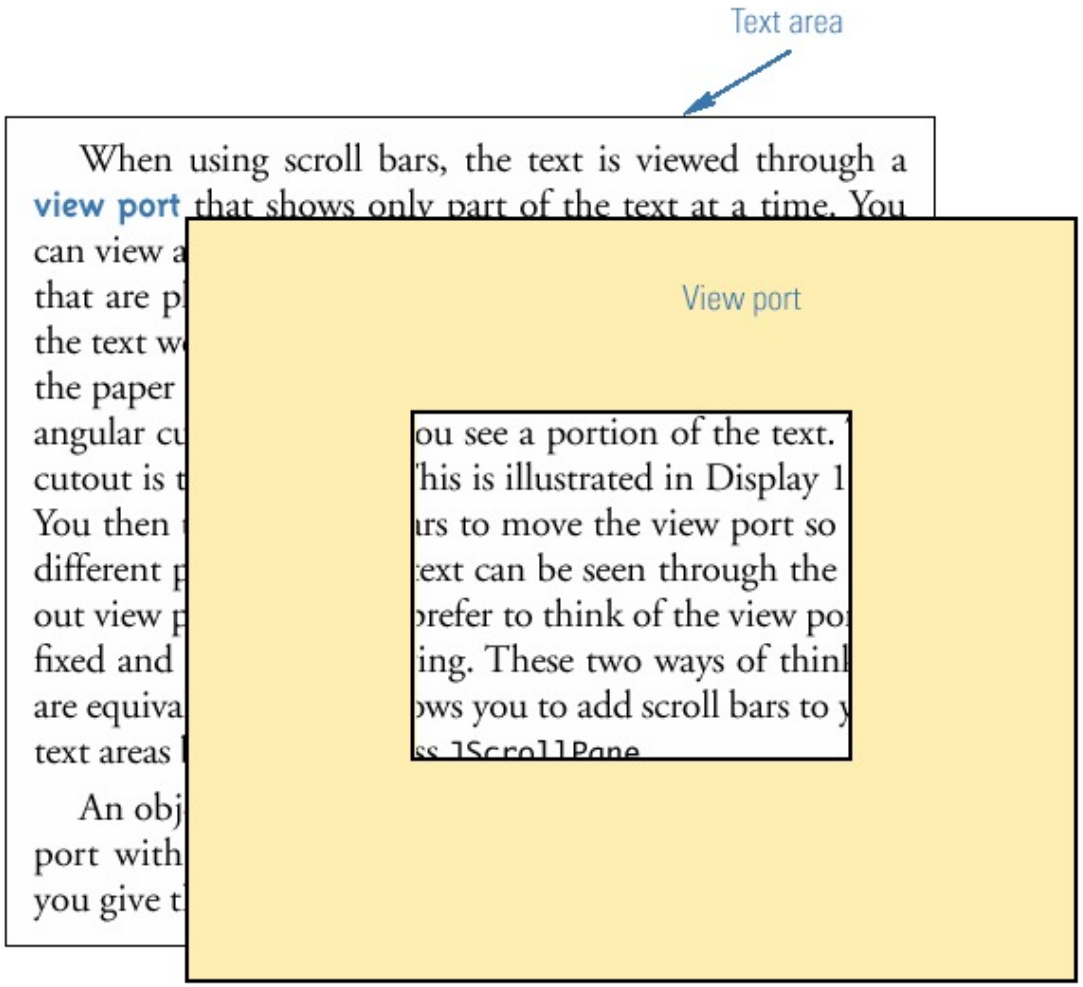
Scroll Bars

- When using scroll bars, the text is viewed through a *view port* that shows only part of the text at a time:
  - A different part of the text may be viewed by using the scroll bars placed along the side and bottom of the view port.

- Scroll bars can be added to text areas using the **JScrollPane** class:
  - The **JScrollPane** class is in the **javax.swing** package.
  - An object of the class **JScrollPane** is like a view port with scroll bars.

View Port of a Text Area

Display 18.6 View Port for a Text Area



Scroll Bars

- The scroll bar policies can be set as follows:

```
JScrollPane scrolledText = new JScrollPane(memoDisplay);
scrolledText.setHorizontalScrollBarPolicy(
   JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
scrolledText.setVerticalScrollBarPolicy(
   JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
```

- If invocations of these methods are omitted, then the scroll bars will be visible only when needed:
  - If all the text fits in the view port, then no scroll bars will be visible.
  - If enough text is added, the scroll bars will appear automatically.

Using Scroll Bars (1/8)

Display 19.8 A Text Area with Scroll Bars

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

(continued)

Using Scroll Bars (2/8)

Display 19.8 A Text Area with Scroll Bars

```
12  public class ScrollBarDemo extends JFrame
13                              implements ActionListener
14  {
15      public static final int WIDTH = 600;
16      public static final int HEIGHT = 400;
17      public static final int LINES = 15;
18      public static final int CHAR_PER_LINE = 30;


19      private JTextArea memoDisplay;
20      private String memo1;
21      private String memo2;
```

(continued)

Using Scroll Bars (3/8)

Display 19.8 A Text Area with Scroll Bars

```
22      public static void main(String[] args)
23      {
24          ScrollBarDemo gui = new ScrollBarDemo();
25          gui.setVisible(true);
26      }


27      public ScrollBarDemo()
28      {
29          super("Scroll Bars Demo");
30          setSize(WIDTH, HEIGHT);
31          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

(continued)

Using Scroll Bars (4/8)

Display 19.8 A Text Area with Scroll Bars

```
32          JPanel buttonPanel = new JPanel();
33          buttonPanel.setBackground(Color.LIGHT_GRAY);
34          buttonPanel.setLayout(new FlowLayout());
35          JButton memo1Button = new JButton("Save Memo 1");
36          memo1Button.addActionListener(this);
37          buttonPanel.add(memo1Button);

38          JButton memo2Button = new JButton("Save Memo 2");
39          memo2Button.addActionListener(this);
40          buttonPanel.add(memo2Button);

41          JButton clearButton = new JButton("Clear");
42          clearButton.addActionListener(this);
43          buttonPanel.add(clearButton);
```

(continued)

Using Scroll Bars (5/8)

Display 19.8 A Text Area with Scroll Bars

```
44          JButton get1Button = new JButton("Get Memo 1");
45          get1Button.addActionListener(this);
46          buttonPanel.add(get1Button);

47          JButton get2Button = new JButton("Get Memo 2");
48          get2Button.addActionListener(this);
49          buttonPanel.add(get2Button);

50          add(buttonPanel, BorderLayout.SOUTH);

51          JPanel textPanel = new JPanel();
52          textPanel.setBackground(Color.BLUE);
```

(continued)

Using Scroll Bars (6/8)

Display 19.8 A Text Area with Scroll Bars

```
53          memoDisplay = new JTextArea(LINES, CHAR_PER_LINE);
54          memoDisplay.setBackground(Color.WHITE);

55          JScrollPane scrolledText = new JScrollPane(memoDisplay);
56          scrolledText.setHorizontalScrollBarPolicy(
57                      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
58          scrolledText.setVerticalScrollBarPolicy(
59                      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

60          textPanel.add(scrolledText);

61          add(textPanel, BorderLayout.CENTER);
62      }
```

(continued)

Using Scroll Bars (7/8)

Display 19.8 A Text Area with Scroll Bars

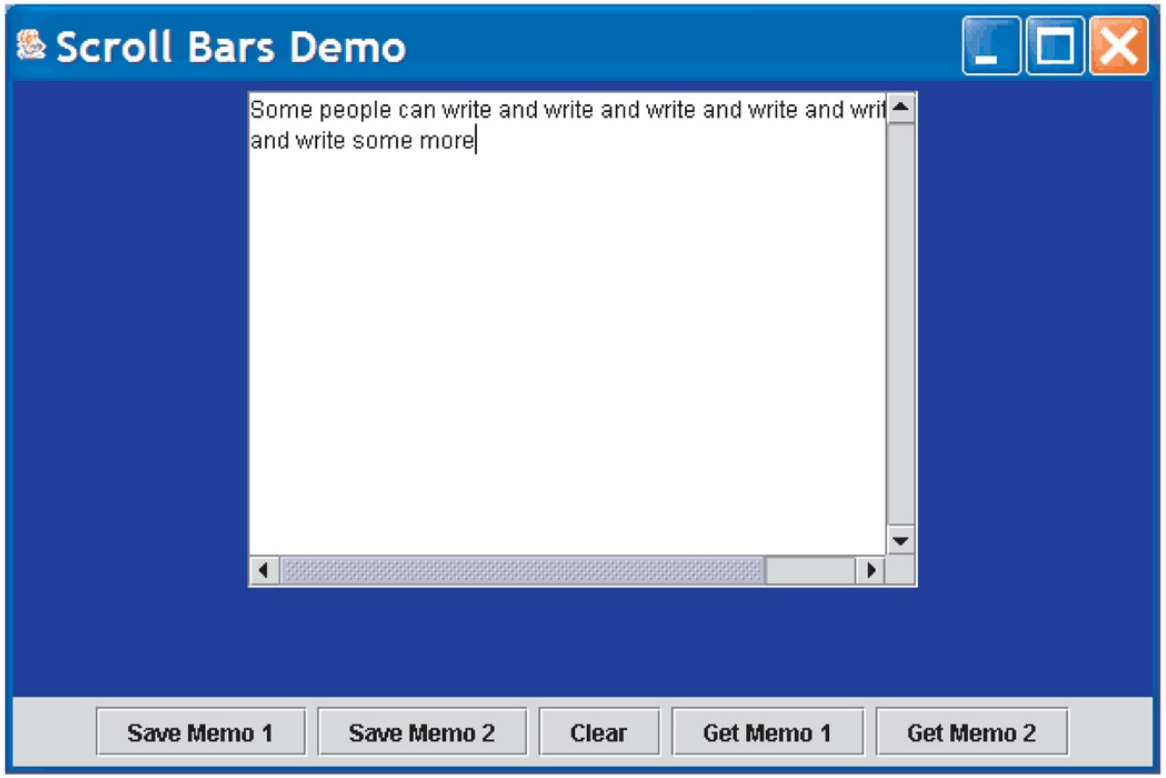
```
63        public void actionPerformed(ActionEvent e)
64        {
65            String actionCommand = e.getActionCommand();


66            if (actionCommand.equals("Save Memo 1"))
67                memo1 = memoDisplay.getText();
68            else if (actionCommand.equals("Save Memo 2"))
69                memo2 = memoDisplay.getText();
70            else if (actionCommand.equals("Clear"))
71                memoDisplay.setText("");
72            else if (actionCommand.equals("Get Memo 1"))
73                memoDisplay.setText(memo1);
74            else if (actionCommand.equals("Get Memo 2"))
75                memoDisplay.setText(memo2);
76            else
77                memoDisplay.setText("Error in memo interface");
78        }
79    }
```

(continued)

Using Scroll Bars (8/8)

Display 19.8 **A Text Area with Scroll Bars**

RESULTING GUI



Methods for `JScrollPane` (1/2)

Display 19.7 Some Methods in the Class `JScrollPane`

The `JScrollPane` class is in the `javax.swing` package.

```
public JScrollPane(Component objectToBeScrolled)
```

Creates a new `JScrollPane` for the `objectToBeScrolled`. Note that the `objectToBeScrolled` need not be a `JTextArea`, although that is the only type of argument considered in this book.

```
public void setHorizontalScrollBarPolicy(int policy)
```

Sets the policy for showing the horizontal scroll bar. The `policy` should be one of

```
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

The phrase AS_NEEDED means the scroll bar is shown only when it is needed. This is explained more fully in the text. The meanings of the other policy constants are obvious from their names.
(As indicated, these constants are defined in the class `JScrollPane`. You should not need to even be aware of the fact that they have `int` values. Think of them as policies, not as `int` values.)

(continued)

Methods for `JScrollPane` (2/2)

Display 19.7 Some Methods in the Class `JScrollPane`

```
public void setVerticalScrollBarPolicy(int policy)
```

Sets the policy for showing the vertical scroll bar. The `policy` should be one of

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
```

The phrase AS_NEEDED means the scroll bar is shown only when it is needed. This is explained more fully in the text. The meanings of the other policy constants are obvious from their names.
(As indicated, these constants are defined in the class `JScrollPane`. You should not need to even be aware of the fact that they have `int` values. Think of them as policies, not as `int` values.)

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.

- Menus can be added to other menus to create nested menus.

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 **this** parameter has been registered as an action listener for the menu item.

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:

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

- The menu bar can be added to a **JFrame** in two different ways:
  - Using the **setJMenuBar** method: **setJMenuBar(bar)**.
  - 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 (1/8)

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 (2/8)

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 (3/8)

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 (4/8)

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 (5/8)

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 (6/8)

Display 17.14 A GUI with a Menu

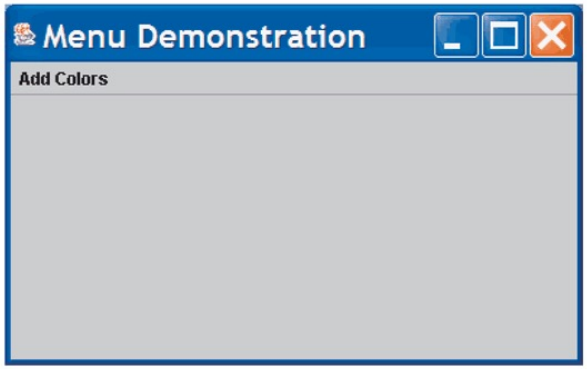
```
51      public void actionPerformed(ActionEvent e)
52      {
53          String buttonString = e.getActionCommand();

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)

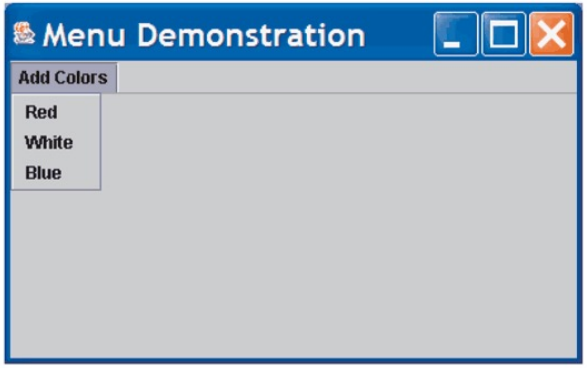
A GUI with a Menu (7/8)

Display 17.14 **A GUI with a Menu**

RESULTING GUI



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

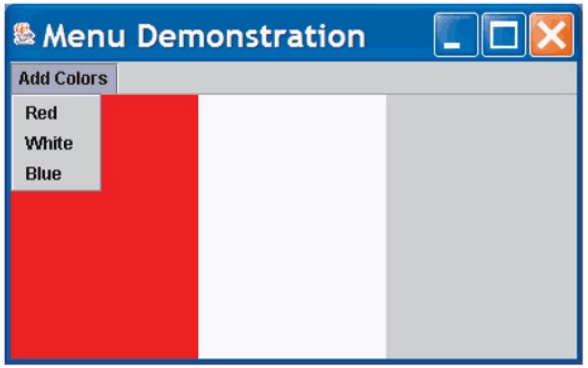


(continued)

A GUI with a Menu (8/8)

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)

The `validate` Method

- An invocation of **validate** causes a container to lay out its components again:
  - It is a kind of "update" method that makes changes in the components shown on the screen.
  - Every container class has the **validate** method, which has no arguments.

- Many simple changes made to a Swing GUI happen automatically, while others require an invocation of **validate** or some other "update" method:
  - When in doubt, it will do no harm to invoke **validate**.