CHAPTER 10

GRAPHICAL USER INTERFACES

Slides by Donald W. Smith
TechNeTrain.com

Final Draft
10/30/11

Chapter Goals

- To implement simple graphical user interfaces
- To add buttons, text fields, and other components to a frame window
- To handle events that are generated by buttons
- To write programs that display simple drawings

In this chapter, you will learn how to write graphical user-interface applications, process the events that are generated by button clicks, and process user input,

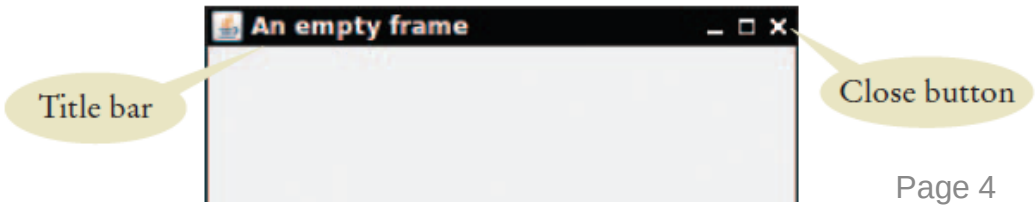
Contents

- Frame Windows
- Events and Event Handling
- Processing Text Input
- Creating Drawings

10.1 Frame Windows

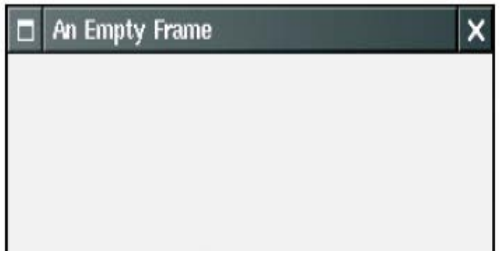
- Java provides classes to create graphical applications that can run on any major graphical user interface
  - A graphical application shows information inside a **frame**: a window with a title bar

- Java's `JFrame` class allows you to display a frame
  - `JFrame` class is part of the `javax.swing` package (the graphical user-interface library)
    - X: demotes the Swing started out as a Java *extension* before added to the standard library



The JFrame Class

- Five steps to displaying a frame:
  1) Construct an object of the JFrame class




```
JFrame frame = new JFrame();
```

  2) Set the size of the frame

Width, height

```
frame.setSize(300,400);
```

If you omit this step the frame will be 0 by 0 pixels, and you won't be able to see it.

  3) Set the title of the frame(optional)

```
frame.setTitle("An Empty Frame");
```

  4) Set the "default close operation"

When the user closes the frame, the program automatically exits.

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

  5) Make it visible

```
frame.setVisible (true);
```

EmptyFrameViewer.java

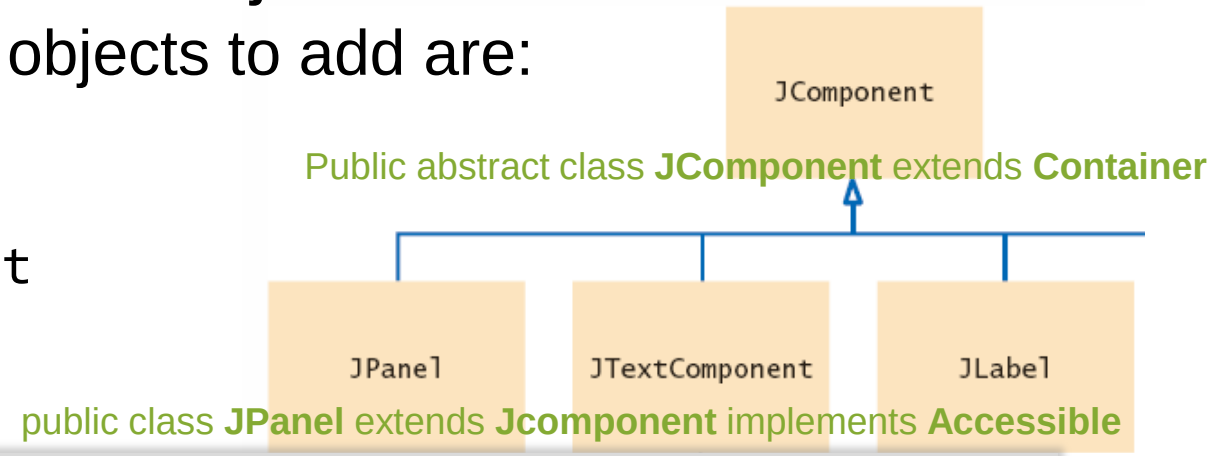
- Your JVM (Java Virtual Machine does all of the work of displaying the frame on your GUI)
  - This application is portable to all supported GUIs!

```
import javax.swing.JFrame;

/**
   This program displays an empty frame.
*/
public class EmptyFrameViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      final int FRAME_WIDTH = 300;
      final int FRAME_HEIGHT = 400;
      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setTitle("An empty frame");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.setVisible(true);
   }
}
```

You are using the java `Swing` library

Adding Components

- You CANNOT draw directly on a `JFrame` object
- Instead, construct an object and add it to the frame
  - A few examples objects to add are:
    - `JComponent`
    - `Jpanel`
    - `JTextComponent`
    - `JLabel`




```
public class RectangleComponent extends JComponent
{
  public void paintComponent(Graphics g)
  {
    // Drawing instructions go here
  }
}
```

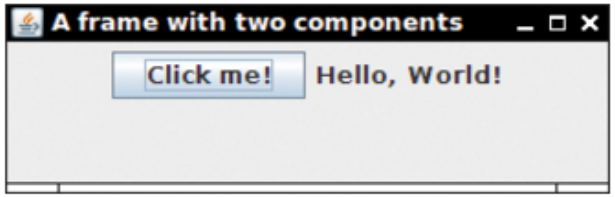
With the exception of top-level containers, all Swing components whose names begin with "**J**" descend from the **JComponent** class

Extend the `JComponent` Class and override its `paintComponent` method

Adding Panels

Ex. buttons, text labels

- If you have more than one component, put them into a **panel** (a **container** for other user-interface components), and then add the panel to the frame:



  - First Create the components

```
JButton button = new JButton("Click me!");
JLabel label = new JLabel("Hello, World!");
```

  - Then Add them to the panel

```
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```

Use a `JPanel` to group multiple user-interface components together.

Add the panel to the frame

Prevent to add components directly to the frame, they get placed on top of each other.

FilledFrameViewer.java

```java
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

/**
   This program shows a frame that is filled with two components.
*/
public class FilledFrameViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      JButton button = new JButton("Click me!");
      JLabel label = new JLabel("Hello, World!");

      JPanel panel = new JPanel();
      panel.add(button);
      panel.add(label);
      frame.add(panel);

      final int FRAME_WIDTH = 300;
      final int FRAME_HEIGHT = 100;
      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setTitle("A frame with two components");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      frame.setVisible(true);
   }
}
```

Using Inheritance to Customize Frames

- For complex frames that has many components:
  - Design a subclass of `JFrame`
  - Store the components as instance variables
  - Initialize them in the constructor of your subclass.

```
public class FilledFrame extends JFrame
{
   private JButton button;
   private JLabel label;
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 100;

   public FilledFrame()
   {
      createComponents();
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }
}
```

Components are instance variables

Initialize and add them in the constructor of your subclass with a helper method

Using Inheritance to Customize Frames

- Cont'd

```
private void createComponents()
  {
   button = new JButton("Click me!");
   label = new JLabel("Hello, World!");
   JPanel panel = new JPanel();
   panel.add(button);
   panel.add(label);
   add(panel);
  }
```

Using Inheritance to Customize Frames

- Then instantiate the customized frame from the `main` method

```
public class FilledFrameViewer2
{
   public static void main(String[] args)
   {
     JFrame frame = new FilledFrame();
     frame.setTitle("A frame with two components");
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.setVisible(true);
   }
}
```

Special Topic 10.1

- **Adding the `main` Method to the Frame Class**
  - Some programmers prefer this technique
    - e.g. combine the `FilledFrame` and `FilledFrameViewer2` classes

```
public class FilledFrame extends JFrame
{
  . . .
  public static void main(String[] args)
  {
    JFrame frame = new FilledFrame();
    frame.setTitle("A frame with two components");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
  public FilledFrame()
  {
    createComponents();
    setSize(FRAME_WIDTH, FRAME_HEIGHT);
  }
  . . .
}
```

Once main has instantiated the `FilledFrame`, non-static instance variables and methods can be used.

Classes `Graphics` and `Graphics2D`

- `Graphics` class lets you manipulate the graphics state (such as current color)
- `Graphics2D` class has methods to draw shape objects
- Use a cast to recover the `Graphics2D` object from the `Graphics` parameter:

```
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      // Recover Graphics2D
      Graphics2D g2 = (Graphics2D) g;
      . . .
   }
}
```

Classes `Graphics` and `Graphics2D`

- Call method `draw` of the `Graphics2D` class to draw shapes, such as rectangles, ellipses, line segments, polygons, and arcs:

```
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      . . .
      Rectangle box = new Rectangle(5, 10, 20, 30);
      g2.draw(box);
      . . .
   }
}
```

Drawing Rectangles

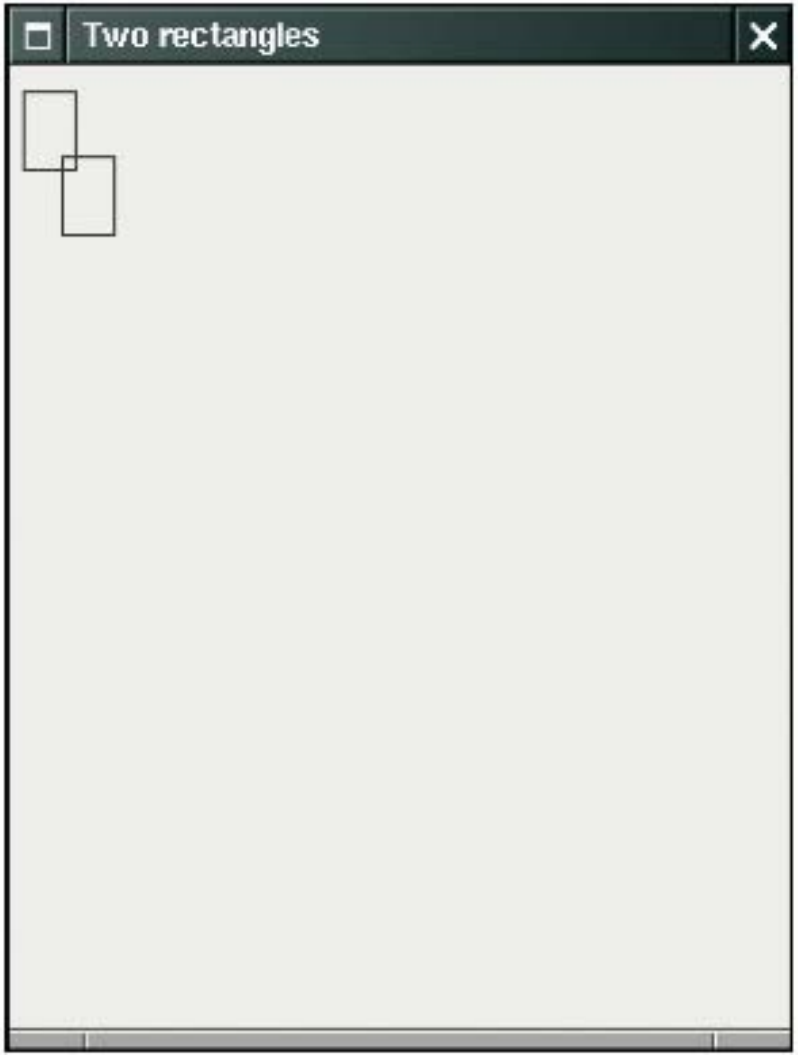


Figure 23
Drawing Rectangles

rectangles/RectangleComponent.java

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
   A component that draws two rectangles.
*/
public class RectangleComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      // Recover Graphics2D
      Graphics2D g2 = (Graphics2D) g;

      // Construct a rectangle and draw it
      Rectangle box = new Rectangle(5, 10, 20, 30);
      g2.draw(box);

```

Continued

rectangles/RectangleComponent.java (cont.)

```
        // Move rectangle 15 units to the right and 25 units down
        box.translate(15, 25);

        // Draw moved rectangle
        g2.draw(box);
    }
}
```

Using a Component

1. Construct a frame.
2. Construct an object of your component class:
```
RectangleComponent component = new RectangleComponent();
```
3. Add the component to the frame:
```
frame.add(component);
```
4. Make the frame visible.

rectangles/RectangleViewer.java

```
import javax.swing.JFrame;

public class RectangleViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      frame.setSize(300, 400);
      frame.setTitle("Two rectangles");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      RectangleComponent component = new RectangleComponent();
      frame.add(component);

      frame.setVisible(true);
   }
}
```

10.2 Events and Event Handling

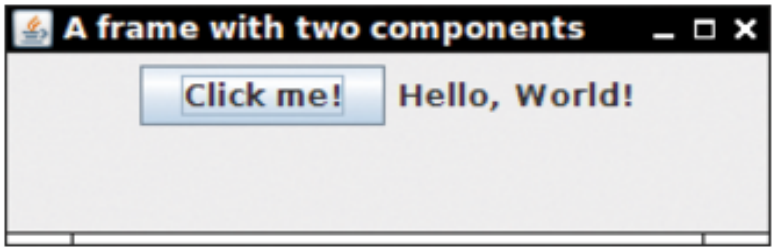
- In a modern **graphical user interface** program, the user controls the program through the *mouse* and *keyboard*.
- The user can enter information into text fields, pull down menus, click buttons, and drag scroll bars in any order.
  - The program must react to the user commands
  - The program can choose to receive and handle events such as “mouse move” or a button push “action event”

Events and Action Listeners

- Programs must indicate which events it wants to receive
  - E.g. Button clicks, mouse inputs
- It does so by installing event listener objects
  - These *event listener objects* are **instances** of classes that you declare
  - The **methods** of your event listener classes contain the instructions that you want to have executed when the events occur
- To install a listener, you need to know the event source
  - E.g. Button
- You add an event listener object to selected event sources:
  - E.g. OK Button clicked, Cancel Button clicked, Menu Choice..
- Whenever the event occurs, the event source calls the appropriate methods of all attached event listeners

Events and Action Listeners

Event source:
The user-interface component, e.g. a button




```
JButton button = new JButton("Click me!");
JLabel label = new JLabel("Hello, World!");
```

Event listener:
The event needs to receive, e.g. “I was clicked”

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

/**
   An action listener that prints a message.
*/
public class ClickListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      System.out.println("I was clicked.");
   }
}
```

Example ActionListener

- The `ActionListener` interface has one method:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

- `ClickListener` class implements the `ActionListener` interface

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

/**
   An action listener that prints a message.
*/
public class ClickListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      System.out.println("I was clicked.");
   }
}
```

The event handling classes are defined in the **java.awt.event package**. (AWT is the Abstract Window Toolkit, the Java library for dealing with windows and events.)

We can ignore the *event parameter* – it has information such as when the event occurred

Registering ActionListener

- A `ClickListener` object must be created, and then 'registered' (added) to a specific event source

```
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

- Now whenever the `button` object is clicked, it will call

```
listener.actionPerformed(event);
```

passing it the event as a parameter

ButtonFrame1.java

```
/**
   This frame demonstrates how to install an action listener.
*/
public class ButtonFrame1 extends JFrame
{
   private static final int FRAME_WIDTH = 100;
   private static final int FRAME_HEIGHT = 60;

   public ButtonFrame1()
   {
      createComponents();
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   private void createComponents()
   {
      JButton button = new JButton("Click me!");
      JPanel panel = new JPanel();
      panel.add(button);
      add(panel);

      ActionListener listener = new ClickListener();
      button.addActionListener(listener);
   }
}
```

Creates and adds a `JButton` to the frame

Tells the button to ‘call us back’ when an event occurs.

ButtonViewer1.java

- No changes required to the main to implement an event handler

```
1   import javax.swing.JFrame;
2
3   /**
4      This program demonstrates how to install an action listener.
5   */
6   public class ButtonViewer1
7   {
8      public static void main(String[] args)
9      {
10        JFrame frame = new ButtonFrame1();
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12        frame.setVisible(true);
13     }
14  }
```

- *actionPerform()*:
  - The method that will be executed after the user hit Enter
  - Implemented in *ActionListener* interface
  - *ActionListener* defined in *java.awt.event* library

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

/**
   An action listener that prints a message.
*/
public class ClickListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      System.out.println("I was clicked.");
   }
}
```

- *caretUpdate()*:
  - The method that will be executed after any change of the textfield
  - Implemented in *CaretListener* interface
  - *CaretListener* defined in *java.swing.event* library

```
import java.awt.event.ActionEvent;
Import java.awt.event.ActionListener;

public class caretUpdate implements CaretListener
{
    public void caretUpdate(CaretEvent event)
    {
        System.out.println("Your input is: ");
    }
}
```

AWT v.s. Swing

- Swing (Oracle's Java Foundation Classes, JFC)
  - JFC encompasses a group of features for *building graphical user interfaces (GUIs)* and *adding rich graphics functionality* and interactivity to Java applications.
  - Swing GUI Components – one feature of JFC
    - Includes everything from buttons to split panes to tables. Many components are capable of sorting, printing, and drag and drop, to name a few of the supported features.

- Java.awt (Abstract Window Toolkit)
  - Contains all of the classes for creating user interfaces and for painting graphics and images.
  - In 1995, AWT widgets provided a thin level of abstraction over the underlying native user-interface.
    - For example, a check box on Microsoft Windows is not exactly the same as a check box on Mac OS or on the various types of Unix.

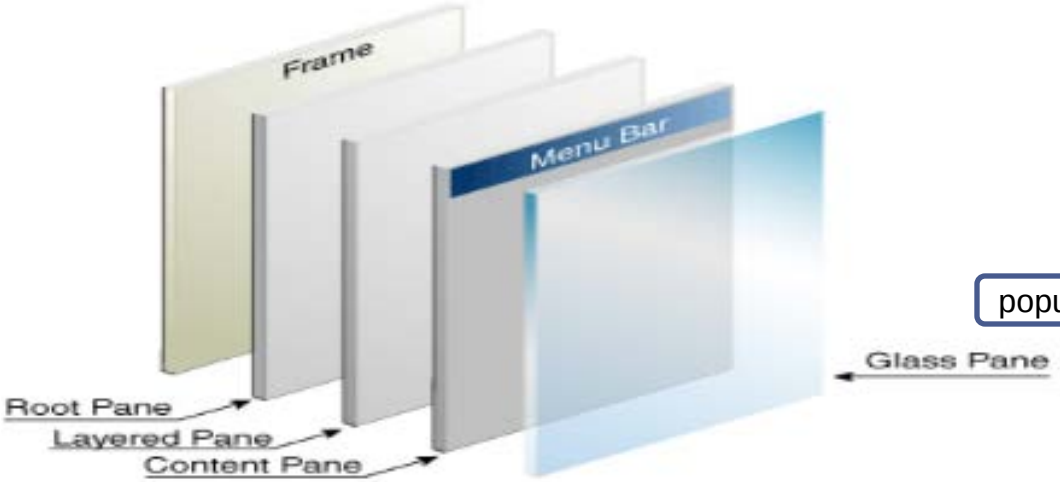
- The Swing API has 18 public packages:

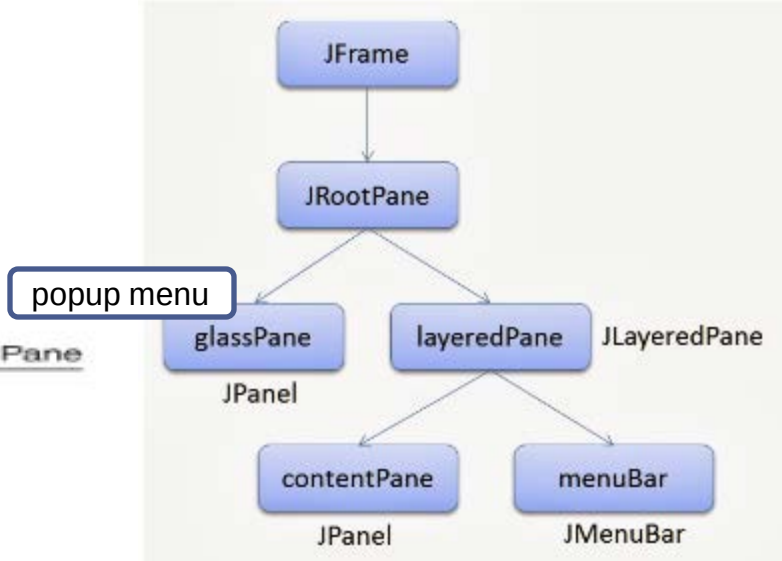
javax.accessibility	javax.swing.plaf	javax.swing.text
javax.swing	javax.swing.plaf.basic	javax.swing.text.html
javax.swing.border	javax.swing.plaf.metal	javax.swing.text.html.parser
javax.swing.colorchooser	javax.swing.plaf.multi	javax.swing.text.rtf
javax.swing.event	javax.swing.plaf.synth	javax.swing.tree
javax.swing.filechooser	javax.swing.table	javax.swing.undo

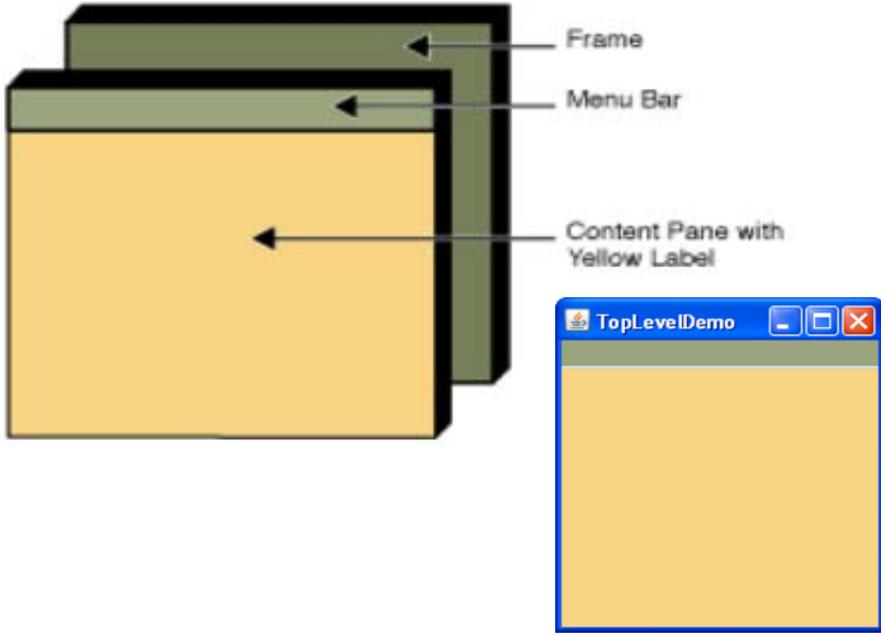
```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

```

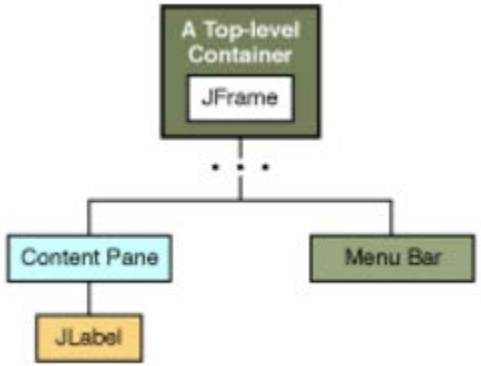
Public abstract class **Jcomponent** extends **Container**

- Swing provides three generally useful top-level container classes: *JFrame*, *JDialog*, and *JApplet*
- To appear on screen, every GUI component must be part of a containment hierarchy.


Frame
Menu Bar
Glass Pane
Root Pane
Layered Pane
Content Pane


JFrame
JRootPane
popup menu
glassPane
JPanel
layeredPane
JLayeredPane
contentPane
JPanel
menuBar
JMenuBar


Frame
Menu Bar
Content Pane with
Yellow Label
TopLevelDemo


A Top-level
Container
JFrame
Content Pane
Menu Bar
JLabel

Inner Classes for Listeners

- In the preceding section, you saw how the code that is executed when a button is clicked is placed into a listener class.
- Inner Classes are often used for ActionListeners
- An Inner class is a class that is declared **inside** another class
  - It may be declared inside or outside a method of the class
- Why inner classes? Two reasons:
  1) It places the trivial listener class exactly where it is needed, without cluttering up the remainder of the project
  2) Their methods can access variables that are declared in surrounding blocks.
  - In this regard, inner classes declared inside methods behave similarly to nested blocks

Example Inner Class Listener

- The inner class `ClickListener` declared inside the class `ButtonFrame2` can access local variables inside the surrounding scope

Outer Block

Inner Block

```
public class ButtonFrame2 extends JFrame
{
  private JButton button;
  private JLabel label;
  . . .
  class ClickListener implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
    {
      label.setText("I was clicked");
    }
  }
  . . .
}
```

Can easily access methods of the private instance of a label object.

ButtonFrame2.java (1)

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class ButtonFrame2 extends JFrame
{
   private JButton button;
   private JLabel label;

   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 100;

   public ButtonFrame2()
   {
      createComponents();
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

```

ButtonFrame2.java (2)

- Changes label from “Hello World!” to “I was clicked.”:

```
22      /**
23         An action listener that changes the label text.
24      */
25      class ClickListener implements ActionListener
26      {
27         public void actionPerformed(ActionEvent event)
28         {
29            label.setText("I was clicked.");
30         }
31      }
32
33      private void createComponents()
34      {
35         button = new JButton("Click me!");
36         ActionListener listener = new ClickListener();
37         button.addActionListener(listener);
38
39         label = new JLabel("Hello, World!");
40
41         JPanel panel = new JPanel();
42         panel.add(button);
43         panel.add(label);
44         add(panel);
45      }
46   }
```

ButtonFrame1.java v.s. ButtonFrame2.java
Outer class v.s. Inner Class

```java
public class ClickListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      System.out.println("I was clicked.");
   }
}
```

```java
public class ButtonFrame1 extends JFrame
{
   private static final int FRAME_WIDTH = 100;
   private static final int FRAME_HEIGHT = 60;

   public ButtonFrame1()
   {
      createComponents();
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   private void createComponents()
   {
      JButton button = new JButton("Click me!");
      JPanel panel = new JPanel();
      panel.add(button);
      add(panel);

      ActionListener listener = new ClickListener();
      button.addActionListener(listener);
   }
}
```

```java
public class ButtonFrame2 extends JFrame
{
   private JButton button;
   private JLabel label;

   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 100;

   public ButtonFrame2()
   {
      createComponents();
      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }
   /**
      An action listener that changes the label text.
   */
   class ClickListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         label.setText("I was clicked.");
      }
   }

   private void createComponents()
   {
      button = new JButton("Click me!");
      ActionListener listener = new ClickListener();
      button.addActionListener(listener);

      label = new JLabel("Hello, World!");

      JPanel panel = new JPanel();
      panel.add(button);
      panel.add(label);
      add(panel);
   }
}
```

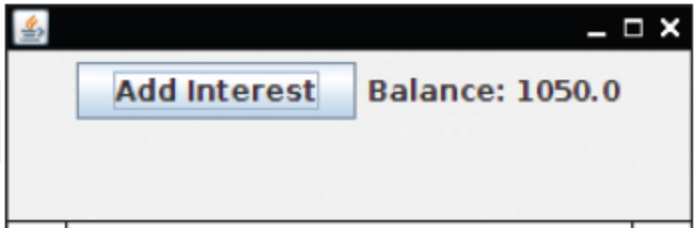
InvestmentFrame.java (1)

```
1   import java.awt.event.ActionEvent;
2   import java.awt.event.ActionListener;
3   import javax.swing.JButton;
4   import javax.swing.JFrame;
5   import javax.swing.JLabel;
6   import javax.swing.JPanel;
7
8   public class InvestmentFrame extends JFrame
9   {
10     private JButton button;
11     private JLabel resultLabel;
12     private double balance;
13
14     private static final int FRAME_WIDTH = 300;
15     private static final int FRAME_HEIGHT = 100;
16
17     private static final double INTEREST_RATE = 5;
18     private static final double INITIAL_BALANCE = 1000;
19
20     public InvestmentFrame()
21     {
22        balance = INITIAL_BALANCE;
23
24        createComponents();
25        setSize(FRAME_WIDTH, FRAME_HEIGHT);
26     }
```

Application:
Showing Growth of an Investment
-Whenever the user clicks a button, 5 percent interest is added, and the new balance is displayed

InvestmentFrame.java (2)

```
28      /**
29         Adds interest to the balance and updates the display.
30      */
31      class AddInterestListener implements ActionListener
32      {
33         public void actionPerformed(ActionEvent event)
34         {
35            double interest = balance * INTEREST_RATE / 100;
36            balance = balance + interest;
37            resultLabel.setText("Balance: " + balance);
38         }
39      }
40
41      private void createComponents()
42      {
43         button = new JButton("Add Interest");
44         ActionListener listener = new AddInterestListener();
45         button.addActionListener(listener);
46
47         resultLabel = new JLabel("Balance: " + balance);
48
49         JPanel panel = new JPanel();
50         panel.add(button);
51         panel.add(resultLabel);
52         add(panel);
53      }
54   }
```



- User clicks the button four times for output:

Common Error 10.1

- **Modifying Parameter Types in the Implementing Method**

```
public interface ActionListener
{
  void actionPerformed(ActionEvent event);
}
```

  - When you implement an interface, you must declare each method exactly as it is specified in the interface.
  - Accidentally making small changes to the parameter types is a common error: For example:

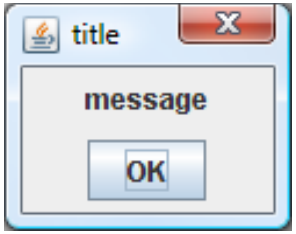
```
class MyListener implements ActionListener
{
  public void actionPerformed()
  // Oops . . . forgot ActionEvent parameter
  {     . . .    }
}
```

Common Error 10.2

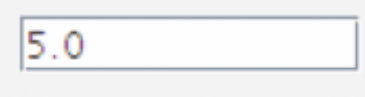
- **Forgetting to Attach a Listener**
  - If you run your program and find that your buttons seem to be dead, double-check that you attached the button listener.
  - The same holds for other user-interface components. It is a surprisingly common error to program the listener class and the event handler action without actually attaching the listener to the event source.

```
...
ActionListener listener = new ClickListener();
button.addActionListener(listener);
```

10.3 Processing Text Input



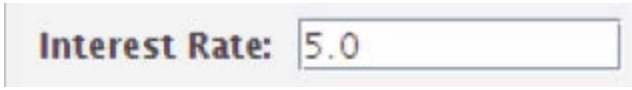
- **Dialog boxes** allows for user input… but
  - Popping up a separate dialog box for each input is not a natural user interface
- Most graphical programs collect text input through **text fields**
  - The `JTextField` class provides a text field
    - When you construct a text field, supply the width:
      - The approximate number of *characters* that you expect
      - If the user exceeds this number, text will 'scroll' left




```
final int FIELD_WIDTH = 10;
final JTextField rateField = new JTextField(FIELD_WIDTH);
```

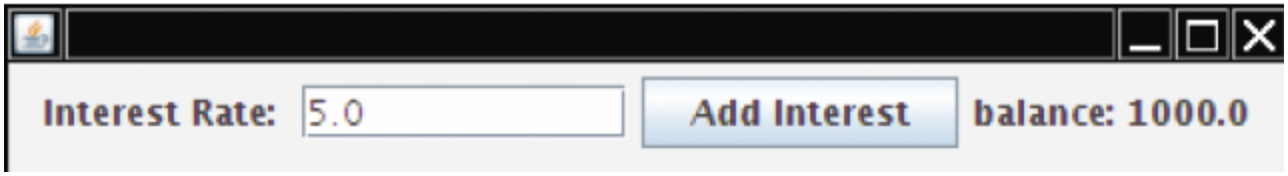
Add a Label and a Button

- A **Label** helps the user know what you want
  - Normally to the left of a textbox




```
JLabel rateLabel = new JLabel("Interest Rate: ");
```

- A **Button** with an `actionPerformed` method can be used to read the text from the textbox with the `getText` method
  - Note that `getText` returns a **`String`**, and must be converted to a numeric value if it will be used in calculations




```
double rate = Double.parseDouble(rateField.getText());
double interest = account.getBalance() * rate / 100;
account.deposit(interest);
resultLabel.setText("balance: " + account.getBalance());
```

InvestmentFrame2.java

```
/**
   A frame that shows the growth of an investment with variable interest.
*/
public class InvestmentFrame2 extends JFrame
{
   private static final int FRAME_WIDTH = 450;
   private static final int FRAME_HEIGHT = 100;

   private static final double DEFAULT_RATE = 5;
   private static final double INITIAL_BALANCE = 1000;

   private JLabel rateLabel;
   private JTextField rateField;
   private JButton button;
   private JLabel resultLabel;
   private double balance;

   public InvestmentFrame2()
   {
      balance = INITIAL_BALANCE;

      resultLabel = new JLabel("Balance: " + balance);

      createTextField();
      createButton();
      createPanel();

```

- Use this as a framework for GUIs that do calculations

Place input components into the frame

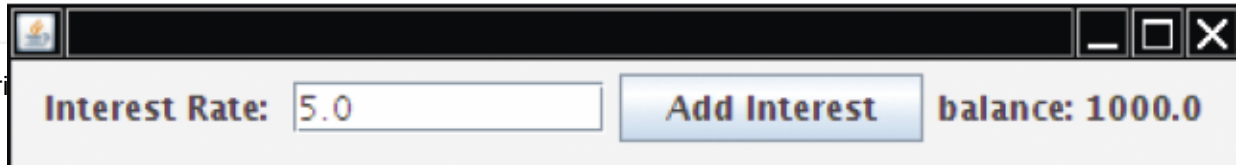
InvestmentFrame2.java (2)

```
   private void createTextField()
   {
      rateLabel = new JLabel("Interest Rate: ");

      final int FIELD_WIDTH = 10;
      rateField = new JTextField(FIELD_WIDTH);
      rateField.setText("" + DEFAULT_RATE);
   }

   /**
      Adds interest to the balance and updates the display.
   */
   class AddInterestListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         double rate = Double.parseDouble(rateField.getText());
         double interest = balance * rate / 100;
         balance = balance + interest;
         resultLabel.setText("Balance: " + balance);
      }
   }

   private void createButton()
   {
      button = new JButton("Add Interest");

      ActionListener listener = new AddInterestListener();
      button.addActionListener(listener);
   }
```

```
   private void createPanel()
   {
      panel = new JPanel();
      panel.add(rateLabel);
      panel.add(rateField);
      panel.add(button);
      panel.add(resultLabel);
      add(panel);
   }
```

Do calculations in ActionPerformed method

Keep the code for the listener and the object (Button) in the same area



Text Areas

- `JTextField`: holds a single line of text
- Create multi-line text areas with a `JTextArea` object
  - Set the size in rows and columns

```
final int ROWS = 10;
final int COLUMNS = 30;
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
```

  - Use the **`setText`** method to set the text of a *text field* or *text area*

```
textArea.setText("Account Balance");
```

Text Areas

- The **`append`** method adds text to the end of a text area
  - Use newline characters to separate lines

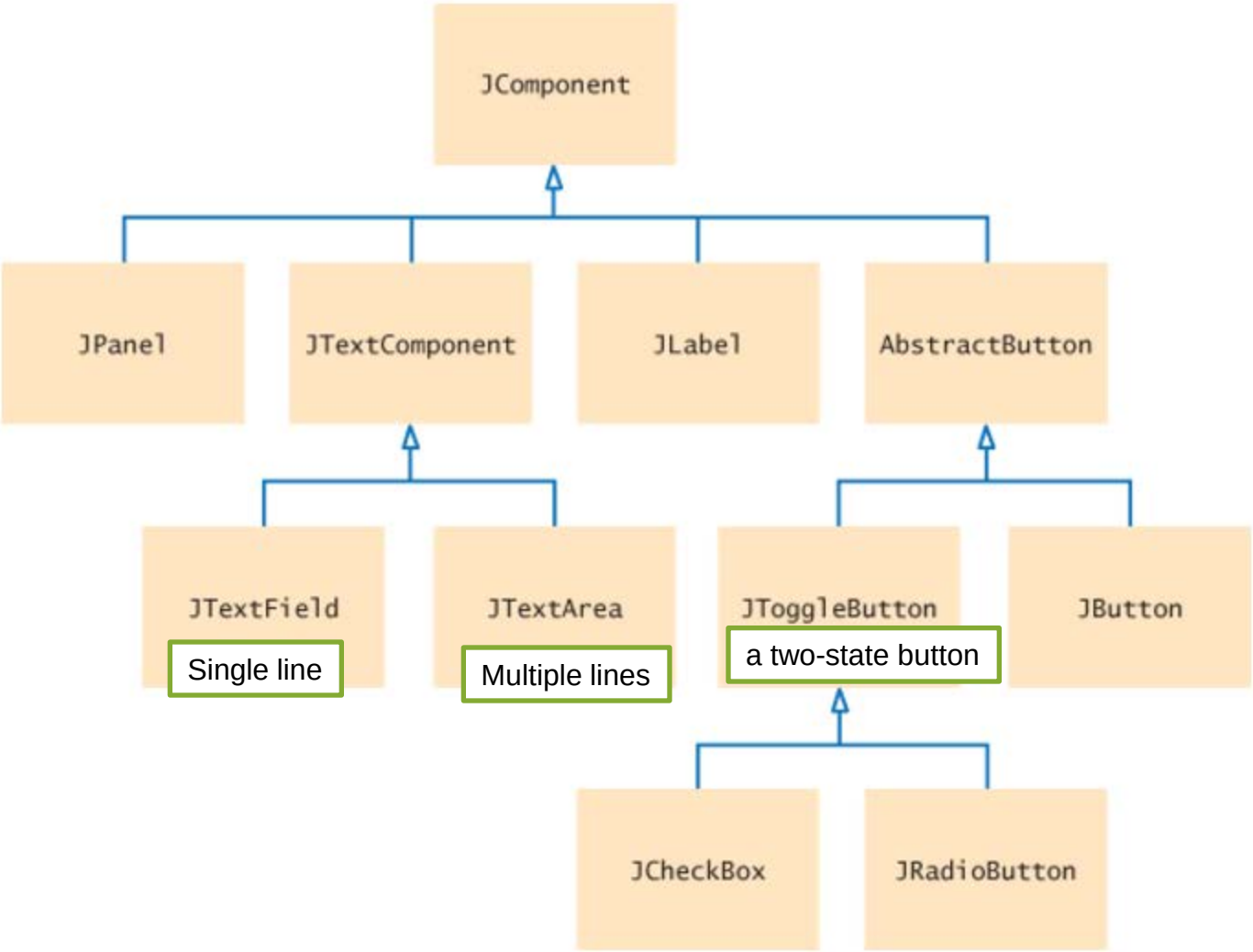
```
textArea.append(account.getBalance() + "\n");
```

- Use the **`setEditable`** method to control user input

```
textArea.setEditable(false);
```

  - To use a text field or text area for display purposes only
  - The user can no longer edit the contents of the field, but your program can still call `setText` and append to change it.

A Part of the Hierarchy of Swing User-Interface Components

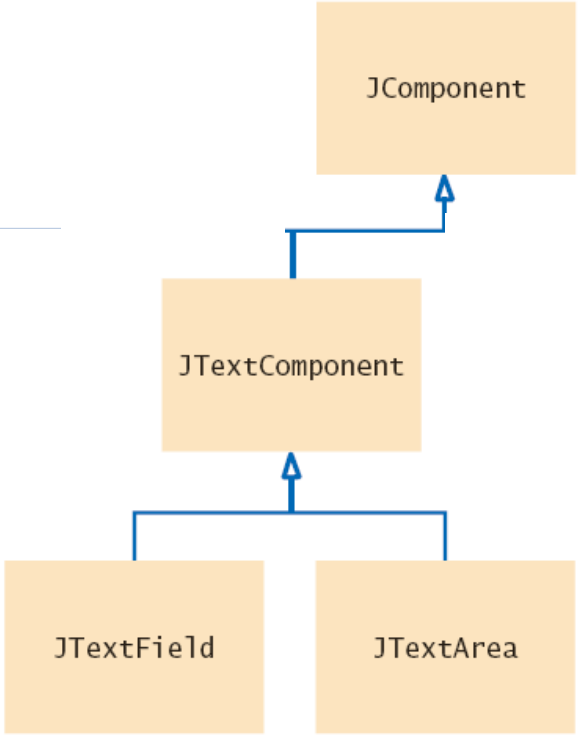


JTextField and JTextArea

- JTextField and JTextArea are inherit from JTextComponent:
  - The methods declared in the JTextComponent class are inherited by JTextField and JTextArea
  - setText
  - setEditable

The methods setText and setEditable are declared in the JTextComponent class

However, the append method is declared in the JTextArea class.



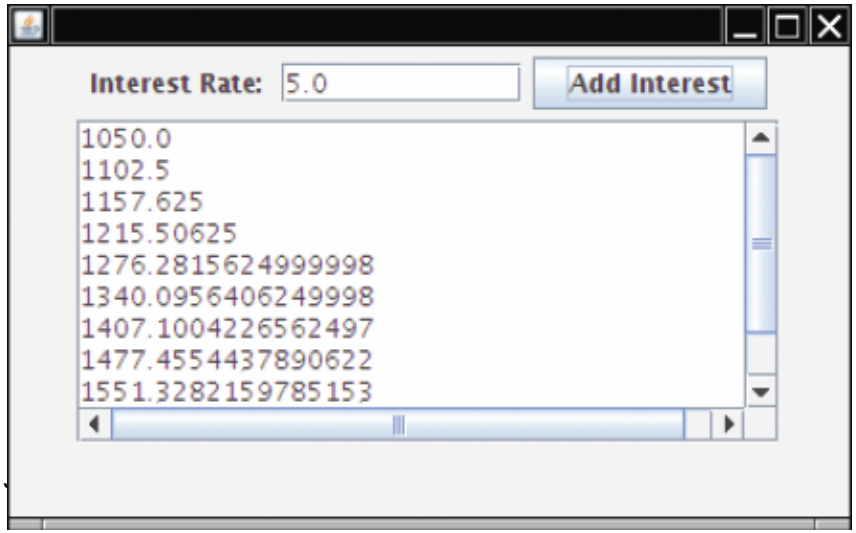
JTextField and JTextArea

- The append method is declared in the JTextArea class

```
textArea.append(balance + "\n");
```

- To add scroll bars to a JTextArea , use JScrollPane:

```
JScrollPane scrollPane = new JScrollPane(textArea);
```



It keeps track of all the bank balance, not just the last one

InvestmentFrame3.java (1)

```
/**
   A frame that shows the growth of an investment with variable interest,
   using a text area.
*/
public class InvestmentFrame3 extends JFrame
{
   private static final int FRAME_WIDTH = 400;
   private static final int FRAME_HEIGHT = 250;

   private static final int AREA_ROWS = 10;
   private static final int AREA_COLUMNS = 30;

   private static final double DEFAULT_RATE = 5;
   private static final double INITIAL_BALANCE = 1000;

   private JLabel rateLabel;
   private JTextField rateField;
   private JButton button;
   private JTextArea resultArea;
   private double balance;

```

Declare the components to be used

InvestmentFrame.java (2)

```
32      public InvestmentFrame3()
33      {
34         balance = INITIAL_BALANCE;
35         resultArea = new JTextArea(AREA_ROWS, AREA_COLUMNS);
36         resultArea.setText(balance + "\n");
37         resultArea.setEditable(false);
38
39         createTextField();
40         createButton();
41         createPanel();
42
43         setSize(FRAME_WIDTH, FRAME_HEIGHT);
44      }
45
46      private void createTextField()
47      {
48         rateLabel = new JLabel("Interest Rate: ");
49
50         final int FIELD_WIDTH = 10;
51         rateField = new JTextField(FIELD_WIDTH);
52         rateField.setText("" + DEFAULT_RATE);
53      }
```

Constructor calls methods to create the components

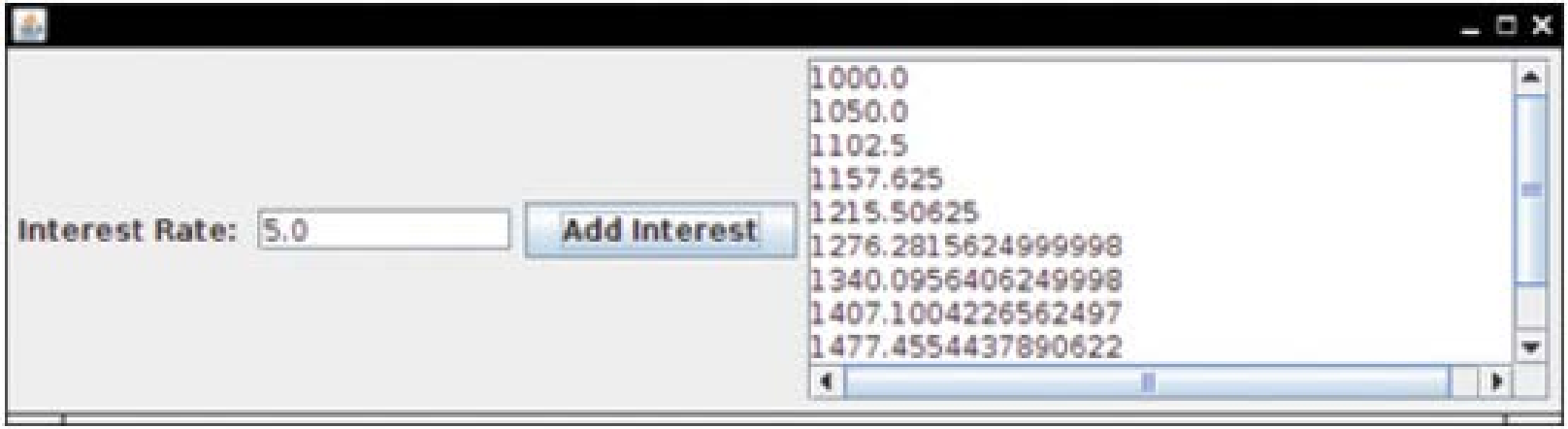
InvestmentFrame.java (3)

```
54
55      class AddInterestListener implements ActionListener
56      {
57         public void actionPerformed(ActionEvent event)
58         {
59            double rate = Double.parseDouble(rateField.getText());
60            double interest = balance * rate / 100;
61            balance = balance + interest;
62            resultArea.append(balance + "\n");
63         }
64      }
65
66      private void createButton()
67      {
68         button = new JButton("Add Interest");
69
70         ActionListener listener = new AddInterestListener();
71         button.addActionListener(listener);
72      }
```

The listener class and associated `createButton` method

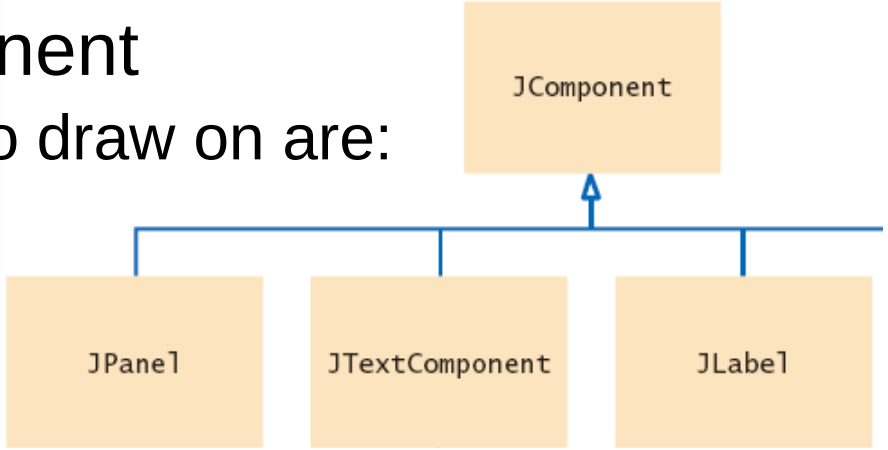
InvestmentFrame.java (4)

```
   private void createPanel()
   {
      JPanel = new JPanel();
      panel.add(rateLabel);
      panel.add(rateField);
      panel.add(button);
      JScrollPane scrollPane = new JScrollPane(resultArea);
      panel.add(scrollPane);
      add(panel);
   }
}
```



10.4 Creating Drawings

- You cannot draw directly on a `JFrame` object
- Instead, construct an object and add it to the frame, then draw on the component
  - A few examples objects to draw on are:
    - `JComponent`
    - `Jpanel`
    - `JTextComponent`
    - `JLabel`



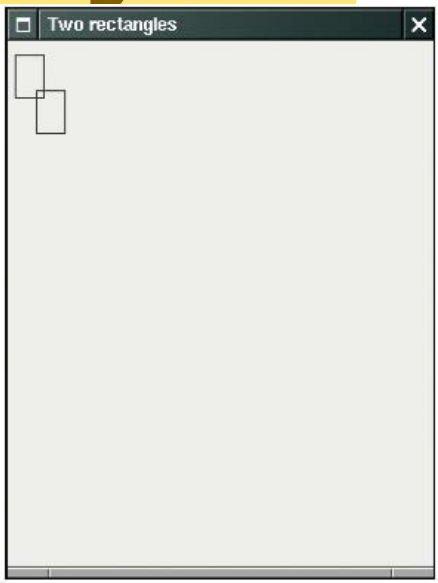
Extend the `JComponent` Class and override its `paintComponent` method

```
public class chartComponent extends JComponent
{
  public void paintComponent(Graphics g)
  {
    // Drawing instructions go here
  }
}
```

The method receives an object of type `Graphics` which store graphics state – the current color, font, etc.

RectangleComponent.java

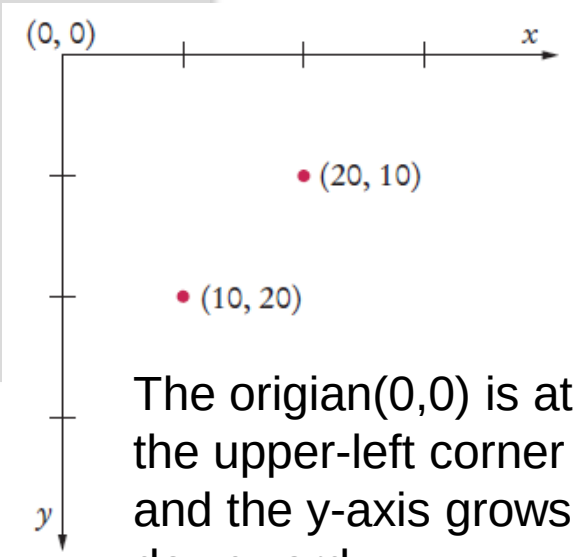
```
1   import java.awt.Graphics;
2   import java.awt.Graphics2D;
3   import java.awt.Rectangle;
4   import javax.swing.JComponent;
6   /**
7       A component that draws two rectangles.
8   */
9   public class RectangleComponent extends JComponent
10  {
11      public void paintComponent(Graphics g)
12      {
13          // Recover Graphics2D
14          Graphics2D g2 = (Graphics2D) g;
15
16          // Construct a rectangle and draw it
17          Rectangle box = new Rectangle(5, 10, 20, 30);
18          g2.draw(box);
20          // Move rectangle 15 units to the right and 25 units down
21          box.translate(15, 25);
22
23          // Draw moved rectangle
24          g2.draw(box);
25      }
26  }
```



The `paintComponent` method

- The `paintComponent` method is called automatically when:
  - The component is **shown** for the **first time**
  - Every time the window is **resized**, or after being **hidden**

```
public class chartComponent extends JComponent
{
  public void paintComponent(Graphics g)
  {
    g.fillRect(0, 10, 200, 10);
    g.fillRect(0, 30, 300, 10);
    g.fillRect(0, 50, 100, 10);
  }
}
```



The origian(0,0) is at the upper-left corner and the y-axis grows downward

ChartComponent.java

```
import java.awt.Graphics;
import javax.swing.JComponent;

/**
   A component that draws a bar chart.
*/
public class ChartComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      g.fillRect(0, 10, 200, 10);
      g.fillRect(0, 30, 300, 10);
      g.fillRect(0, 50, 100, 10);
   }
}
```

The **`Graphics`** class is part of the **`java.awt`** package

- We now have a `JComponent` object that can be added to a `Jframe`

ChartViewer.java

```
import javax.swing.JComponent;
import javax.swing.JFrame;

public class ChartViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      frame.setSize(400, 200);
      frame.setTitle("A bar chart");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JComponent component = new ChartComponent();
      frame.add(component);

      frame.setVisible(true);
   }
}
```

Adding the component to the frame

The Graphics parameter

- The paintComponent method receives an object of type Graphics
  - The Graphics object stores the graphics state
    - The current color, font, etc., that are used for drawing operations
  - The Graphics2D class extends the Graphics class
    - Provides more powerful methods to draw 2D objects
    - When using Swing, the Graphics parameter is actually of the Graphics2D type, so we need to cast it to Graphics2D to use it

```
public class RectangleComponent extends JComponent
{
  public void paintComponent(Graphics g)
  {
    Graphics2D g2 = (Graphics2D) g;
  }
}
```

Now you are ready to draw more complex shapes!

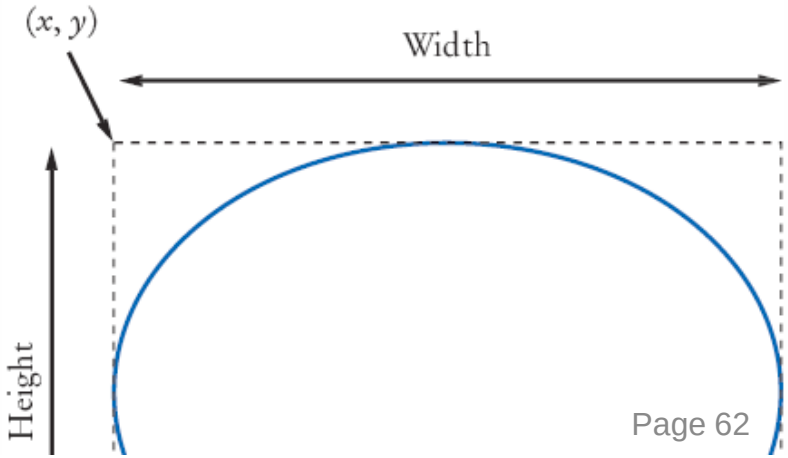
Ovals, Lines, Text, and Color

- Ellipses/oval are drawn inside a *bounding box* in the same way that you specify a rectangle:
  - Provide the *x* and *y coordinates* of the *top-left corner*
  - Provide the *width* and *height* of the bounding box
  - Use the `Graphics` class `drawOval` method to create an ellipse `g.drawOval(x, y, width, height);`
  - drawLine between two points: `g.drawLine(x1, y1, x1, y2);`

(x1, y1) is the start point of the line, and (x2, y2) is the end point of the line.

  - `fillOval` method to fill the inside

    fillOval(int x, int y, int width, int height)
  - `drawRect` method to outline the rectangle



Drawing Text

- Use the `drawString` method of the `Graphics` class to draw a string anywhere in a window
  - Specify the String
  - Specify the Basepoint (x and y coordinates of the first character)
  - The Baseline is the y coordinate of the Basepoint

```
g2.drawString("Message", 50, 100);
```

  - Use `drawLIne` method to draw a line

```
g.drawLine(x1, y1, x1, y2);
```

Using Color

- All shapes and strings are drawn with a *black pen* and white fill by default
- To change the color, call `setColor` with an object of type `Color`
  - Java uses the RGB color model
  - You can use predefined colors, or create your own

```
g.setColor(Color.YELLOW);
g.fillOval(350, 25, 35, 20);
```

All shapes drawn after `setColor` will use it

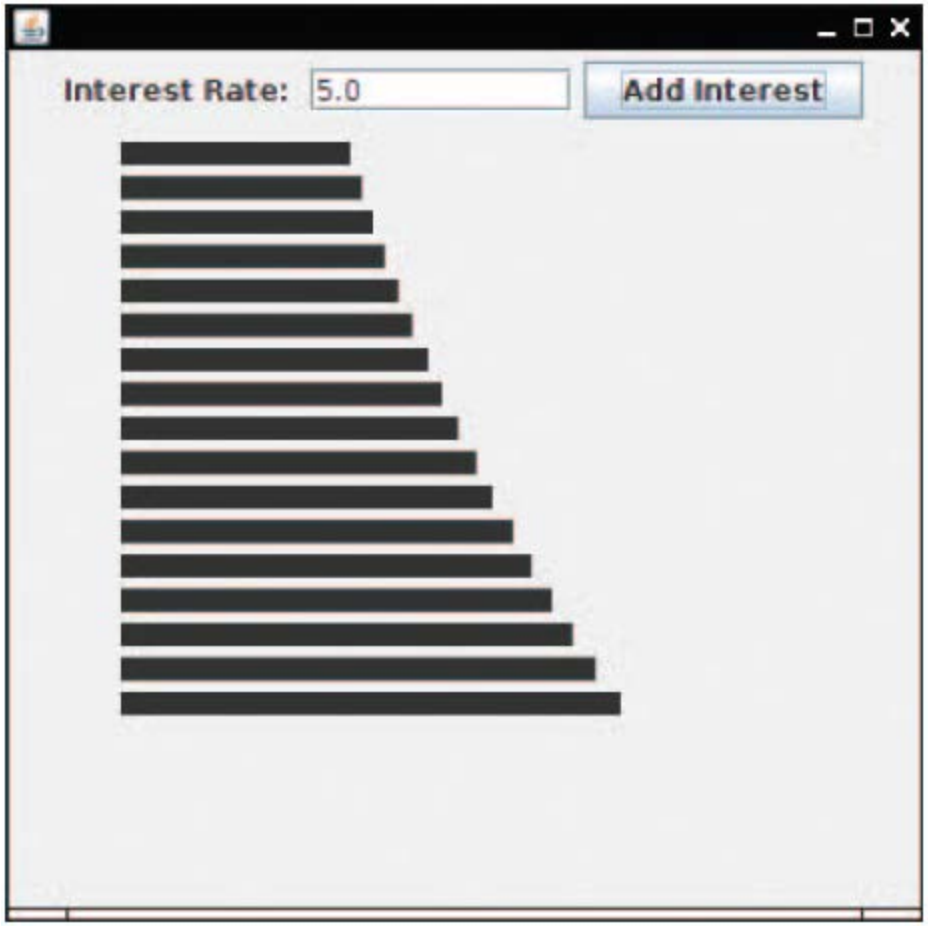
Color		RGB Value
Color.BLACK		0, 0, 0
Color.BLUE		0, 0, 255
Color.CYAN		0, 255, 255
Color.GRAY		128, 128, 128
Color.DARKGRAY		64, 64, 64
Color.LIGHTGRAY		192, 192, 192
Color.GREEN		0, 255, 0
Color.MAGENTA		255, 0, 255
Color.ORANGE		255, 200, 0
Color.PINK		255, 175, 175
Color.RED		255, 0, 0
Color.WHITE		255, 255, 255
Color.YELLOW		255, 255, 0

ChartComponent2.java

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JComponent;

/**
   A component that draws a demo chart.
*/
public class ChartComponent2 extends JComponent
{
   public void paintComponent(Graphics g)
   {
      // Draw the bars
      g.fillRect(0, 10, 200, 10);
      g.fillRect(0, 30, 300, 10);
      g.fillRect(0, 50, 100, 10);

      // Draw the arrow
      g.drawLine(350, 35, 305, 35);
      g.drawLine(305, 35, 310, 30);
      g.drawLine(305, 35, 310, 40);

      // Draw the highlight and the text
      g.setColor(Color.YELLOW);
      g.fillOval(350, 25, 35, 20);
      g.setColor(Color.BLACK);
      g.drawString("Best", 355, 40);
   }
}
```

Application: Investment Growth

- Input the interest rate
- Click on the Add Interest button to add bars to the graph
- Maintains a list of values to redraw all bars each time



ChartComponent.java(1)

```
/**
   A component that draws a chart.
*/
public class ChartComponent extends JComponent
{
   private ArrayList<Double> values;
   private double maxValue;

   public ChartComponent(double max)
   {
      values = new ArrayList<Double>();
      maxValue = max;
   }

   public void append(double value)
   {
      values.add(value);
      repaint();
   }
```

Use an ArrayList to hold bar values

know the largest value that should still fit inside the chart

Add a new value to Arraylist

The call to repaint forces a call to the paintComponent method.

ChartComponent.java(2)

```
26      public void paintComponent(Graphics g)
27      {
28         final int GAP = 5;
29         final int BAR_HEIGHT = 10;
30
31         int y = GAP;
32         for (double value : values)
33         {
34            int barWidth = (int) (getWidth() * value / maxValue);
35            g.fillRect(0, y, barWidth, BAR_HEIGHT);
36            y = y + BAR_HEIGHT + GAP;
37         }
38      }
39   }
```

Paint bars in a loop

Draw a bar proportional to the interest rate, but cannot actually draw a bar that is 10,050 pixel long!
Need scale!

The getWidth method returns the width of the component in pixels. If the value to be drawn equals maxValue, the bar stretches across the entire component width.

InvestmentFrame4.java (1)

```
/**
   A frame that shows the growth of an investment with variable interest,
   using a bar chart.
*/
public class InvestmentFrame4 extends JFrame
{
   public InvestmentFrame4()
   {
      balance = INITIAL_BALANCE;
      chart = new ChartComponent(3 * INITIAL_BALANCE);
      chart.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
      chart.append(INITIAL_BALANCE);

      createTextField();
      createButton();
      createPanel();

      setSize(FRAME_WIDTH, FRAME_HEIGHT);
   }

   private void createTextField()
   {
      rateLabel = new JLabel("Interest Rate: ");

      final int FIELD_WIDTH = 10;
      rateField = new JTextField(FIELD_WIDTH);
      rateField.setText("" + DEFAULT_RATE);
   }
```

Instantiates and initializes ChartComponent

Use helper methods to create components

InvestmentFrame4.java (2)

```
   class AddInterestListener implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         double rate = Double.parseDouble(rateField.getText());
         double interest = balance * rate / 100;
         balance = balance + interest;
         chart.append(balance);
      }
   }
```

```
   private void createButton()
   {
      button = new JButton("Add Interest");

      ActionListener listener = new AddInterestListener();
      button.addActionListener(listener);
   }

   private void createPanel()
   {
      JPanel panel = new JPanel();
      panel.add(rateLabel);
      panel.add(rateField);
      panel.add(button);
      panel.add(chart);
      add(panel);
   }
}
```

Listener and Button setup

- The call to repaint forces a call to the *paintComponent* method. The *paintComponent* method redraws the component. Then the graph is drawn again, now showing the appended value.
  - Why not call *paintComponent* directly?
  - The simple answer is that you can't—you don't have a Graphics object that you can pass as an argument. Instead, you need to ask the Swing library to make the call to *paintComponent* at its earliest convenience.
    That is what the repaint method does.

Common Error 10.3

- Forgetting to Repaint
  - When you change the data in a painted component, the component is NOT automatically painted with the new data
    - Note: Only when you make a change to a standard Swing component (e.g. JLabel) that will automatically repaint
  - You must call the `repaint` method of the component
    - Not call the `paintComponent` method directly
  - The best place to call `repaint` is in the method of your component that modifies the data values:

```
void changeData(. . .)
{
  // Update data values
  repaint();
}
```

Common Error 10.4

- By default, Components have zero width and height

  If you place a painted component into a panel, you need to specify its preferred size

  - You must be careful when you add a painted component, such as a component displaying a chart, to a panel

    similar to specifying the number of rows and columns in a text

  - The default size for a `JComponent` is 0 by 0 pixels, and the component will not be visible.
  - The remedy is to call the `setPreferredSize` method:

```
chart.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
```

Call the setPreferredSize method with a Dimension object as argument.
A Dimension argument wraps a width and a height into a single object.

Steps to Drawing Shapes

1) Determine the shapes you need for your drawing.
   - Squares and rectangles
   - Circles and ellipses
   - Lines and Text
2) Find the coordinates of each shape.
   - For rectangles and ellipses, you need the top-left corner, width, and height of the bounding box.
   - For lines, you need the *x*- and *y*-positions of the starting point and the end point.
   - For text, you need the *x*- and *y*-position of the basepoint

Steps to Drawing Shapes

3) Write Java statements to draw the shapes.

```
g.setColor(Color.GREEN);
g.fillRect(100, 100, 30, 60);

g.setColor(Color.RED);
g.fillRect(160, 100, 30, 60);

g.setColor(Color.BLACK);
g.drawLine(130, 100, 160, 100);
g.drawLine(130, 160, 160, 160);
```

- If possible, use variables and ‘offsets’ for the locations and sizes

```
g.fillRect(xLeft, yTop, width / 3, width * 2 / 3);
. . .
g.fillRect(xLeft + 2 * width / 3, yTop, width / 3, width * 2 / 3);
. . .
g.drawLine(xLeft + width / 3, yTop, xLeft + width * 2 / 3, yTop);
```

Steps to Drawing Shapes

4) Consider using methods or classes for repetitive steps.

need to draw more than one flag? Perhaps with different sizes?

```
void drawItalianFlag(Graphics g, int xLeft, int yTop, int width)
{
  // Draw a flag at the given location and size
}
```

```
drawItalianFlag(g, 10, 10, 100);
drawItalianFlag(g, 10, 125, 150);
```

Steps to Drawing Shapes

5) Place the drawing instructions in the paintComponent method.

```
public class ItalianFlagComponent extends JComponent
{
  public void paintComponent(Graphics g)
  {
    // Drawing instructions
  }
}
```

If your drawing is simple, simply place all drawing statements here.

If the drawing is complex, use call methods of Step 4

Steps to Drawing Shapes

6) Write the viewer class.

Provide a viewer class, with a main method in which you construct a frame, add your component, and make your frame visible.

```
public class ItalianFlagViewer
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      frame.setSize(300, 400);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JComponent component = new ItalianFlagComponent();
      frame.add(component);
      frame.setVisible(true);
   }
}
```

Summary: Frames and Components

- To show a frame, construct a `JFrame` object, set its size, and make it visible.
- Use a `JPanel` to group multiple user-interface components together.
- Declare a `JFrame` subclass for a complex frame.

Summary: Events and Handlers

- User-interface events include key presses, mouse moves, button clicks, menu selections, and so on.
- An event listener belongs to a class created by the application programmer.
  - Its methods describe the actions to be taken when an event occurs.
  - Event sources report on events. When an event occurs, the event source notifies all event listeners.
- Attach an ActionListener to each button so that your program can react to button clicks.
- Methods of an inner class can access variables from the surrounding class.

Summary: TextFields and TextAreas

- Use `JTextField` components to provide space for user input.
  - Place a `JLabel` next to each text field
- Use a `JTextArea` to show multiple lines of text
  - You can add scroll bars to any component with a `JScrollPane`
- You can add scroll bars to any component with a `JScrollPane`.

Summary: Simple Shapes

- In order to display a drawing, provide a class that extends the `JComponent` class.
- Place drawing instructions inside the `paintComponent` method.
  - That method is called whenever the component needs to be repainted.
- The `Graphics` class has methods to draw rectangles and other shapes.
  - Use `drawRect`, `drawOval`, and `drawLine` to draw geometric shapes.
  - The `drawString` method draws a string, starting at its basepoint.

Summary: Color and `repaint`

- When you set a new color in the graphics context, it is used for subsequent drawing operations.
- Call the repaint method whenever the state of a painted component changes.
- When placing a painted component into a panel, you need to specify its preferred size.