Fall, 2011
CISC181 Final Review

Prof. Christopher Rasmussen

Course page: http://goo.gl/jJ8HT

Administrative details I

- Deadline for course evaluations: Thursday, Dec. 8, midnight
- Written comments are most helpful...

Administrative details II

- Preliminary grades posted on course page
- Vincent should have sent each of you a random "ID code" to look yourself up...

Administrative details III

- You have one extra day to finish Project #3: the due date is now Wednesday, December 7 (at midnight)
- You can still use late days after that if you have them
- Don't forget Lab #8 -- due at the usual time for your section this week

Final Details

- Next Thursday, December 15
- Closed book, no notes, no calculators, cell phones, etc.
- Worth 15% of your grade (same as midterm)
- Covers all lectures from Tuesday, October 25 through Tuesday, November 29 class
  - Pay close attention to exact pages in readings
  - Topics in the textbook: Assertions/exceptions, Swing
  - Topics totally outside the textbook: Unit testing, Android
  - Will NOT cover anything about Eclipse or its built-in debugging facilities (including Android logging)
  - STUDY SAMPLE PROGRAMS WE WENT OVER IN CLASS!
- Question types
  - Language, API feature/concept definitions and explanations
  - Write a function that does X or a whole class with certain variables and methods
  - If we call method f() with arguments a and b, what does it return/print/do?

Topics Covered

- Exception handling, assertions
- Unit testing (separate slides #1)
- Deployment
  - JARs (separate slides #2)
  - Applets: no main(), derive your class from JApple, put set-up code in init()
- Swing (separate slides #3)
  - Windows, button/mouse events, listeners
  - 2-D drawing
  - Layout managers, swapping panels
  - Timers, animation
- Android
  - Activities: concepts, starting, communicating between
  - Views, resources, layouts
  - 2-D graphics, animation
  - Storing preferences, reading text file
  - Sound, text-to-speech, vibration, accelerometer

try-throw-catch Mechanism

- A **throw** statement is similar to a method call:

  throw new *ExceptionClassName*(*SomeString*);

  - In the above example, the object of class ***ExceptionClassName*** is created using a string as its argument
  - This object, which is an argument to the **throw** operator, is the exception object thrown
- Instead of calling a method, a **throw** statement calls a **catch** block

Defining Exception Classes

- A **throw** statement can throw an exception object of any exception class
- Instead of using a predefined class, exception classes can be programmer-defined
  - These can be tailored to carry the precise kinds of information needed in the **catch** block
  - A different type of exception can be defined to identify each different exceptional situation

Multiple **catch** Blocks

- A **try** block can potentially throw any number of exception values, and they can be of differing types
  - In any one execution of a **try** block, at most one exception can be thrown (since a throw statement ends the execution of the **try** block)
  - However, different types of exception values can be thrown on different executions of the **try** block

Multiple **catch** Blocks

- Each **catch** block can only catch values of the exception class type given in the **catch** block heading
- Different types of exceptions can be caught by placing more than one **catch** block after a **try** block
  - Any number of **catch** blocks can be included, but they must be placed in the correct order

Pitfall: Catch the More Specific Exception First

- When catching multiple exceptions, the order of the **catch** blocks is important
  - When an exception is thrown in a **try** block, the **catch** blocks are examined in order
  - The first one that matches the type of the exception thrown is the one that is executed

The **finally** Block

- The **finally** block contains code to be executed whether or not an exception is thrown in a **try** block
  - If it is used, a **finally** block is placed after a **try** block and its following **catch** blocks

```
try
{ . . . }
catch(ExceptionClass1 e)
{ . . . }
. . .
catch(ExceptionClassN e)
{ . . . }
finally
{
CodeToBeExecutedInAllCases
}
```

The **finally** Block

- If the **try-catch-finally** blocks are inside a method definition, there are three possibilities when the code is run:
  1. The **try** block runs to the end, no exception is thrown, and the finally block is executed
  2. An exception is thrown in the **try** block, caught in one of the **catch** blocks, and the **finally** block is executed
  3. An exception is thrown in the **try** block, there is no matching **catch** block in the method, the **finally** block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

The Catch or Declare Rule

- Most ordinary exceptions that might be thrown within a method must be accounted for in one of two ways:
  1. The code that can throw an exception is placed within a **try** block, and the possible exception is caught in a **catch** block within the same method
  2. The possible exception can be declared at the start of the method definition by placing the exception class name in a **throws** clause

When to Use Exceptions

- Exceptions should be reserved for situations where a method encounters *an unusual or unexpected case that cannot be handled easily in some other way*
- When exception handling must be used, here are some basic guidelines:
  - Include **throw** statements and list the exception classes in a **throws** clause within a method definition
  - Place the **try** and **catch** blocks in a different method

Assertion Checks

- An *assertion* is a sentence that says (asserts) something about the state of a program
  - An assertion must be either true or false, and should be true if a program is working properly
  - Assertions can be placed in a program as comments
- Java has a statement that can check if an assertion is true

  assert Boolean_Expression;
  - If assertion checking is turned on and the **Boolean_Expression** evaluates to **false**, the program ends, and outputs an *assertion failed error message*
  - Otherwise, the program finishes execution normally

Assertion Checks

- A program or other class containing assertions is compiled in the usual way
- After compilation, a program can run with assertion checking turned on or turned off
  - Normally a program runs with assertion checking turned off
- In order to run a program with assertion checking turned on, use the following command (using the actual **ProgramName**):

  java –enableassertions ProgramName

Miscellaneous + Swing

- Unit testing
  - Web slide show: **http://www.slideshare.net/tom.zimmermann/unit-testing-with-junit**
  - Sample code: PokerTest.java (for CardGame)
- JARs:
  - Separate slides in **java_jars.ppt**
- Applets: Java apps embedded in web pages
  - No main()
  - Derive your class from JApplet
  - Put set-up code in init()
  - Create JAR
  - Link JAR in web page
  - Sample code: HelloApplet.java, HelloApplet.html; DragMouseApplet.java
- Swing: Separate slides

JAR Files (yousa likey!)

- **JAR**: **J**ava **AR**chive. A group of Java classes and supporting files combined into a single file compressed with ZIP format, and given .JAR extension.

- Advantages of JAR files:
  - compressed; quicker download
  - just one file; less mess
  - can be executable

- The closest you can get to having a .exe file for your Java application.

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

Creating a JAR archive

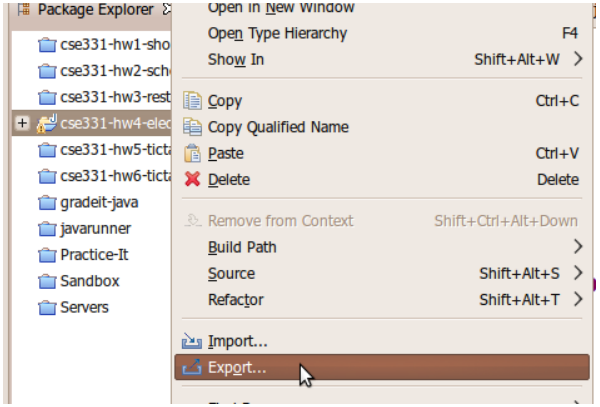
- from the command line:

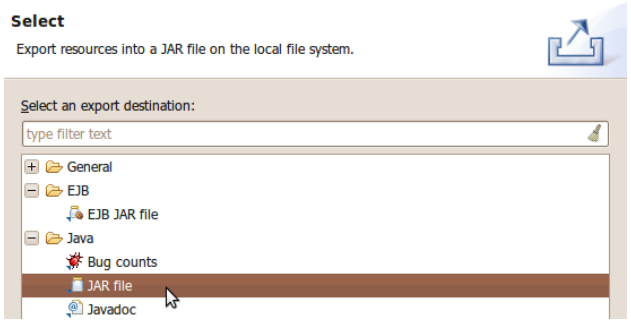
  `jar -cvf` **`filename`**`.jar` **`files`**

  - Example:

    `jar -cvf MyProgram.jar *.class *.gif *.jpg`

- some IDEs (e.g. Eclipse) can create JARs automatically
  - File → Export... → JAR file

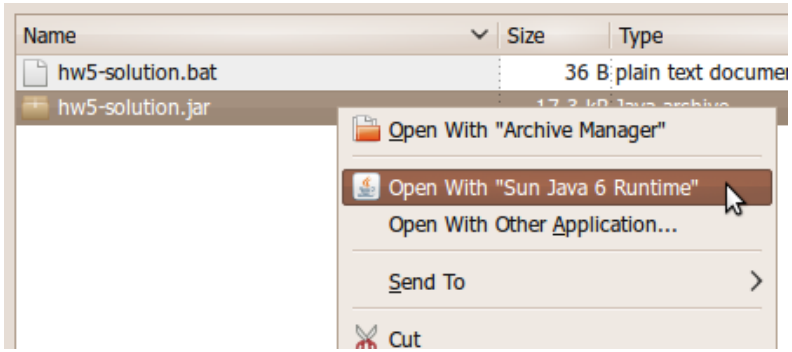




Running a JAR

- Running a JAR from the command line:
  - `java -jar` **`filename`**`.jar`

- Most OSes can run JARs directly by double-clicking them:



Making a runnable JAR

- **manifest file**: Used to create a JAR runnable as a program.

```
jar -cvmf manifestFile MyAppletJar.jar
          mypackage/*.class *.gif
```

Contents of MANIFEST file:

```
Main-Class: MainClassName
```

  - Eclipse will automatically generate and insert a proper manifest file into your JAR if you specify the main-class to use.

Resources inside a JAR

- You can embed external resources inside your JAR:
  - images (GIF, JPG, PNG, etc.)
  - audio files (WAV, MP3)
  - input data files (TXT, DAT, etc.)
  - ...

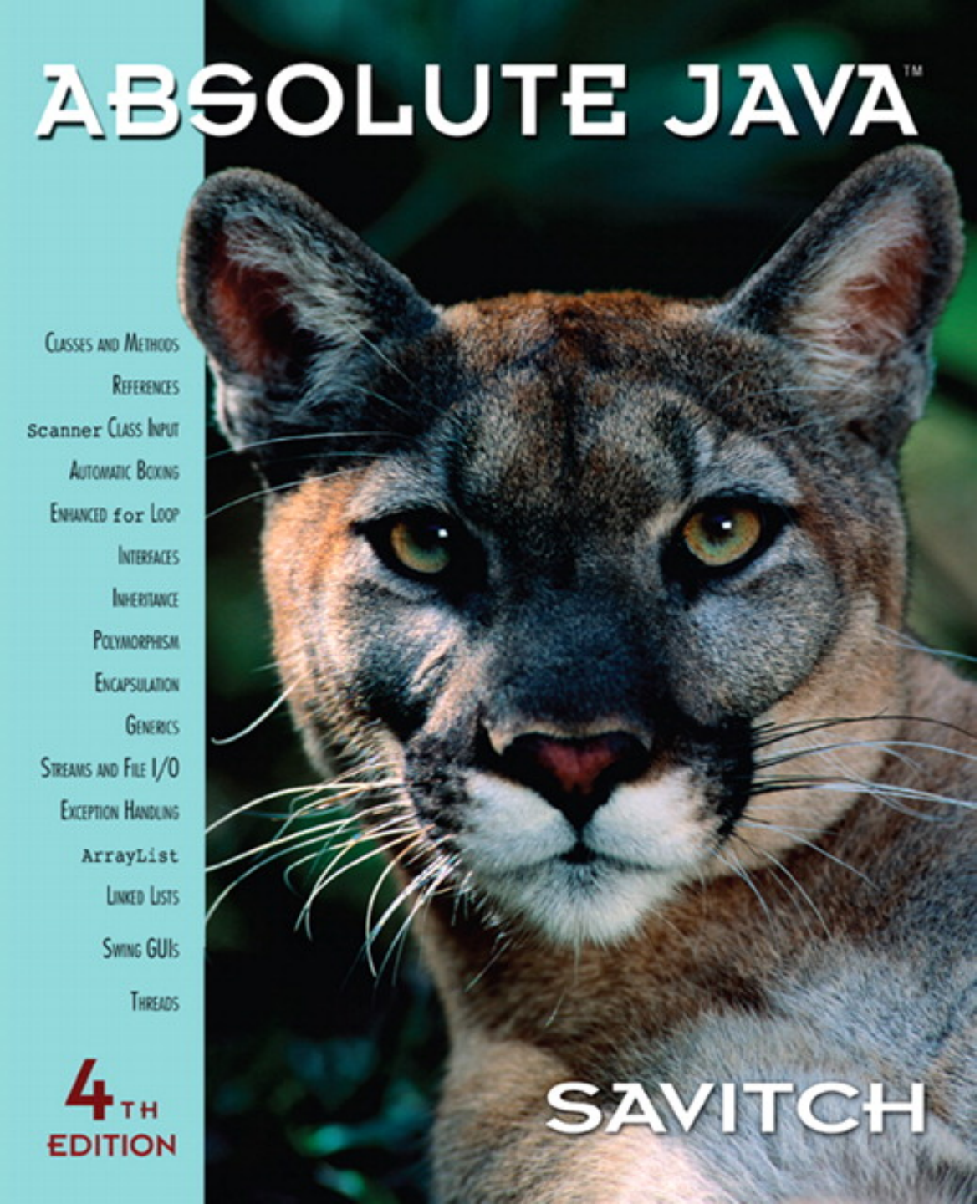
- But code for opening files will look outside your JAR, not inside it.
  - `Scanner in = new Scanner(new File("data.txt"));  // fail`
  - `ImageIcon icon = new ImageIcon("pony.png");  // fail`
  - `Toolkit.getDefaultToolkit().getImage("cat.jpg");  // fail`

Accessing JAR resources

- Every class has an associated .class object with these methods:
  - `public URL getResource(String filename)`
  - `public InputStream getResourceAsStream(String name)`

- If a class named `Example` wants to load resources from within a JAR, its code to do so should be the following:
  - ```
    Scanner in = new Scanner(
        Example.class.getResourceAsStream("/data.txt"));
    ```
  - ```
    ImageIcon icon = new ImageIcon(
        Example.class.getResource("/pony.png"));
    ```
  - ```
    Toolkit.getDefaultToolkit().getImage(
        Example.class.getResource("/images/cat.jpg"));
    ```

  - (Some classes like Scanner read from streams; some like Toolkit read from URLs.)
  - NOTE the very important leading `/` character; without it, you will get a `null` result
    ```



Chapter 17

Swing I

PEARSON Addison Wesley

Introduction to Swing

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

Events

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

Listeners

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

Exception Objects

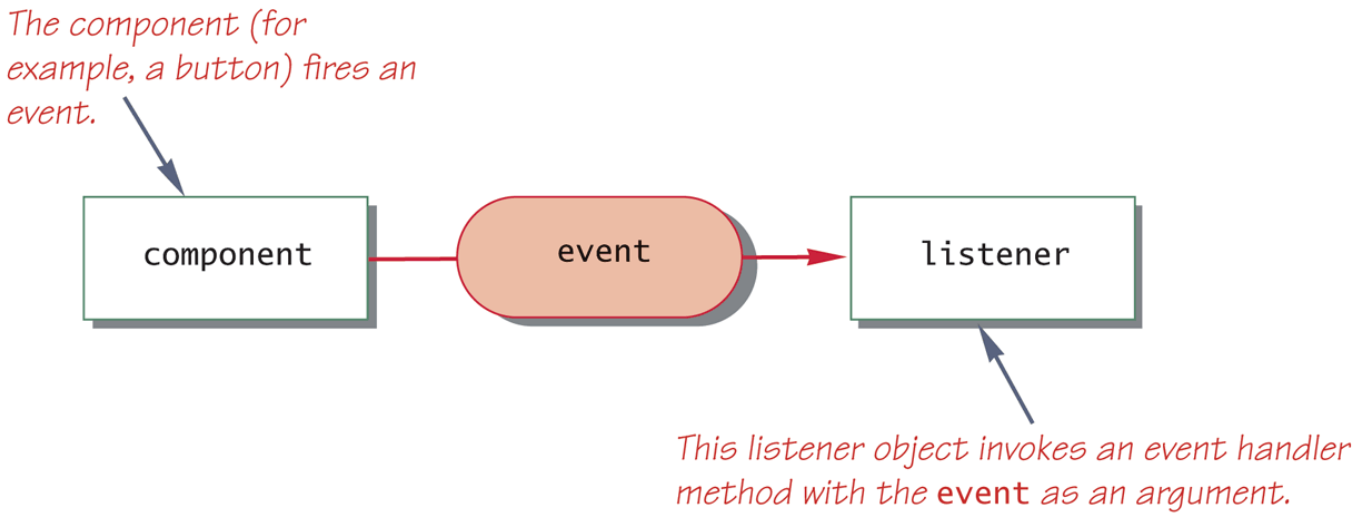
- An exception object is an event
  - The throwing of an exception is an example of firing an event
- The listener for an exception object is the **`catch`** block that catches the event

Event Handlers

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

Event Firing and an Event Listener

Display 17.1 Event Firing and an Event Listener



Event-Driven Programming

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

Event-Driven Programming

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

Event-Driven Programming

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

A Simple Window

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

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

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

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

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

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

A First Swing Demonstration (Part 1 of 4)

Display 17.2 A First Swing Demonstration Program

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

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

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

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

(continued)

A First Swing Demonstration (Part 2 of 4)

Display 17.2 A First Swing Demonstration Program

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

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

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

This is the file `FirstSwingDemo.java`.

(continued)

A First Swing Demonstration (Part 3 of 4)

Display 17.2 A First Swing Demonstration Program

```
1  import java.awt.event.ActionListener;
2  import java.awt.event.ActionEvent;    This is the file EndingListener.java.

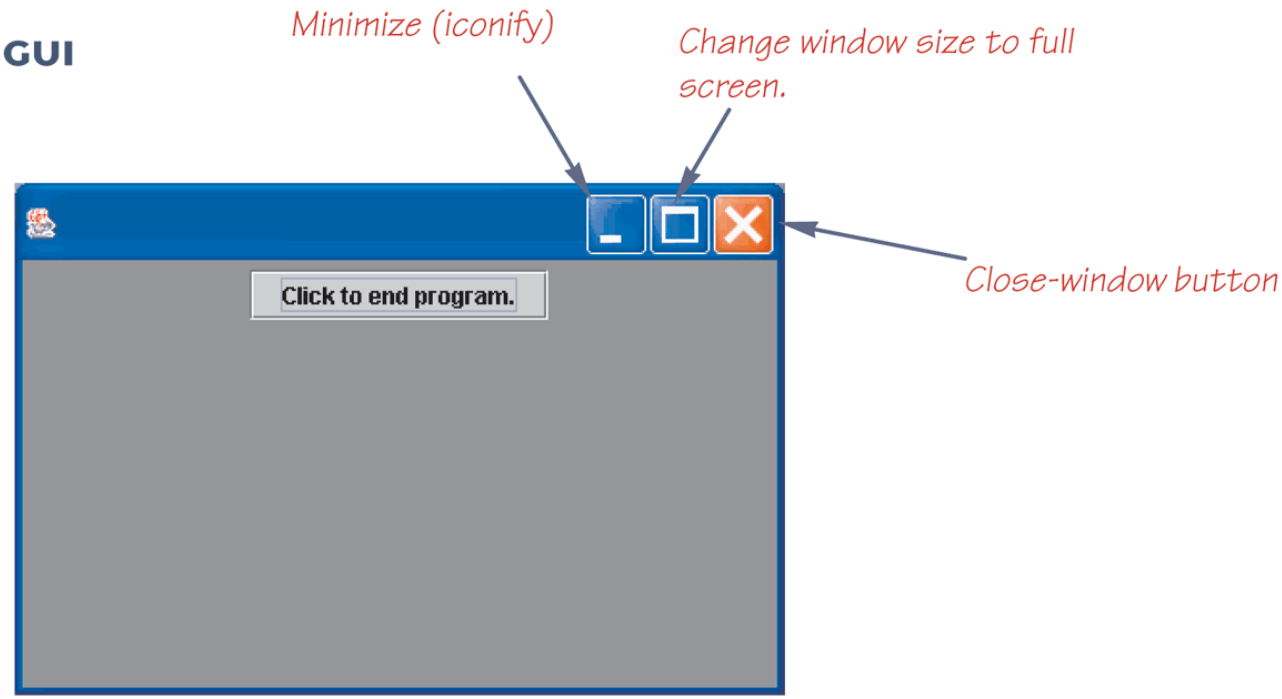
3  public class EndingListener implements ActionListener
4  {
5      public void actionPerformed(ActionEvent e)
6      {
7          System.exit(0);
8      }
9  }
```

(continued)

A First Swing Demonstration (Part 4 of 4)

Display 17.2 A First Swing Demonstration Program

RESULTING GUI



Some Methods in the Class `JFrame` (Part 1 of 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 in the Class `JFrame` (Part 2 of 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 in the Class **JFrame** (Part 3 of 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.)

Pitfall: Forgetting to Program the Close-Window Button

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

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

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

Buttons

- A *button* object is created from the class **JButton** and can be added to a **JFrame**
  - The argument to the **JButton** constructor is the string that appears on the button when it is displayed

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

Action Listeners and Action Events

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

Action Listeners and Action Events

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

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

Action Listeners and Action Events

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

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

Action Listeners and Action Events

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

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

Pitfall: Changing the Heading for `actionPerformed`

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

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

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

Tip: Ending a Swing Program

- GUI programs are often based on a kind of infinite loop
  - The windowing system normally stays on the screen until the user indicates that it should go away
- 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

A Better Version of Our First Swing GUI

- A better version of **`FirstWindow`** makes it a derived class of the class **`JFrame`**
  - This is the normal way to define a windowing interface
- The constructor in the new **`FirstWindow`** class starts by calling the constructor for the parent class using **`super();`**
  - This ensures that any initialization that is normally done for all objects of type **`JFrame`** will be done
- Almost all initialization for the window **`FirstWindow`** is placed in the constructor for the class
- Note that this time, an anonymous object is used as the action listener for the **`endButton`**

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

Display 17.4 The Normal Way to Define a `JFrame`

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


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

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

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

(continued)

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

Display 17.4 The Normal Way to Define a `JFrame`

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

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

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

This is the file `FirstWindow.java`.

(continued)

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

Display 17.4 The Normal Way to Define a `JFrame`

This is the file `DemoWindow.java`.

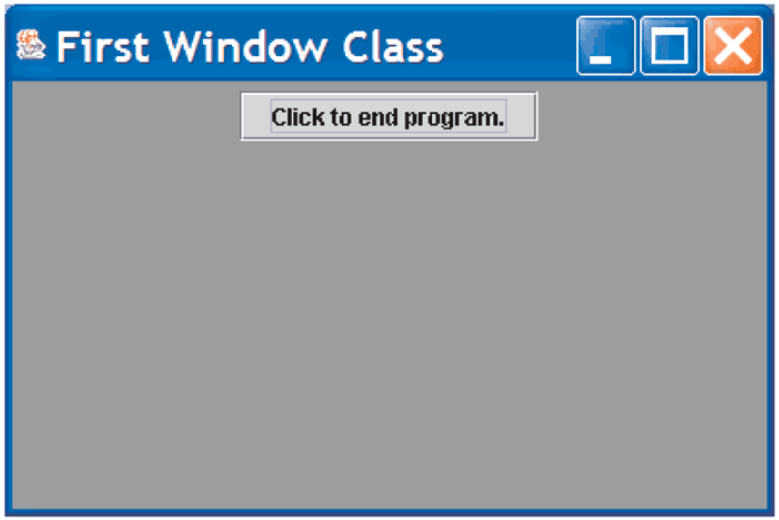
```
1  public class DemoWindow
2  {
3      public static void main(String[] args)
4      {
5          FirstWindow w = new FirstWindow();
6          w.setVisible(true);
7      }
8  }
```

(continued)

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

Display 17.4 The Normal Way to Define a `JFrame`

RESULTING GUI



Labels

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

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

Color

- In Java, a *color* is an object of the class **`Color`**
  - The class **`Color`** is found in the **`java.awt`** package
  - There are constants in the **`Color`** class that represent a number of basic colors
- A **`JFrame`** can not be colored directly
  - Instead, a program must color something called the *content pane* of the **`JFrame`**
  - Since the content pane is the "inside" of a **`JFrame`**, coloring the content pane has the effect of coloring the inside of the **`JFrame`**
  - Therefore, the background color of a **`JFrame`** can be set using the following code:

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

The Color Constants

Display 17.5 The Color Constants

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

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

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

Display 17.6 A JFrame with Color

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

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

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

(continued)

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

Display 17.6 **A JFrame with Color**

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

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

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

This is an invocation of the other constructor.

This is the file `ColoredWindow.java`.

(continued)

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

Display 17.6 **A JFrame with Color**

This is the file `ColoredWindow.java`.

```
1  import java.awt.Color;

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

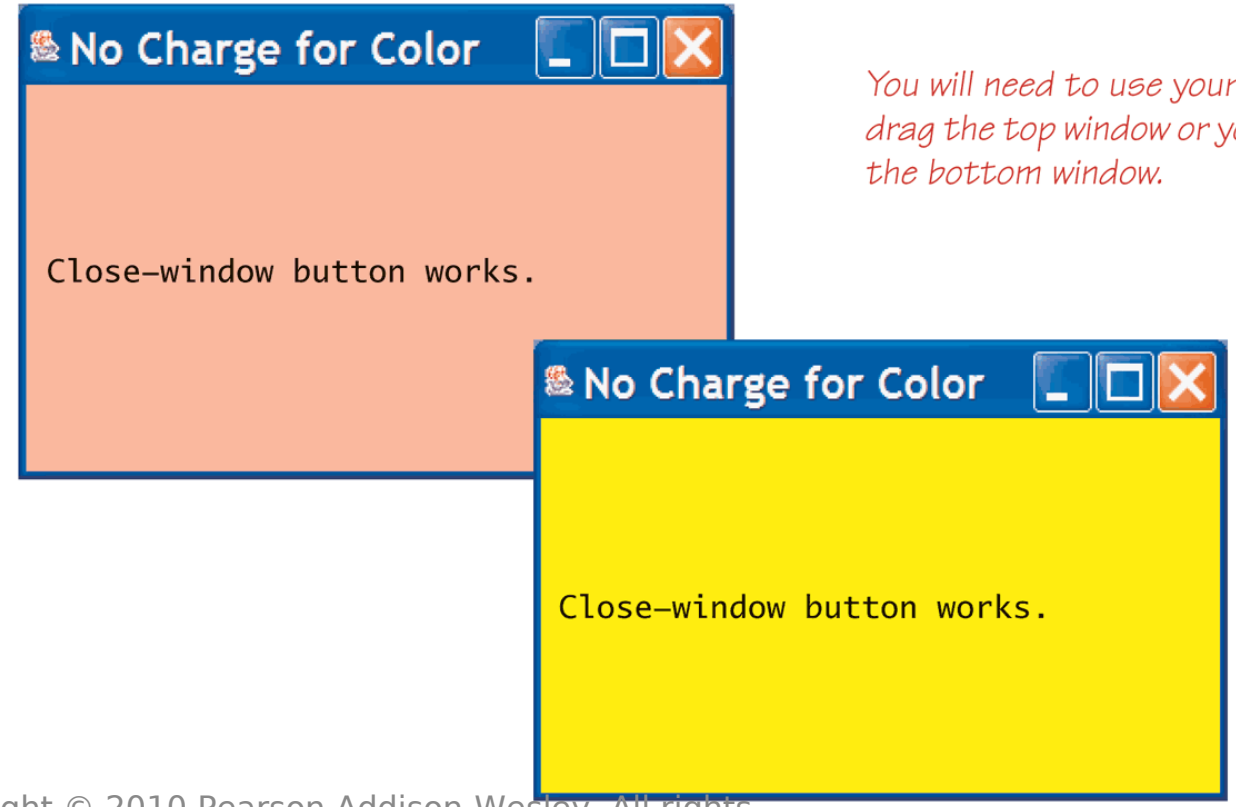
8          ColoredWindow w2 = new ColoredWindow(Color.YELLOW);
9          w2.setVisible(true);
10     }
11 }
```

(continued)

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

Display 17.6 **A JFrame with Color**

RESULTING GUI



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

Containers and Layout Managers

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

Border Layout Managers

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

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

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

Display 17.7 The BorderLayout Manager

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

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

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

(continued)

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

Display 17.7 The BorderLayout Manager

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

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

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

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

This is the file `BorderLayoutJFrame.java`.

(continued)

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

Display 17.7 The BorderLayout Manager

This is the file BorderLayoutDemo.java.

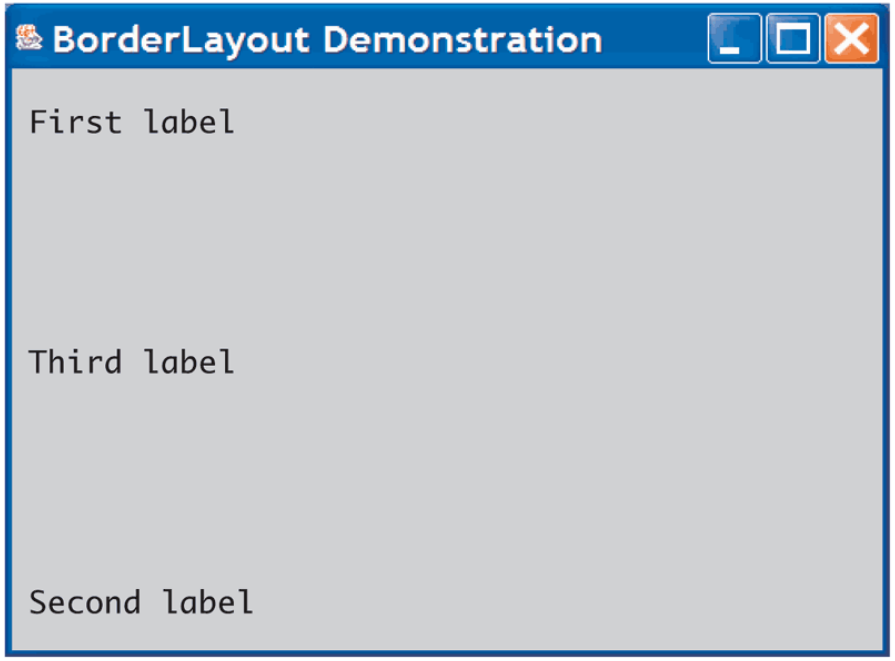
```
1  public class BorderLayoutDemo
2  {
3      public static void main(String[] args)
4      {
5          BorderLayoutJFrame gui = new BorderLayoutJFrame();
6          gui.setVisible(true);
7      }
8  }
```

(continued)

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

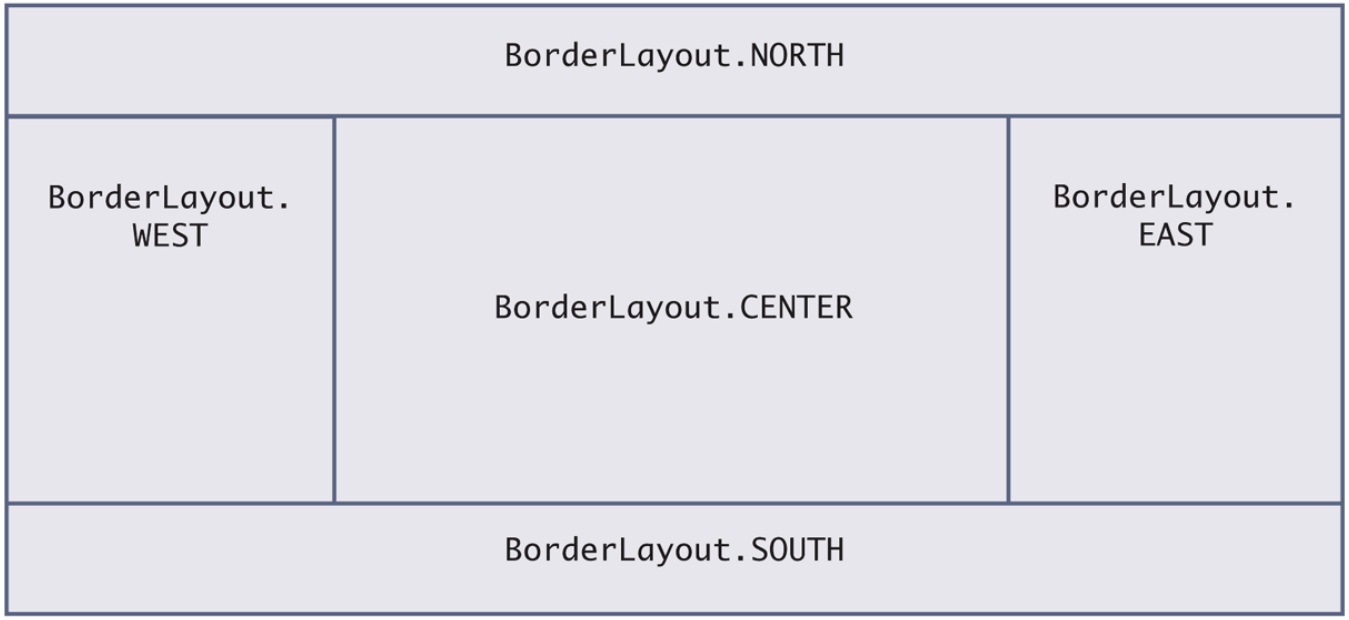
Display 17.7 **The BorderLayout Manager**

RESULTING GUI



BorderLayout Regions

Display 17.8 **BorderLayout Regions**



Border Layout Managers

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

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

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

Flow Layout Managers

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

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

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

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

Panels

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

Panels

- Both a **JFrame** and each panel in a **JFrame** can use different layout managers
  - Additional panels can be added to each panel, and each panel can have its own layout manager
  - This enables almost any kind of overall layout to be used in a GUI

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

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

Using Panels (Part 1 of 8)

Display 17.11 Using Panels

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

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

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

(continued)

Using Panels (Part 2 of 8)

Display 17.11 Using Panels

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

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

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

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

(continued)

Using Panels (Part 3 of 8)

Display 17.11 Using Panels

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

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

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

(continued)

Using Panels (Part 4 of 8)

Display 17.11 Using Panels

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

39          add(biggerPanel, BorderLayout.CENTER);

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

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

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

(continued)

Using Panels (Part 5 of 8)

Display 17.11 Using Panels

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

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

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

(continued)

Using Panels (Part 6 of 8)

Display 17.11 Using Panels

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

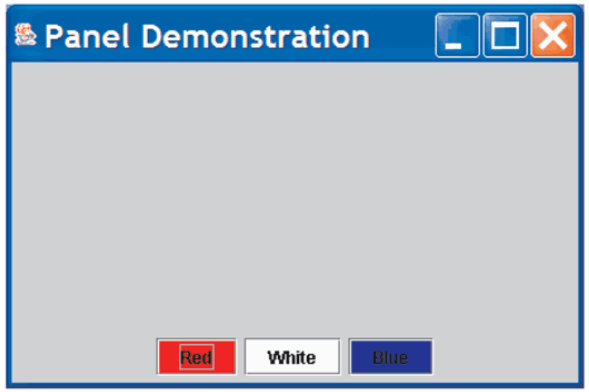
60            if (buttonString.equals("Red"))
61                redPanel.setBackground(Color.RED);
62            else if (buttonString.equals("White"))
63                whitePanel.setBackground(Color.WHITE);
64            else if (buttonString.equals("Blue"))
65                bluePanel.setBackground(Color.BLUE);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

(continued)

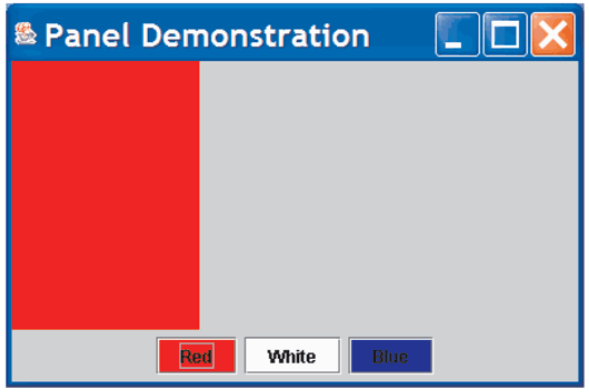
Using Panels (Part 7 of 8)

Display 17.11 Using Panels

RESULTING GUI (When first run)



RESULTING GUI (After clicking Red button)

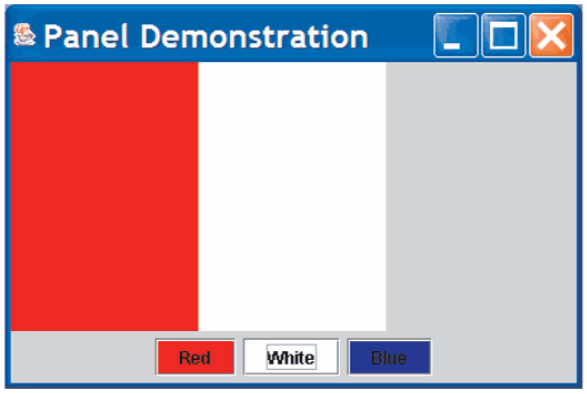


(continued)

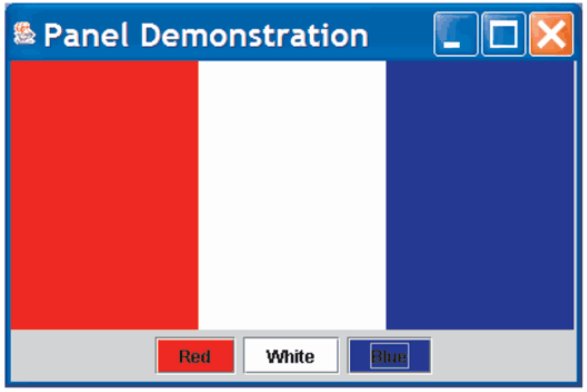
Using Panels (Part 8 of 8)

Display 17.11 Using Panels

RESULTING GUI (After clicking White button)



RESULTING GUI (After clicking Blue button)



Listeners as Inner Classes

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

Listeners as Inner Classes (Part 1 of 6)

Display 17.16 Listeners as Inner Classes

<Import statements are the same as in Display 17.14.>

```
1   public class InnerListenersDemo extends JFrame
2   {
3       public static final int WIDTH = 300;
4       public static final int HEIGHT = 200;

5       private JPanel redPanel;
6       private JPanel whitePanel;
7       private JPanel bluePanel;
```

(continued)

Listeners as Inner Classes (Part 2 of 6)

Display 17.16 Listeners as Inner Classes

```
 8       private class RedListener implements ActionListener
 9       {
10           public void actionPerformed(ActionEvent e)
11           {
12               redPanel.setBackground(Color.RED);
13           }
14       } //End of RedListener inner class

15       private class WhiteListener implements ActionListener
16       {
17           public void actionPerformed(ActionEvent e)
18           {
19               whitePanel.setBackground(Color.WHITE);
20           }
21       } //End of WhiteListener inner class
```

(continued)

Listeners as Inner Classes (Part 3 of 6)

Display 17.16 Listeners as Inner Classes

```
22      private class BlueListener implements ActionListener
23      {
24          public void actionPerformed(ActionEvent e)
25          {
26              bluePanel.setBackground(Color.BLUE);
27          }
28      } //End of BlueListener inner class

29      public static void main(String[] args)
30      {
31          InnerListenersDemo gui = new InnerListenersDemo();
32          gui.setVisible(true);
33      }
```

(continued)

Listeners as Inner Classes (Part 4 of 6)

Display 17.16 Listeners as Inner Classes

The resulting GUI is the same as in Display 17.14.

```
34      public InnerListenersDemo()
35      {
36          super("Menu Demonstration");
37          setSize(WIDTH, HEIGHT);
38          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39          setLayout(new GridLayout(1, 3));

40          redPanel = new JPanel();
41          redPanel.setBackground(Color.LIGHT_GRAY);
42          add(redPanel);

43          whitePanel = new JPanel();
44          whitePanel.setBackground(Color.LIGHT_GRAY);
45          add(whitePanel);
```

(continued)

Listeners as Inner Classes (Part 5 of 6)

Display 17.16 Listeners as Inner Classes

```
46          bluePanel = new JPanel();
47          bluePanel.setBackground(Color.LIGHT_GRAY);
48          add(bluePanel);

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

50          JMenuItem redChoice = new JMenuItem("Red");
51          redChoice.addActionListener(new RedListener());
52          colorMenu.add(redChoice);
```

(continued)

Listeners as Inner Classes (Part 6 of 6)

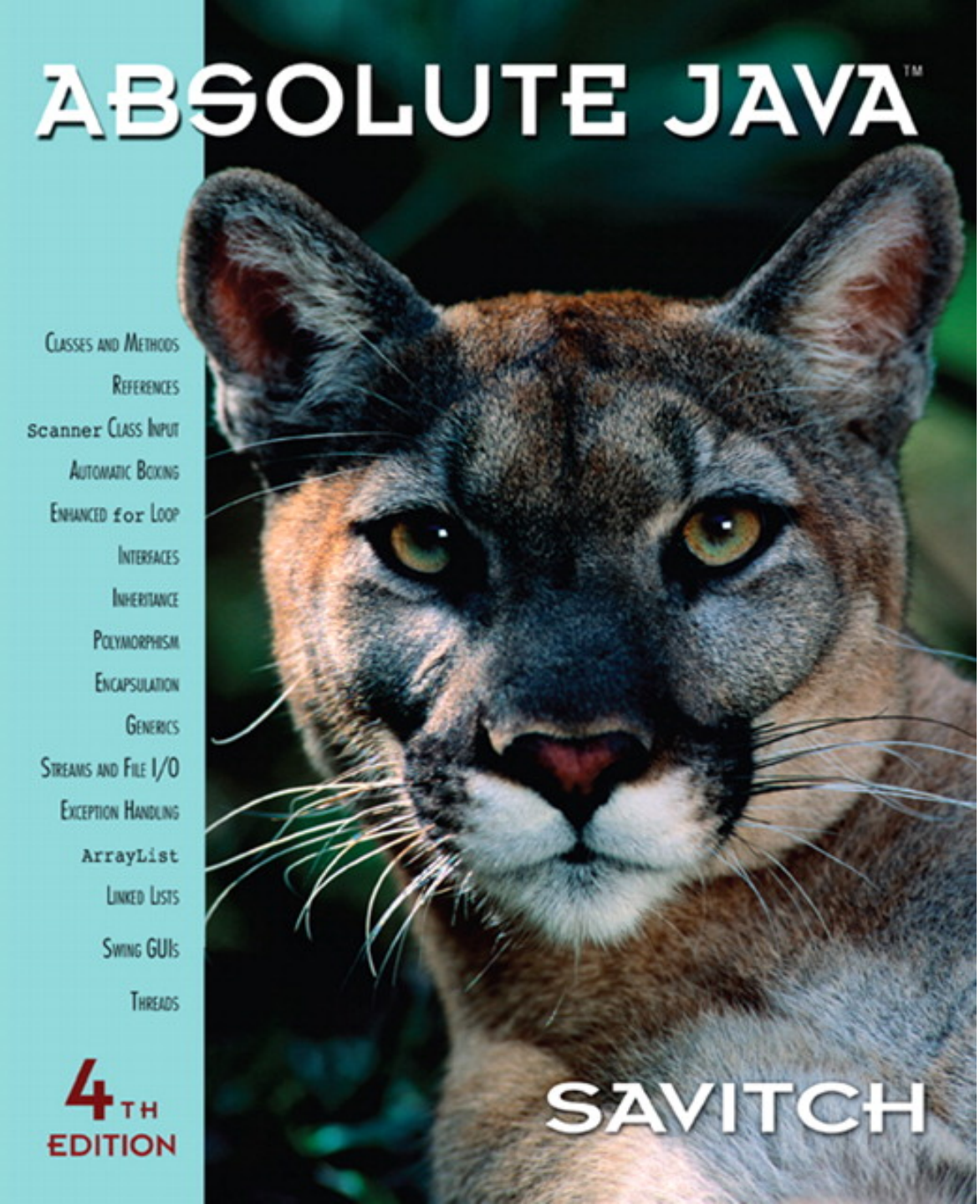
Display 17.16 Listeners as Inner Classes

```
53          JMenuItem whiteChoice = new JMenuItem("White");
54          whiteChoice.addActionListener(new WhiteListener());
55          colorMenu.add(whiteChoice);

56          JMenuItem blueChoice = new JMenuItem("Blue");
57          blueChoice.addActionListener(new BlueListener());
58          colorMenu.add(blueChoice);

59          JMenuBar bar = new JMenuBar();
60          bar.add(colorMenu);
61          setJMenuBar(bar);
62      }

63  }
```



Chapter 18

Swing II

PEARSON Addison Wesley

Coordinate System for Graphics Objects

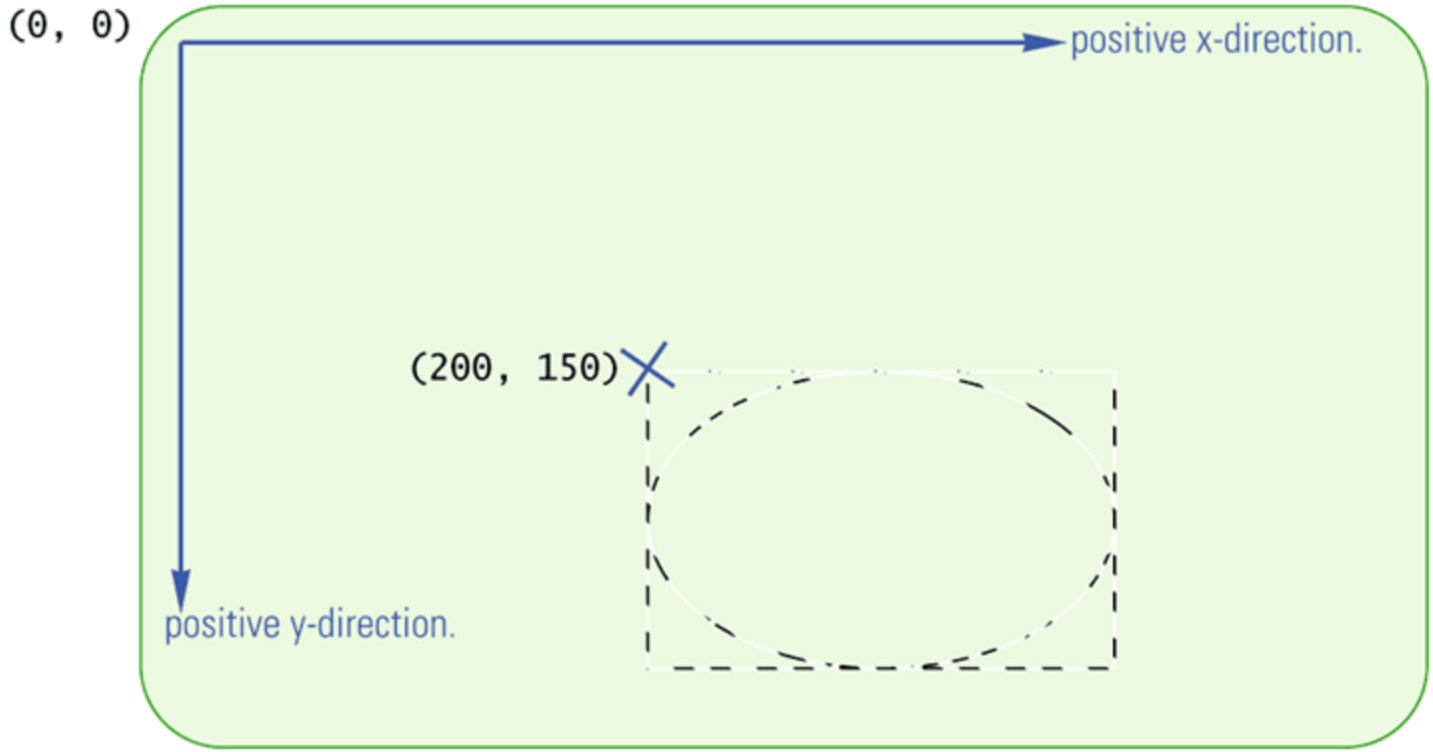
- When drawing objects on the screen, Java uses a coordinate system where the origin point (0,0) is at the upper-left corner of the screen area used for drawing
  - The x-coordinate (horizontal) is positive and increasing to the right
  - The y- coordinate(vertical) is positive and increasing down
  - All coordinates are normally positive
  - Units and sizes are in pixels
  - The area used for drawing is typically a **`JFrame`** or **`JPanel`**

Coordinate System for Graphics Objects

- The point **(x,y)** is located **x** pixels in from the left edge of the screen, and down **y** pixels from the top of the screen
- When placing a rectangle on the screen, the location of its upper-left corner is specified
- When placing a figure other than a rectangle on the screen, Java encloses the figure in an imaginary rectangle, called a *bounding box*, and positions the upper-left corner of this rectangle

Screen Coordinate System

Screen Coordinate System



The Method `paint` and the Class `Graphics`

- Almost all Swing and Swing-related components and containers have a method called **`paint`**
- The method **`paint`** draws the component or container on the screen
  - It is already defined, and is called automatically when the figure is displayed on the screen
  - However, it must be redefined in order to draw geometric figures like circles and boxes
  - When redefined, always include the following:

```
super.paint(g);
```

The Method **paint** and the Class **Graphics**

- Every container and component that can be drawn on the screen has an associated **Graphics** object
  - The **Graphics** class is an abstract class found in the **java.awt** package
- This object has data specifying what area of the screen the component or container covers
  - The **Graphics** object for a **JFrame** specifies that drawing takes place inside the borders of the **JFrame** object

The Method **paint** and the Class **Graphics**

- The object **g** of the class **Graphics** can be used as the calling object for a drawing method
  - The drawing will then take place inside the area of the screen specified by **g**
- The method **paint** has a parameter **g** of type **Graphics**
  - When the **paint** method is invoked, **g** is replaced by the **Graphics** object associated with the **JFrame**
  - Therefore, the figures are drawn inside the **JFrame**

Drawing a Very Simple Face (part 1 of 5)

Drawing a Very Simple Face

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


4   public class Face extends JFrame
5   {
6       public static final int WINDOW_WIDTH = 400;
7       public static final int WINDOW_HEIGHT = 400;


8       public static final int FACE_DIAMETER = 200;
9       public static final int X_FACE = 100;
10      public static final int Y_FACE = 100;
```

(continued)

Drawing a Very Simple Face (part 2 of 5)

Drawing a Very Simple Face

```
11      public static final int EYE_WIDTH = 20;
12      public static final int X_RIGHT_EYE = X_FACE + 55;
13      public static final int Y_RIGHT_EYE = Y_FACE + 60;
14      public static final int X_LEFT_EYE = X_FACE + 130;
15      public static final int Y_LEFT_EYE = Y_FACE + 60;


16      public static final int MOUTH_WIDTH = 100;
17      public static final int X_MOUTH = X_FACE + 50;
18      public static final int Y_MOUTH = Y_FACE + 150;
```

(continued)

Drawing a Very Simple Face (part 3 of 5)

Drawing a Very Simple Face

```
19      public static void main(String[] args)
20      {
21          Face drawing = new Face();
22          drawing.setVisible(true);
23      }


24      public Face()
25      {
26          super("First Graphics Demo");
27          setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
28          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29          getContentPane().setBackground(Color.white);
30      }
```

(continued)

Drawing a Very Simple Face (part 4 of 5)

Drawing a Very Simple Face

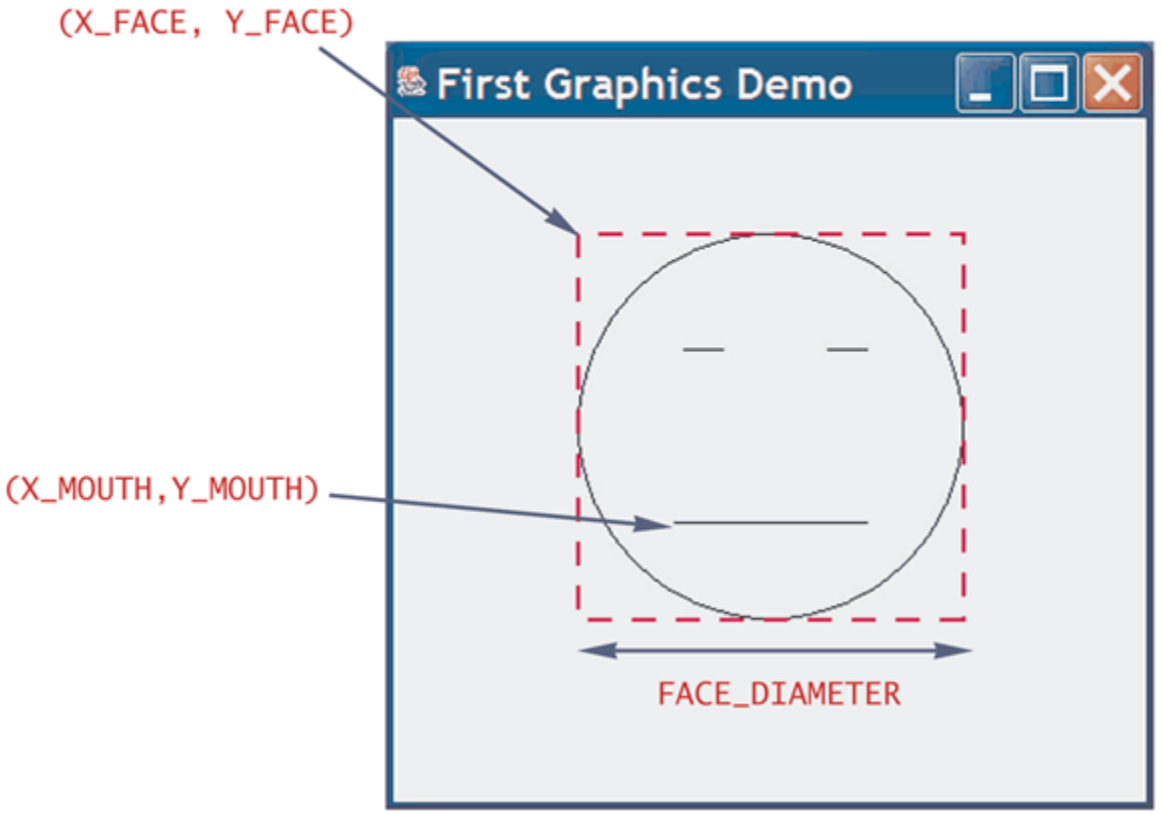
```
31      public void paint(Graphics g)
32      {
33          super.paint(g);
34          g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
35          //Draw Eyes:
36          g.drawLine(X_RIGHT_EYE, Y_RIGHT_EYE,
37                                   X_RIGHT_EYE + EYE_WIDTH,Y_RIGHT_EYE);
38          g.drawLine(X_LEFT_EYE, Y_LEFT_EYE,
39                                  X_LEFT_EYE + EYE_WIDTH, Y_LEFT_EYE);
40          //Draw Mouth:
41          g.drawLine(X_MOUTH, Y_MOUTH, X_MOUTH + MOUTH_WIDTH, Y_MOUTH);
42      }
43  }
```

(continued)

Drawing a Very Simple Face (part 5 of 5)

Drawing a Very Simple Face

RESULTING GUI



The red box is not shown on the screen. It is there to help you understand the relationship between the `paint` *method code and the resulting drawing.*

Some Methods in the Class **`Graphics`** (part 1 of 4)

Some Methods in the Class `Graphics`

`Graphics` is an abstract class in the `java.awt` package.
Although many of these methods are abstract, we always use them with objects of a concrete descendent class of `Graphics`, even though we usually do not know the name of that concrete class.

```
public abstract void drawLine(int x1, int y1, int x2, int y2)
```

Draws a line between points (x1, y1) and (x2, y2).

```
public abstract void drawRect(int x, int y,
                              int width, int height)
```

Draws the outline of the specified rectangle. (x, y) is the location of the upper-left corner of the rectangle.

```
public abstract void fillRect(int x, int y,
                              int width, int height)
```

Fills the specified rectangle. (x, y) is the location of the upper-left corner of the rectangle.

(continued)

Some Methods in the Class **Graphics** (part 3 of 4)

Some Methods in the Class Graphics

```
public abstract void drawRoundRect(int x, int y,
            int width, int height, int arcWidth, int arcHeight)
```

Draws the outline of the specified round-cornered rectangle. (x, y) is the location of the upper-left corner of the enclosing regular rectangle. `arcWidth` and `arcHeight` specify the shape of the round corners. See the text for details.

```
public abstract void fillRoundRect(int x, int y,
            int width, int height, int arcWidth, int arcHeight)
```

Fills the rounded rectangle specified by

```
drawRoundRec(x, y, width, height, arcWidth, arcHeight)
```

```
public abstract void drawOval(int x, int y,
                              int width, int height)
```

Draws the outline of the oval with the smallest enclosing rectangle that has the specified width and height. The (imagined) rectangle has its upper-left corner located at (x, y).

(continued)

Some Methods in the Class **Graphics** (part 4 of 4)

Some Methods in the Class Graphics

```
public abstract void fillOval(int x, int y,
                              int width, int height)
```

Fills the oval specified by

```
drawOval(x, y, width, height)
```

```
public abstract void drawArc(int x, int y,
                             int width, int height,
                             int startAngle, int arcSweep)
```

Draws part of an oval that just fits into an invisible rectangle described by the first four arguments. The portion of the oval drawn is given by the last two arguments. See the text for details.

```
public abstract void fillArc(int x, int y,
                             int width, int height,
                             int startAngle, int arcSweep)
```

Fills the partial oval specified by

```
drawArc(x, y, width, height, startAngle, arcSweep)
```

Drawing Ovals

- An oval is drawn by the method **drawOval**
  - The arguments specify the location, width, and height of the smallest rectangle that can enclose the oval

```
g.drawOval(100, 50, 300, 200);
```

- A circle is a special case of an oval in which the width and height of the rectangle are equal

```
g.drawOval(X_FACE, Y_FACE,
           FACE_DIAMETER, FACE_DIAMETER);
```

`paintComponent` for Panels

- A **`JPanel`** is a **`JComponent`**, but a **`JFrame`** is a **`Component`**, not a **`JComponent`**
  - Therefore, they use different methods to paint the screen
- Figures can be drawn on a **`JPanel`**, and the **`JPanel`** can be placed in a **`JFrame`**
  - When defining a **`JPanel`** class in this way, the **`paintComponent`** method is used instead of the **`paint`** method
  - Otherwise the details are the same as those for a **`JFrame`**

Action Drawings and `repaint`

- The **`repaint`** method should be invoked when the graphics content of a window is changed
  - The **`repaint`** method takes care of some overhead, and then invokes the method **`paint`**, which redraws the screen
  - Although the **`repaint`** method must be explicitly invoked, it is already defined
  - The **`paint`** method, in contrast, must often be defined, but is not explicitly invoked

An Action Drawing (Part 1 of 7)

An Action Drawing

```
1   import javax.swing.JFrame;
2   import javax.swing.JButton;
3   import java.awt.event.ActionListener;
4   import java.awt.event.ActionEvent;
5   import java.awt.BorderLayout;
6   import java.awt.Graphics;
7   import java.awt.Color;

8   public class ActionFace extends JFrame
9   {
10      public static final int WINDOW_WIDTH = 400;
11      public static final int WINDOW_HEIGHT = 400;
```

(continued)

An Action Drawing (Part 2 of 7)

An Action Drawing

```
12      public static final int FACE_DIAMETER = 200;
13      public static final int X_FACE = 100;
14      public static final int Y_FACE = 100;

15      public static final int EYE_WIDTH = 20;
16      public static final int EYE_HEIGHT = 10;
17      public static final int X_RIGHT_EYE = X_FACE + 55;
18      public static final int Y_RIGHT_EYE = Y_FACE + 60;
19      public static final int X_LEFT_EYE = X_FACE + 130;
20      public static final int Y_LEFT_EYE = Y_FACE + 60;
```

(continued)

An Action Drawing (Part 3 of 7)

An Action Drawing

```
21      public static final int MOUTH_WIDTH = 100;
22      public static final int MOUTH_HEIGHT = 50;
23      public static final int X_MOUTH = X_FACE + 50;
24      public static final int Y_MOUTH = Y_FACE + 100;
25      public static final int MOUTH_START_ANGLE = 180;
26      public static final int MOUTH_ARC_SWEEP = 180;

27      private boolean wink;

28      private class WinkAction implements ActionListener
29      {
30          public void actionPerformed(ActionEvent e)
31          {
32              wink = true;
33              repaint();
34          }
35      } // End of WinkAction inner class
```

(continued)

An Action Drawing (Part 4 of 7)

An Action Drawing

```
36      public static void main(String[] args)
37      {
38          ActionFace drawing = new ActionFace();
39          drawing.setVisible(true);
40      }

41      public ActionFace()
42      {
43          setSize(WINDOW_WIDTH, WINDOW_HEIGHT);
44          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
45          setTitle("Hello There!");
46          setLayout(new BorderLayout());
47          getContentPane().setBackground(Color.white);

48          JButton winkButton = new JButton("Click for a Wink.");
49          winkButton.addActionListener(new WinkAction());
50          add(winkButton, BorderLayout.SOUTH);
51          wink = false;
52      }
```

(continued)

An Action Drawing (Part 5 of 7)

An Action Drawing

```
53      public void paint(Graphics g)
54      {
55          super.paint(g);
56          g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
57          //Draw Right Eye:
58          g.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
59          //Draw Left Eye:
60          if (wink)
61              g.drawLine(X_LEFT_EYE, Y_LEFT_EYE,
62                                     X_LEFT_EYE + EYE_WIDTH, Y_LEFT_EYE);
63          else
64              g.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
65          //Draw Mouth:
66          g.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
67                    MOUTH_START_ANGLE, MOUTH_ARC_SWEEP);
68      }

69  }
```

(continued)

An Action Drawing (Part 6 of 7)

An Action Drawing

RESULTING GUI (When started)

(continued)

An Action Drawing (Part 7 of 7)

An Action Drawing

RESULTING GUI (After clicking the button)

Some More Details on Updating a GUI

- With Swing, most changes to a GUI are updated automatically to become visible on the screen
  - This is done by the *repaint manager* object
- Although the repaint manager works automatically, there are a few updates that it does not perform
  - For example, the ones taken care of by **`validate`** or **`repaint`**
- One other updating method is **`pack`**
  - **`pack`** resizes the window to something known as the preferred size

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`**

Specifying a Drawing Color

- Using the method **drawLine** inside the **paint** method is similar to drawing with a pen that can change colors
  - The method **setColor** will change the color of the pen
  - The color specified can be changed later on with another invocation of **setColor** so that a single drawing can have multiple colors

```
g.setColor(Color.BLUE)
```

Adding Color

Adding Color

```
1       public void paint(Graphics g)
2       {
3           super.paint(g);
4           //Default is equivalent to: g.setColor(Color.black);
5           g.drawOval(X_FACE, Y_FACE, FACE_DIAMETER, FACE_DIAMETER);
6           //Draw Eyes:
7           g.setColor(Color.BLUE);
8           g.fillOval(X_RIGHT_EYE, Y_RIGHT_EYE, EYE_WIDTH, EYE_HEIGHT);
9           g.fillOval(X_LEFT_EYE, Y_LEFT_EYE, EYE_WIDTH, EYE_HEIGHT);
10          //Draw Mouth:
11          g.setColor(Color.RED);
12          g.drawArc(X_MOUTH, Y_MOUTH, MOUTH_WIDTH, MOUTH_HEIGHT,
13                    MOUTH_START_ANGLE, MOUTH_ARC_SWEEP);
14      }
```

If you replace the **paint** method in Display 18.13 with this version then the happy face will have blue eyes and red lips.

Defining Colors

- Standard colors in the class **`Color`** are already defined
  - These are listed in Display 17.5 in Chapter 17, and shown on the following slide
- The **`Color`** class can also be used to define additional colors
  - It uses the *RGB color system* in which different amounts of red, green, and blue light are used to produce any color

The Color Constants

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`	

Defining Colors

- Integers or floats may be used when specifying the amount of red, green, and/or blue in a color
  - Integers must be in the range 0-255 inclusive

```
Color brown = new Color(200, 150, 0);
```

  - `float` values must be in the range 0.0-1.0 inclusive

```
Color brown = new Color(
  (float)(200.0/255),(float)(150.0/255),
                                (float)0.0);
```

The `drawString` Method

- The method **`drawString`** is similar to the drawing methods in the **`Graphics`** class
  - However, it displays text instead of a drawing
  - If no font is specified, a default font is used

```
g.drawString(theText, X_START, Y_Start);
```

Using **drawString** (Part 1 of 7)

Using drawString

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

(continued)

Using **drawString** (Part 2 of 7)

Using drawString

```
10  public class DrawStringDemo extends JFrame
11                               implements ActionListener
12  {
13      public static final int WIDTH = 350;
14      public static final int HEIGHT = 200;
15      public static final int X_START = 20;
16      public static final int Y_START = 100;
17      public static final int POINT_SIZE = 24;

18      private String theText = "Push the button.";
19      private Color penColor = Color.BLACK;
20      private Font fontObject =
21                        new Font("SansSerif", Font.PLAIN, POINT_SIZE);

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

(continued)

Using **drawString** (Part 3 of 7)

Using drawString

```
27      public DrawStringDemo()
28      {
29          setSize(WIDTH, HEIGHT);
30          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31          setTitle("drawString Demonstration");

32          getContentPane().setBackground(Color.WHITE);
33          setLayout(new BorderLayout());

34          JPanel buttonPanel = new JPanel();
35          buttonPanel.setBackground(Color.GRAY);
36          buttonPanel.setLayout(new BorderLayout());
```

(continued)

Using `drawString` (Part 4 of 7)

Using `drawString`

```
37          JButton theButton = new JButton("The Button");
38          theButton.addActionListener(this);


39          buttonPanel.add(theButton, BorderLayout.CENTER);


40          add(buttonPanel, BorderLayout.SOUTH);
41      }


42      public void paint(Graphics g)
43      {
44          super.paint(g);
45          g.setFont(fontObject);
46          g.setColor(penColor);
47          g.drawString(theText, X_START, Y_START);
48      }
```

(continued)

Using **drawString** (Part 5 of 7)

Using drawString

```
49        public void actionPerformed(ActionEvent e)
50        {
51            penColor = Color.RED;
52            fontObject =
53                   new Font("Serif", Font.BOLD|Font.ITALIC, POINT_SIZE);
54            theText = "Thank you. I needed that.";


55            repaint();
56        }
57    }
```

(continued)

Android Introduction

Application Fundamentals

Goal

- Understand applications and their components
- Concepts:
  - activity,
  - service,
  - broadcast receiver,
  - content provider,
  - intent,
  - AndroidManifest

Applications

- Written in Java (it's possible to write native code – will not cover that here)
- Good separation (and corresponding security) from other applications:
  - Each application runs in its own process
  - Each process has its own separate VM
  - Each application is assigned a unique Linux user ID – by default files of that application are only visible to that application (can be explicitly exported)

Application Components

- Activities – visual user interface focused on a single thing a user can do
- Services – no visual interface – they run in the background
- Broadcast Receivers – receive and react to broadcast announcements
- Content Providers – allow data exchange between applications

Activities

- Basic component of most applications
- Most applications have several activities that start each other as needed
- Each is implemented as a subclass of the base Activity class

Activities – The View

- Each activity has a default window to draw in (although it may prompt for dialogs or notifications)
- The content of the window is a view or a group of views (derived from View or ViewGroup)
- Example of views: buttons, text fields, scroll bars, menu items, check boxes, etc.
- View(Group) made visible via Activity. setContentView() method.

Services

- Does not have a visual interface
- Runs in the background indefinitely
- Examples
  - Network Downloads
  - Playing Music
  - TCP/UDP Server
- You can bind to a an existing service and control its operation

Intents

- An intent is an Intent object with a message content.
- Activities, services and broadcast receivers are started by intents. ContentProviders are started by ContentResolvers:
  - An activity is started by Context.startActivity(Intent intent) or Activity.startActivityForResult(Intent intent, int RequestCode)
  - A service is started by Context.startService(Intent service)
  - An application can initiate a broadcast by using an Intent in any of Context.sendBroadcast(Intent intent), Context. sendOrderedBroadcast(), and Context.sendStickyBroadcast()

Shutting down components

- Activities
  - Can terminate itself via finish();
  - Can terminate other activities it started via finishActivity();
- Services
  - Can terminate via stopSelf(); or Context.stopService();
- Content Providers
  - Are only active when responding to ContentResolvers
- Broadcast Receivers
  - Are only active when responding to broadcasts

Android Manifest

- Its main purpose in life is to declare the components to the system:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest . . . >
    <application . . . >
        <activity android:name="com.example.project.FreneticActivity"
                  android:icon="@drawable/small_pic.png"
                  android:label="@string/freneticLabel"
                  . . .  >
        </activity>
        . . .
    </application>
</manifest>
```

Intent Filters

- Declare Intents handled by the current application (in the AndroidManifest):

```
<?xml version="1.0" encoding="utf-8"?>
<manifest . . . >
   <application . . . >
      <activity android:name="com.example.project.FreneticActivity"
                android:icon="@drawable/small_pic.png"
                android:label="@string/freneticLabel"
                . . .  >
         <intent-filter . . . >
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
         <intent-filter . . . >
            <action android:name="com.example.project.BOUNCE" />
            <data android:mimeType="image/jpeg" />
            <category android:name="android.intent.category.DEFAULT" />
         </intent-filter>
      </activity>
      . . .
   </application>
</manifest>
```

Shows in the Launcher and is the main activity to start

Handles JPEG images in some way

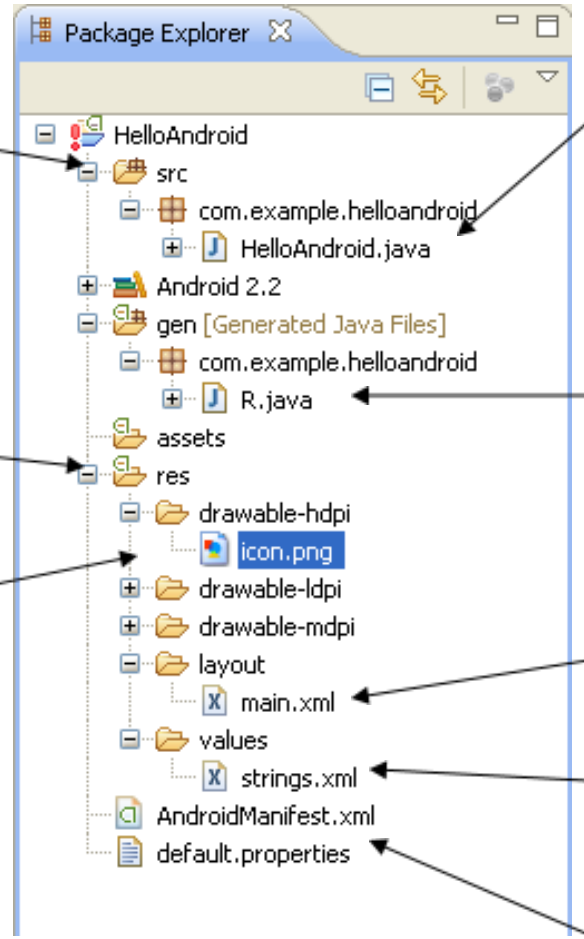
Android Introduction

Hello World

Package Content

All source code here

Java code for our activity



All non-code resources

Generated Java code
Helps link resources to Java code

Images

Layout of the activity

Strings used in the program

Android Manifest

Android Manifest

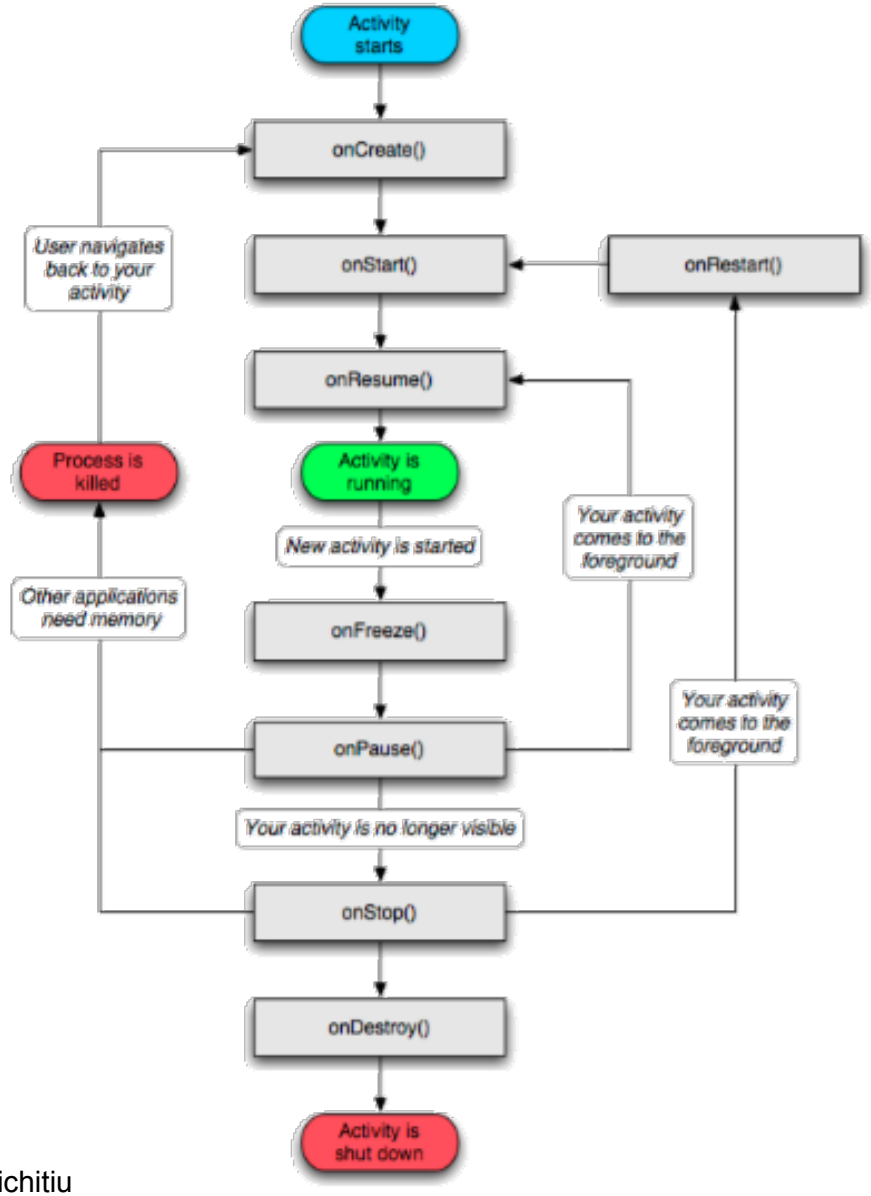
- <?xml version="1.0" encoding="utf-8"?>
- <manifest xmlns:android="http://schemas.android.com/apk/res/android"
- package="com.example.helloandroid"
- android:versionCode="1"
- android:versionName="1.0">
- <application android:icon="@drawable/icon" android:label="@string/app_name">
- <activity android:name=".HelloAndroid"
- android:label="@string/app_name">
- <intent-filter>
- <action android:name="android.intent.action.MAIN" />
- <category android:name="android.intent.category.LAUNCHER" />
- </intent-filter>
- </activity>

- </application>

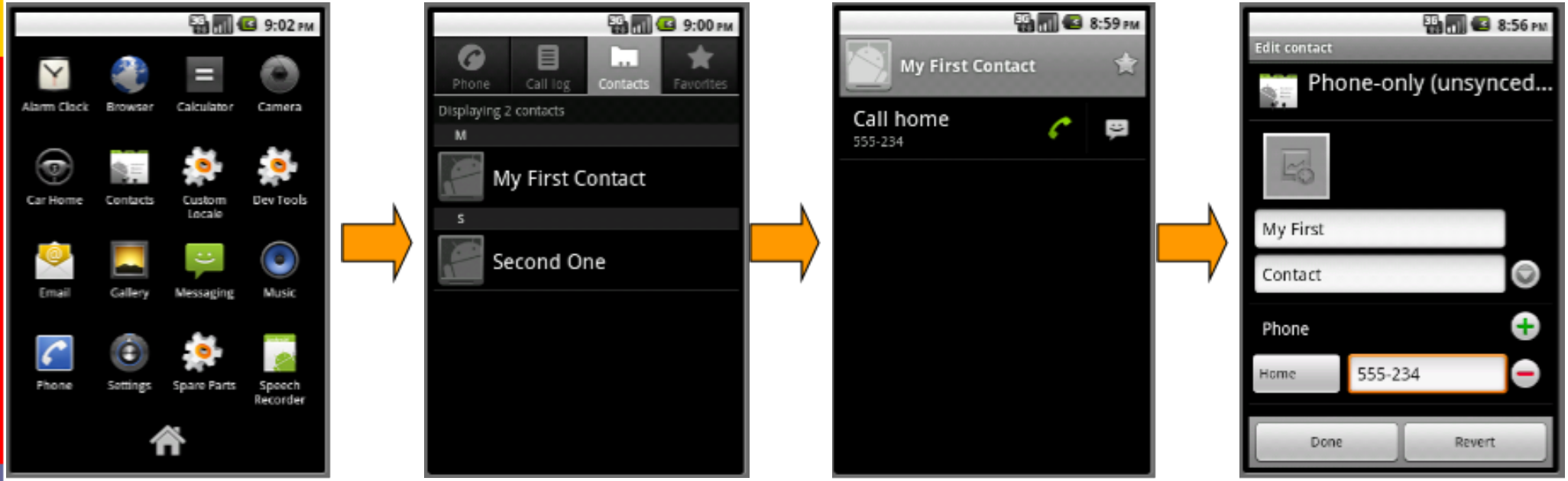
- </manifest>

Activity "Lifecycle"

- An Android activity is focused on a single thing a user can do.
- Most applications have multiple activities



Activities start each other



You should understand the differences between startActivity() and startActivityForResult()--see sample code in HelloWidgetMania

Revised HelloAndroid.java

```
package com.example.helloandroid;

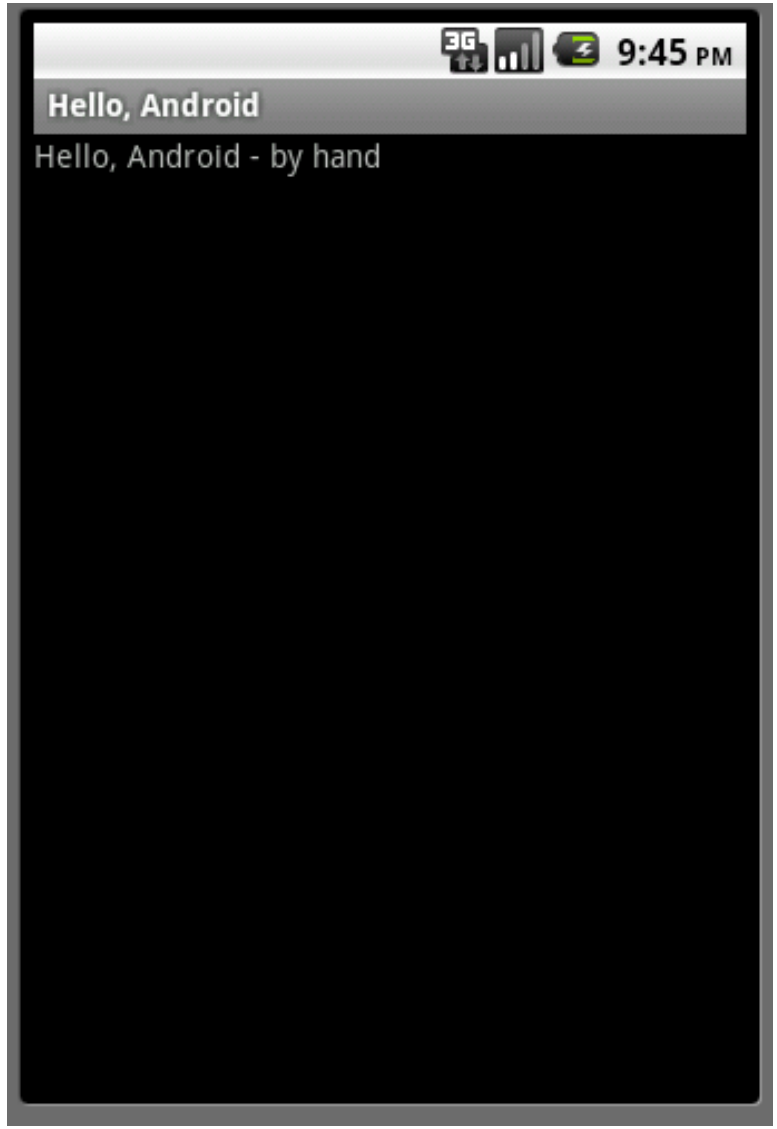
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloAndroid extends Activity {
   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
TextView tv = new TextView(this);
      tv.setText("Hello, Android – by hand");
      setContentView(tv);
   }
}
```

Inherit from the Activity Class

Set the view “by hand” – from the program

Run it!



/res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
>
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/hello"
/>
</LinearLayout>
```

Further redirection to
/res/values/strings.xml

/res/values/strings.xml

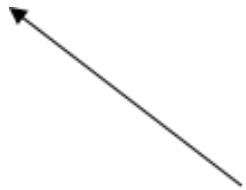
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="hello">Hello World, HelloAndroid – by resources!</string>
<string name="app_name">Hello, Android</string>
</resources>
```

HelloAndroid.java

```
package com.example.helloandroid;

import android.app.Activity;
import android.os.Bundle;
public class HelloAndroid extends Activity {

/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState); setContentView(R.layout.main);
}
}
```



Set the layout of the view as described in the main.xml layout

Android Introduction

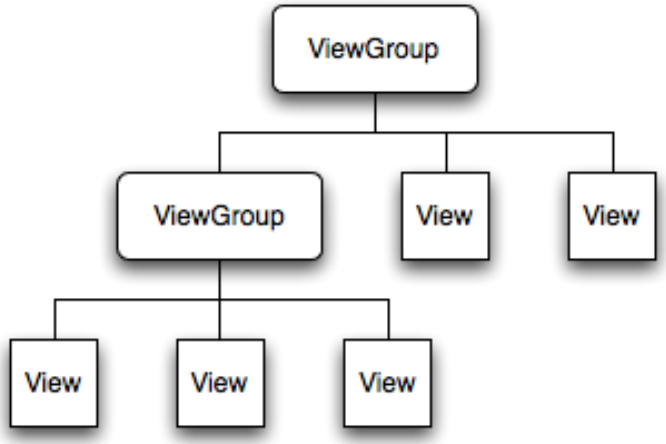
Graphical User Interface

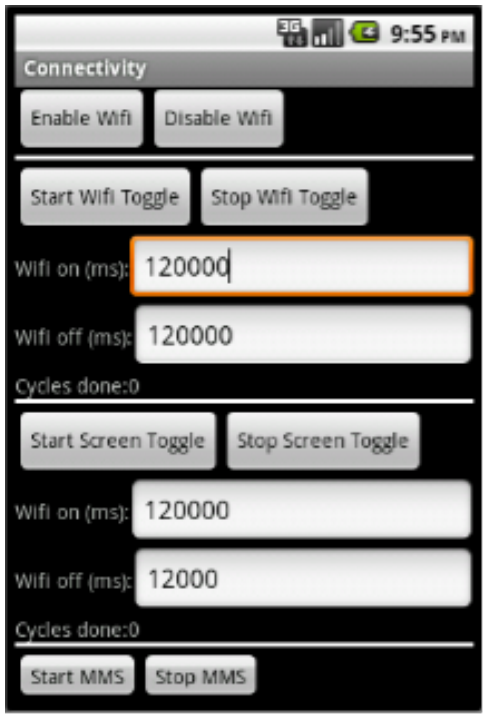
Goal

- Be familiar with the main types of GUI concepts:
  - Layouts
  - Widgets
  - Events

View Hierarchy

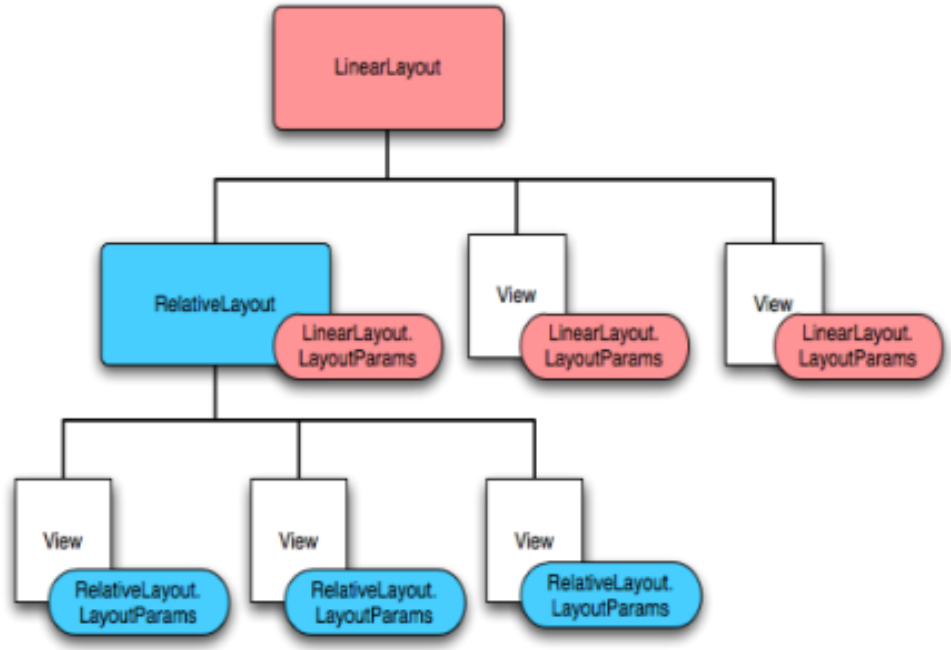
- All the views in a window are arranged in a tree
- You show the tree by calling setContentView(rootNode) in the activity





Layout

- Defines how elements are positioned relative to each other (next to each other, under each other, in a table, grid, etc.)
- Can have a different layouts for each ViewGroup



Linear Layout (Horizontal vs. vertical)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">

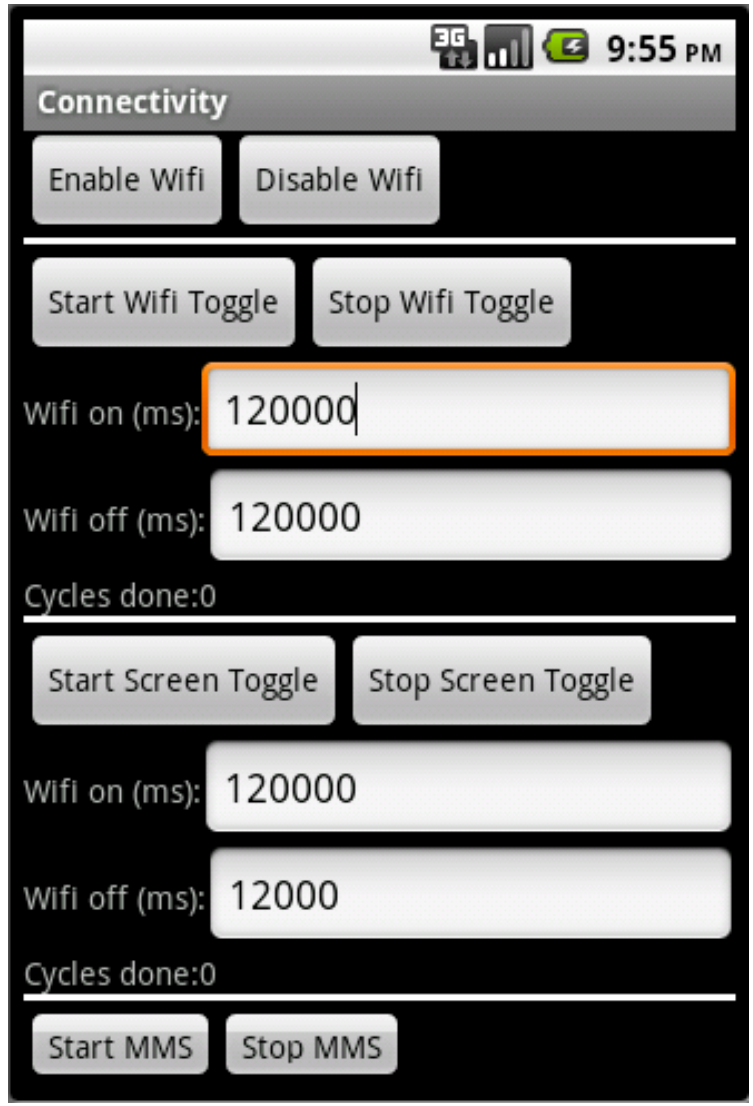
  <LinearLayout
     android:orientation="horizontal"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     android:layout_weight="1">
     <TextView
        android:text="red"
        android:gravity="center_horizontal"
[……….]
         </LinearLayout>

  <LinearLayout
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1">
    <TextView
       android:text="row one"
       android:textSize="15pt"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:layout_weight="1"/>
    <TextView
       android:text="row two"
       android:textSize="15pt"
       android:layout_width="fill_parent"
       android:layout_height="wrap_content"
       android:layout_weight="1"/>
[……..]
  </LinearLayout>
</LinearLayout>
```

http://developer.android.com/resources/tutorials/views/hello-linearlayout.html

Widgets

- All are View objects
- Examples:
  - TextFields
  - EditFields
  - Buttons
  - Checkboxes
  - RadioButtons
  - etc.

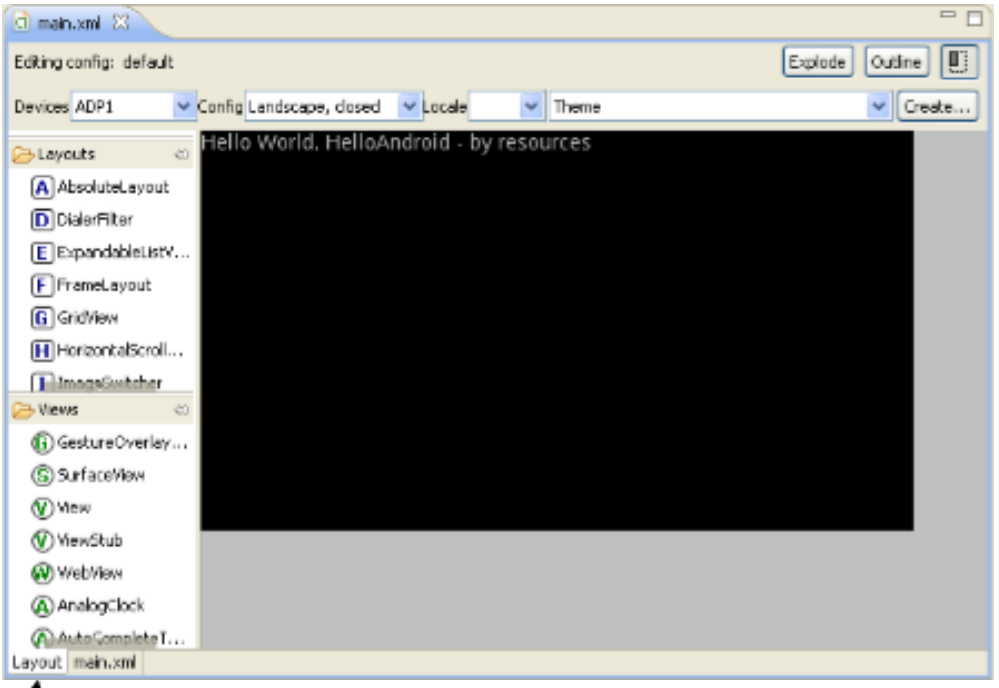


UI Events

- Usually handled by defining a Listener of the form On<something>Listener and register it with the View
- For example:
  - OnClickListener() for handling clicks on Buttons or Lists
  - OnTouchListener() for handling touches
  - OnKeyListerner() for handling key presses
- Alternatively, Override an existing callback if we implemented our own class extending View

  Lots of sample code in HelloWidgetMania

Eclipse layout Manager: Two views



XML File vs Layout Preview

More Android Concepts

- The basic structure of an animated game
  - Represent state of game in variables
  - Render state of game by drawing from variables
  - Update state of game through:
    - User input (touch, keypad, acclerometer, etc.)
    - "Physics": Simulate forces, continue motion, check for collisions, etc.
- 2-D graphics: Familiarity with how to draw shapes, images (loaded from res/drawable), text, change colors, etc.
- Sample code: Be familiar with doDraw() & updatePhysics() in MarsLander
- Miscellaneous
  - Storing/loading preferences, reading text file
  - Sound: SoundPool vs. MediaPlayer -- what each is best suited for
  - Text-to-speech, vibration, accelerometer only at the level of **what** do they do--not **how** they do it