Graphical User Interfaces I

Reading

- 2nd Ed: Chapter 14
- 3rd & 4th Eds: Chapter 18,
- 3rd Ed: Chapter 9.6G, 9.7G, 9.8G
- 4th Ed: Chapter 9.7G, 9.8G, 9.9G

References:

http://www.ugrad.cs.ubc.ca/~cs219/CourseNotes/Swing/intro.html

http://java.sun.com/docs/books/tutorial/ui/index.html

Learning Objectives

- describe basic principles of good user interface design
- given a typical set of Java GUI components, be able to explain how it was created
- use layout managers to produce a well designed GUI
- write code to produce a well designed GUI that includes frames, panels, menus and buttons

Types of User Interfaces

- Command-line interfaces
  - User specifies program behaviour through command-line arguments, or command typed at a prompt
  - Pro: it is fast, especially for massive operations (commands can be read from scripts to control the programs)
  - Con: difficult to use; requires memorizing commands
- Text-based menus
  - User is presented with menus and prompts throughout
  - Pro: easy to use, easy to program
  - Con: it is slow to use

Types of User Interfaces

- Graphical user interfaces
  - Use multiple input modes (keyboard + mouse)
  - Pro: more intuitive and very easy to use
  - Con: more difficult to program

GUI Design Guidelines

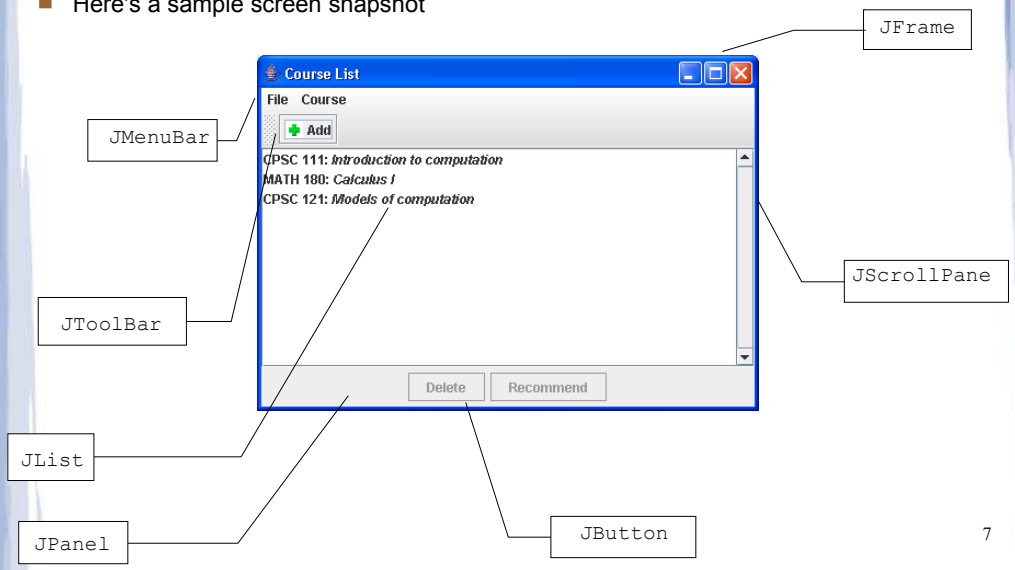
- A GUI should be:
  - as simple as possible (should make things easier not harder)
  - intuitive (explicit dialogs , etc.)
  - consistent (follow conventions)
  - aesthetically pleasing not distracting
  - provide feedback to the user
  - have undo/exit/cancel options when appropriate and possible
  - accommodate multiple skill levels (allow keyboard and mouse input)

Swing Components

- *Swing* is a collection of classes that allows us to incorporate a graphical user interface in a Java program
- A Swing GUI contains:
  - Components (`JComponent`)
    - Parts of the user interface that we interact with
    - Examples: buttons, labels, text fields, scroll panes, menus
  - Panels (`JPanel`)
    - Borderless windows, used to group components together
    - Can contain components, or other panels
    - Each `JPanel` has an associated *layout* that determines how the components inside the panel are placed
  - Windows: Frames (`JFrame`), Dialogs (`JDialog, JOptionPane`)
    - Each `JFrame` has
      - a *title*
      - a *menu bar*
      - a *content pane* (an instance of `JPanel`) into which other elements can be added

GUI Example

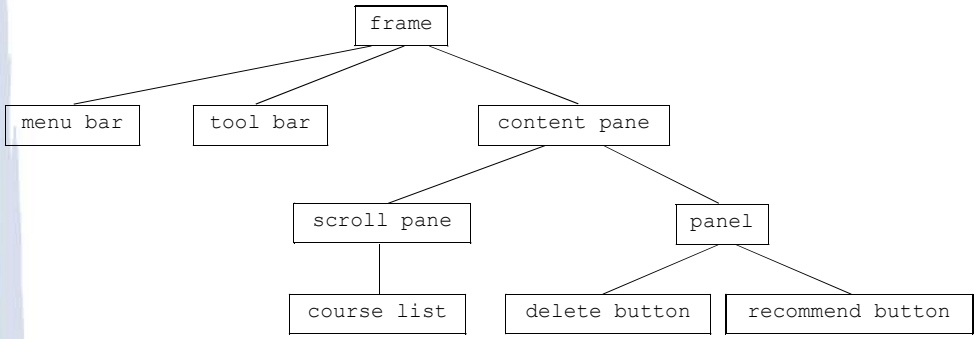
- Here's a sample screen snapshot



Containment Hierarchy

- GUI components can be nested
- The GUI's *containment hierarchy*
  - Represents the structure of the user interface for a specific application
  - Shows which components contain which other components.
- Components that can contain other components are usually called *containers*.
- The containment hierarchy for the example is on next page.

Containment Hierarchy Example



Simple Graphics

To begin with, we need a "canvas" or a "blank sheet of paper" on which to draw. In Java, this is called a frame window or just a frame. You don't put your graphics just anywhere you want...you draw them inside the frame.

It should come as no surprise that a specific frame that we draw in will be an object of some class that serves as a template for frames. Remember, nothing much happens in Java until we create objects.

Making a frame window

Step 1: Construct an object of the JFrame class.

Making a frame window

```
import javax.swing.JFrame;  //Swing is a user interface toolkit

public class FrameViewer
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();  // make a new JFrame object









  }
}
```

Making a frame window

Step 1: Construct an object of the JFrame class.

Step 2: Set the size of the frame.

Making a frame window

```
import javax.swing.JFrame;

public class FrameViewer
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();  // make a new JFrame object

    final int F_WIDTH = 300;        // 300 pixels wide
    final int F_HEIGHT = 400;       // 400 pixels high

    myframe.setSize(F_WIDTH, F_HEIGHT);









  }
}
```

Making a frame window

Step 1: Construct an object of the JFrame class.

Step 2: Set the size of the frame.

Step 3: Set the title of the frame to appear in the title bar (title bar will be blank if no title is set).

Making a frame window

```
import javax.swing.JFrame;

public class FrameViewer
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();  // make a new JFrame object

    final int F_WIDTH = 300;        // 300 pixels wide
    final int F_HEIGHT = 400;       // 400 pixels high

    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("My Frame");   // this is optional




  }
}
```

Making a frame window

Step 1: Construct an object of the JFrame class.

Step 2: Set the size of the frame.

Step 3: Set the title of the frame to appear in the title bar (title bar will be blank if no title is set).

Step 4: Set the default close operation. When the user clicks the close button, the program stops running.

Making a frame window

```
import javax.swing.JFrame;

public class FrameViewer
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();  // make a new JFrame object

    final int F_WIDTH = 300;        // 300 pixels wide
    final int F_HEIGHT = 400;       // 400 pixels high

    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("My Frame");   // this is optional
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);




  }
}
```

Making a frame window

Step 1: Construct an object of the JFrame class.

Step 2: Set the size of the frame.

Step 3: Set the title of the frame to appear in the title bar (title bar will be blank if no title is set).

Step 4: Set the default close operation. When the user clicks the close button, the program stops running.

Step 5: Make the frame visible.

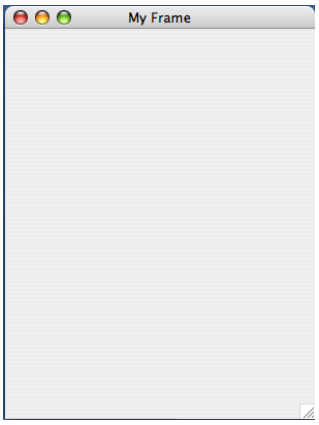
Making a frame window

```
import javax.swing.JFrame;

public class FrameViewer
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();  // make a new JFrame object

    final int F_WIDTH = 300;        // 300 pixels wide
    final int F_HEIGHT = 400;       // 400 pixels high

    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("My Frame");   // this is optional
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);



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

Making a frame window

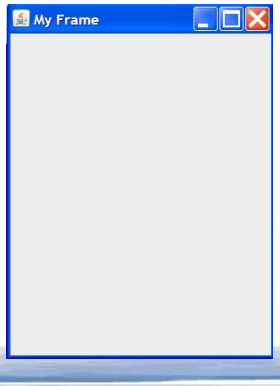
```
import javax.swing.JFrame;

public class FrameViewer
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();  // make a new JFrame object

    final int F_WIDTH = 300;        // 300 pixels wide
    final int F_HEIGHT = 400;       // 400 pixels high

    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("My Frame");   // this is optional
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // when it's time to draw something in the frame,
    // we'll do it here

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


Making a frame window



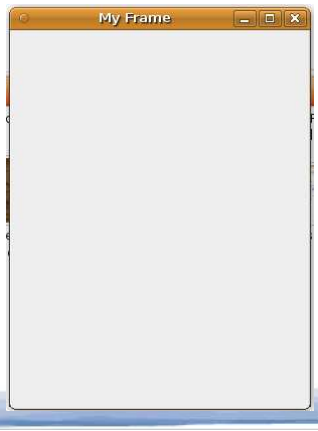
Making a frame window

- The look will be platform-dependent
- The previous example is on a Mac
- Here's one from Windows XP



Making a frame window

- The look will be platform-dependent
- The previous example is on a Mac
- Here's one from Ubuntu Linux



How do we draw something?

Actually, we don't draw anything. We create component objects and add them to the frame we've created. A Swing GUI consists of containers and components. The JFrame frame is our container and we want to add some components to it.

We make our own component in the Swing user interface toolkit by extending the blank component called JComponent to make a RectangleComponent.

The paintComponent() method is inherited from JComponent, then we override the method with our own definition that makes a couple of rectangles.

JComponent

- JComponent is just a generic abstract class
- As we've seen, it's not useful until we extend it

Now let's draw something

```
import java.awt.Graphics;   // AWT is the Abstract Windowing Toolkit,
import java.awt.Graphics2D; // an older graphical user interface
import java.awt.Rectangle;  // toolkit
import javax.swing.JPanel;
import javax.swing.JComponent;

public class RectangleComponent extends JComponent
{
  public void paintComponent(Graphics g)
  {




  }
}
```


Swing

- Swing wasn't originally part of Java
- It was introduced to address deficiencies in AWT (some relating to platform-dependent look-and-feel)

Now let's draw something

The paintComponent() method of an object is called automatically when the frame that contains it is displayed for the first time, resized, or redisplayed after being hidden.

Now let's draw something

```
import java.awt.Graphics;   // AWT is the Abstract Windowing Toolkit,
import java.awt.Graphics2D; // an older graphical user interface
import java.awt.Rectangle;  // toolkit
import javax.swing.JPanel;
import javax.swing.JComponent;

public class RectangleComponent extends JComponent
{
  public void paintComponent(Graphics g)
  {









  }
}
```

Now let's draw something

The paintComponent() method is passed an object of type Graphics2D, which extends the Graphics type, that contains useful information about colour and font to be used, among other things. Graphics2D provides more sophisticated methods for drawing too.
But the paintComponent() method expects a parameter of the older Graphics type, so we use a cast to convert the object to Graphics2D type to recover the methods that come with the Graphics2D class.

Now let's draw something

```
import java.awt.Graphics;   // AWT is the Abstract Windowing Toolkit,
import java.awt.Graphics2D; // an older graphical user interface
import java.awt.Rectangle;  // toolkit
import javax.swing.JPanel;
import javax.swing.JComponent;

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









  }
}
```

Now let's draw something

Now we draw a box. We give the X- and Y- coordinates of the upper left hand corner of the box, along with its width and height in pixels (i.e. picture elements).

Now let's draw something

```
import java.awt.Graphics;   // AWT is the Abstract Windowing Toolkit,
import java.awt.Graphics2D; // an older graphical user interface
import java.awt.Rectangle;  // toolkit
import javax.swing.JPanel;
import javax.swing.JComponent;

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

    Rectangle box = new Rectangle(5, 10, 50, 75);
    g2.draw(box);



  }
}
```

Now let's draw something

The translate() method allows the programmer to start the drawing of the next box at different X- and Y-coordinates.

Now let's draw something

```
import java.awt.Graphics;   // AWT is the Abstract Windowing Toolkit,
import java.awt.Graphics2D; // an older graphical user interface
import java.awt.Rectangle;  // toolkit
import javax.swing.JPanel;
import javax.swing.JComponent;

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

    Rectangle box = new Rectangle(5, 10, 50, 75);
    g2.draw(box);

    box.translate(80,100);


  }
}
```

Now let's draw something

Now we can draw the second and final box.

Now let's draw something

```
import java.awt.Graphics;   // AWT is the Abstract Windowing Toolkit,
import java.awt.Graphics2D; // an older graphical user interface
import java.awt.Rectangle;  // toolkit
import javax.swing.JPanel;
import javax.swing.JComponent;

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

    Rectangle box = new Rectangle(5, 10, 50, 75);
    g2.draw(box);

    box.translate(80,100);

    g2.draw(box);
  }
}
```

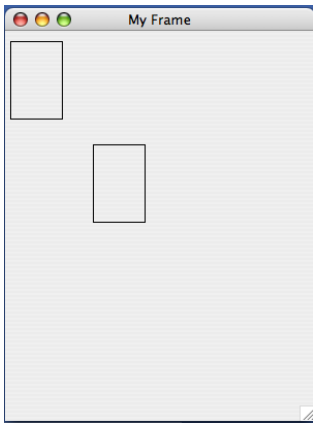
Now let's draw something

One more thing: we have to add the rectangle component to our frame object.

Now let's draw something

```
import javax.swing.JFrame;

public class FrameViewer
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();  // make a new JFrame object

    final int F_WIDTH = 300;        // 300 pixels wide
    final int F_HEIGHT = 400;       // 400 pixels high

    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("My Frame");   // this is optional
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

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

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

Here's what we drew



Graphical user interfaces (GUIs)

The graphical user interface allows us to interact with our programs through mouse movements, button clicks, key presses, and so on.

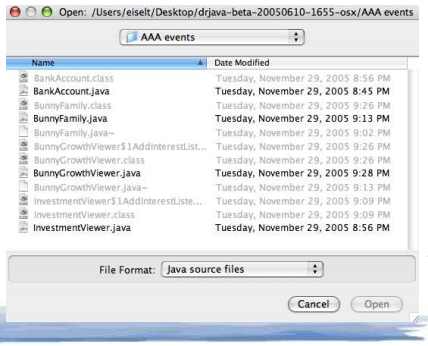
Your Windows or Macintosh or Linux operating system provides you with a GUI so you don't have to remember all sorts of instructions to type at the command line.

Graphical user interfaces (GUIs)

The graphical user interface allows us to interact with our programs through mouse movements, button clicks, key presses, and so on.

Your Windows or Macintosh operating system provides you with a GUI so you don't have to remember all sorts of instructions to type at the command line.

Here's a GUI you may be familiar with.



GUIs

- The graphics we've created so far don't really count as GUIs
- There's no user interaction, except maximize/minimize/close
- To allow user interaction, we need to do something like add a button that can be pushed and which causes some action to occur

GUIs

- GUIs are based on **event-handling**
- What's an **event**? Someone pushing a button, pressing a key on the keyboard, checking a box, and so on. Events are represented as objects.
- An **event source** is an object that turns a user action into an Event object. A **button** is an example of an event source. It creates an Event object when a user clicks the button.

Event listeners

- Say we create a button component. We now have an event source that can create event objects when someone pushes it.
- But how will we know when someone pushes the button?
- We need to create an **event listener**.
- We tell the button that we're interested in its events, and we do so by creating a listener object and passing this to the button

Event handling

An ***event listener*** is an object that belongs to a class which you define. The methods in your event listener contain the instructions to be executed when the events occur.

Any event listener is specific to an event source. For example, you'd have one kind of event listener to respond to the click of a button on your mouse, and another to respond to the press of a key on your keyboard.

When an event occurs, the event source calls the appropriate methods of all associated event listeners.

Event handling

This example is a simple program that prints a message when a button is clicked.

An event listener that responds to button clicks must belong to a class that implements the ActionListener interface. That interface, supplied by the Abstract Windowing Toolkit (AWT), looks like this:

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

Java uses the event parameter to pass details about the event. We don't need to worry about it.

Other listeners

- Mouse listener
- Key listener
- Focus listener
- etc.
- http://download.oracle.com/docs/cd/E17409_01/javase/tutorial/uiswing/events/eventsandcomponents.html

Event handling

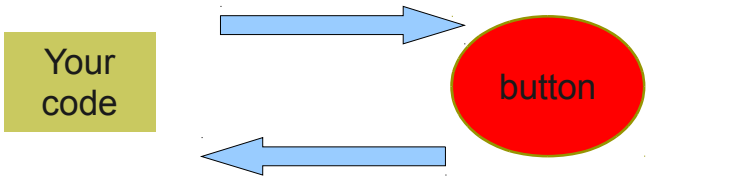
Here's what our example class that implements the ActionListener interface looks like:

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

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

The actionPerformed() method contains the instructions we want to be executed when our button is clicked.

Event listeners

button.addActionListener(listener)
// hey button, I care what happens to you



listener.actionPerformed()
// somebody clicked me!

Event handling

Next we'll see a program that tests our ClickListener class. It looks very much like the program we wrote for testing some of the graphics stuff we did last time.

First we create a frame window object so we have a place to put the button that we want to click.

Event handling

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

public class ButtonTester
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();
    final int F_WIDTH = 100;
    final int F_HEIGHT = 60;
    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("Button Tester");
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);




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


Event handling

Next we'll see a program that tests our ClickListener class. It looks very much like the program we wrote for testing some of the graphics stuff we did earlier.

First we create a frame window object so we have a place to put the button that we want to click.

Then we create a button object and add it to the frame, just like the graphical components from last time.

Event handling

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

public class ButtonTester
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();
    final int F_WIDTH = 100;
    final int F_HEIGHT = 60;
    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("Button Tester");
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton button = new JButton("Click me!");
    myframe.add(button);



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


Event handling

Next we'll see a program that tests our ClickListener class. It looks very much like the program we wrote for testing some of the graphics stuff we did earlier today.

First we create a frame window object so we have a place to put the button that we want to click.

Then we create a button object and add it to the frame, just like the graphical components from earlier today.

Finally we create an event listener object called ClickListener and attach it to the button we just made.

Event handling

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

public class ButtonTester
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();
    final int F_WIDTH = 100;
    final int F_HEIGHT = 60;
    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("Button Tester");
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton button = new JButton("Click me!");
    myframe.add(button);
    ActionListener listener = new ClickListener();
    button.addActionListener(listener);

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

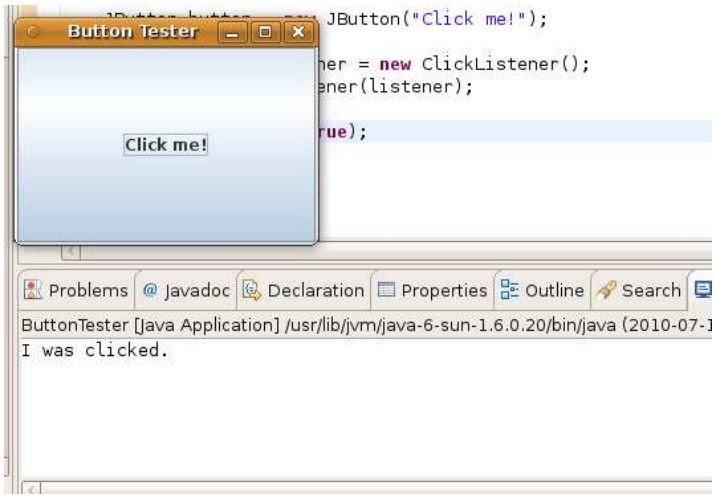

Event handling

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

public class ButtonTester
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();
    final int F_WIDTH = 100;
    final int F_HEIGHT = 60;
    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("Button Tester");
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton button = new JButton("Click me!");
    myframe.add(button);
    ActionListener listener = new ClickListener();
    button.addActionListener(listener);

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

This tells the button “add me to your list of listeners – I want to know what happens to you”

Event handling

```
> java ButtonTester
```



Event handling

A button listener class like ClickListener is likely to be specific to a particular button, so we don't really need it to be widely accessible. We can put the class definition inside the method or class that needs it. So we can put this class:

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

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

inside the main method of the ButtonTester class as an inner class.

Event handling

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent; // note this addition

public class ButtonTester2
{
  public static void main(String[] args)
  {
    JFrame myframe = new JFrame();
    final int F_WIDTH = 100;
    final int F_HEIGHT = 60;
    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("Button Tester");
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton button = new JButton("Click me!");
    myframe.add(button);
```


Event handling

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

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

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

Inner classes

- You can have one class nested in another
- The inner class can use the outer class's variables (and vice-verse)
- If we need to create a really trivial class that is only used by another class, we can use an inner class
- The outer class can have multiple inner classes that implement the interface in different ways

Inner classes

- For example, we can create two buttons that result in different actions when pressed
- So we could have two inner classes: ClickListener and ClickListener2
- We'd then pass an object of ClickListener to the first button and an object of ClickListener2 to the second button
- Here's what it might look like

Anonymous Inner Class

- When we need only a single instance of an inner class, we can create the class and its instance "on the fly" *inside a statement*.
- For readability reasons, this should only be done when the body of the class is very short.
- To create an instance of an anonymous subclass of class `Myclass`:

```
MyClass myObject = new MyClass () {
   public void overriddenFunc() {
      System.out.println("Some message");
   }
};
```


Anonymous Inner Class (cont'd)

- To create an instance of an anonymous class that implements the interface `SomeInterface`:

```
SomeInterface anObject = new SomeInterface () {
   public void interfaceFunc() {
      System.out.println("Some message");
   }
};
```

- Anonymous classes do not have a constructor; they rely on the default constructor of their superclass.

```
public class ButtonTester {

public static void main(String[] args)
{
    JFrame myframe = new JFrame();
    final int F_WIDTH = 100;
    final int F_HEIGHT = 60;
    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("Button Tester");
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JButton button = new JButton("Click me!");
    myframe.add(button);

    button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
          System.out.println("I was clicked.");
    }});

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

Same example as a few slides ago, but now with an anonymous inner class


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

public class FrameViewer
{
    public static void main (String[] args) {
     JFrame myframe = new JFrame();  // make a new JFrame object

      final int F_WIDTH = 300;       // 300 pixels wide
      final int F_HEIGHT = 400;      // 400 pixels high
      myframe.setSize(F_WIDTH, F_HEIGHT);

      myframe.setTitle("My Frame");   // this is optional
      myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      myframe.setLayout( new FlowLayout() );

      JButton myButton = new JButton("Click me!");
      myframe.add(myButton);

      JButton myButton2 = new JButton("Click me too!");
      myframe.add(myButton2);
```

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

public class FrameViewer
{
       public static void main (String[] args) {
       JFrame myframe = new JFrame();  // make a new JFrame object

        final int F_WIDTH = 300;        // 300 pixels wide
        final int F_HEIGHT = 400;       // 400 pixels high
        myframe.setSize(F_WIDTH, F_HEIGHT);

        myframe.setTitle("My Frame");   // this is optional
        myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myframe.setLayout( new FlowLayout() );

        JButton myButton = new JButton("Click me!");
        myframe.add(myButton);

        JButton myButton2 = new JButton("Click me too!");
        myframe.add(myButton2);
```

Now that we have multiple components in our frame, we need a way of getting the layout right

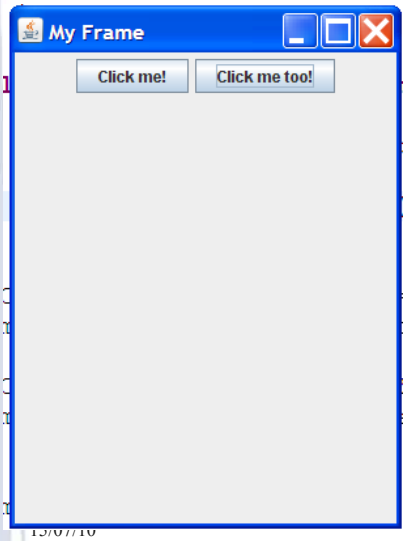

```
  class ClickListener implements ActionListener
     {
       public void actionPerformed(ActionEvent someEvent)
       { System.out.println("I was clicked!"); }
     }

     class ClickListener2 implements ActionListener
     {
         public void actionPerformed(ActionEvent someEvent)
         {System.out.println("Don't you dare click me again!");}
     }
      ClickListener myListener = new ClickListener();
      myButton.addActionListener(myListener);

      ClickListener2 myListener2 = new ClickListener2();
      myButton2.addActionListener(myListener2);


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

We now have two classes implementing ActionListener. They implement actionPerformed() in different ways.

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

 class ClickListener2 implements ActionListener
 {
    public void actionPerformed(ActionEvent someEvent)
    {System.out.println("Don't you dare click me again!");}
 }
  ClickListener myListener = new ClickListener();
  myButton.addActionListener(myListener);

  ClickListener2 myListener2 = new ClickListener2();
  myButton2.addActionListener(myListener2);


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

We then give the first button an object of ClickListener and the second button an object of ClickListener2

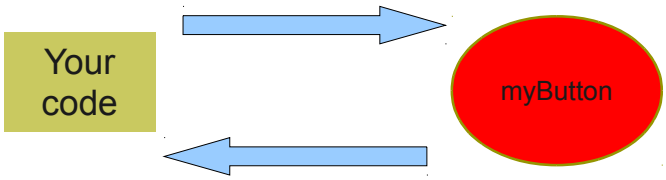
Two Buttons



> I was clicked!
Don't you dare click me again!

Event listeners

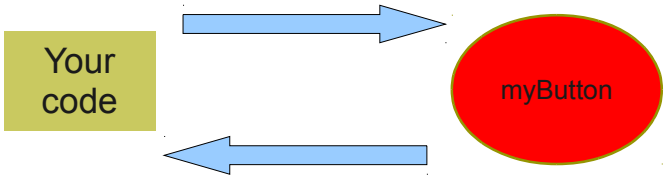
myButton.addActionListener(myListener)
// hey button, I care what happens to you



myListener.actionPerformed()
> I was clicked!

Event listeners

myButton2.addActionListener(myListener2)
// hey button, I care what happens to you



myListener2.actionPerformed()
> Don't you dare click me again!

Inner classes

- We mentioned that inner classes can access the members of the outer class
- Let's play with an example of that
- Rather than just printing out a statement when a button is clicked, let's change the label of the button when it's clicked
- This involves the actionPerformed() method of the inner class object the button of the outer class object

Inner classes

- Let's restructure the code, too
- It doesn't make sense to have all of this in the main method

Changing button labels

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

import java.awt.event.*;

public class FrameViewer
{
    JFrame myframe;
    JButton myButton;

    public static void main (String[] args)
    {   FrameViewer newDemo = new FrameViewer();
        newDemo.go();}
```


Changing button labels

```
public void go()
      {
    myframe = new JFrame();  // make a new JFrame object

    final int F_WIDTH = 300;        // 300 pixels wide
    final int F_HEIGHT = 400;       // 400 pixels high

    myframe.setSize(F_WIDTH, F_HEIGHT);
    myframe.setTitle("My Frame");   // this is optional
    myframe.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    myButton = new JButton("Click me!");

    myframe.add(myButton);
    ClickListener myListener = new ClickListener();
    myButton.addActionListener(myListener);
    myframe.setVisible(true);
      }
```

Changing button labels

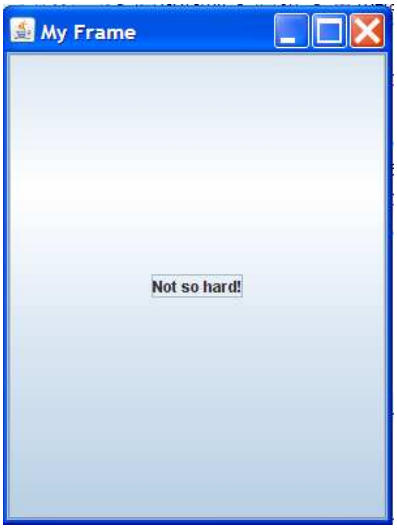
```
class ClickListener implements ActionListener
    {
     public void actionPerformed(ActionEvent
 someEvent)
     {
        myButton.setText("Not so hard!" );

     }
    }
```


Changing button labels

```
class ClickListener implements ActionListener
    {
     public void actionPerformed(ActionEvent
 someEvent)
     {
        myButton.setText("Not so hard!" );

     }
    }
```

This inner class can access the myButton variable of the outer class

Changing button labels



In-Class Exercise I

- Rewrite the ClickListener actionPerformed() method to do the following
  - Set the button text to “Not so hard!”
  - Set the frame title to “Better title!”
  - Double the height and width of the frame
- Hint: the frame has getHeight() and getWidth() methods

Tea break!

Bigger and Smaller

- Hmm, if we keep clicking that button it just keeps getting bigger and bigger
- Maybe we should add another button to make the frame smaller
- We can:
  - Add another button
  - Make another ClickListener
  - Add the new listener to the new button

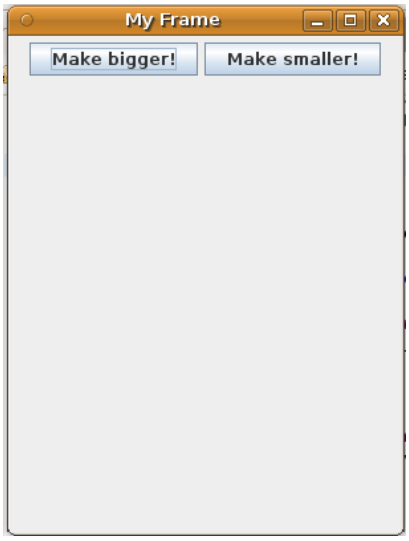
Another ClickListener

```
class ClickListener2 implements ActionListener {

public void actionPerformed(ActionEvent someEvent) {
  System.out.println("I was clicked!");
  myButton2.setText("Ensmallened!");
  myFrame.setTitle("Smaller frame!");
  int currH = myFrame.getHeight();
  int currW = myFrame.getWidth();
  myFrame.setSize(currW/2, currH/2);
}
}
```


Back in our go() method...

```
myButton = new JButton("Make bigger!");
myFrame.add(myButton);
ClickListener myListener = new ClickListener();
myButton.addActionListener(myListener);

myButton2 = new JButton("Make smaller!");
myFrame.add(myButton2);
ClickListener2 myListener2 = new ClickListener2();
myButton2.addActionListener(myListener2);
myFrame.setVisible(true);
```

Bigger and Smaller



Layout Managers

- A *layout manager* is responsible for laying out components (buttons, labels, etc.) inside a container (like a panel)
- Border Layout (default for a JFrame's content pane):
  - has north, south, east, west, center regions

Layout Managers

- Here is how we could create the picture on the previous page:

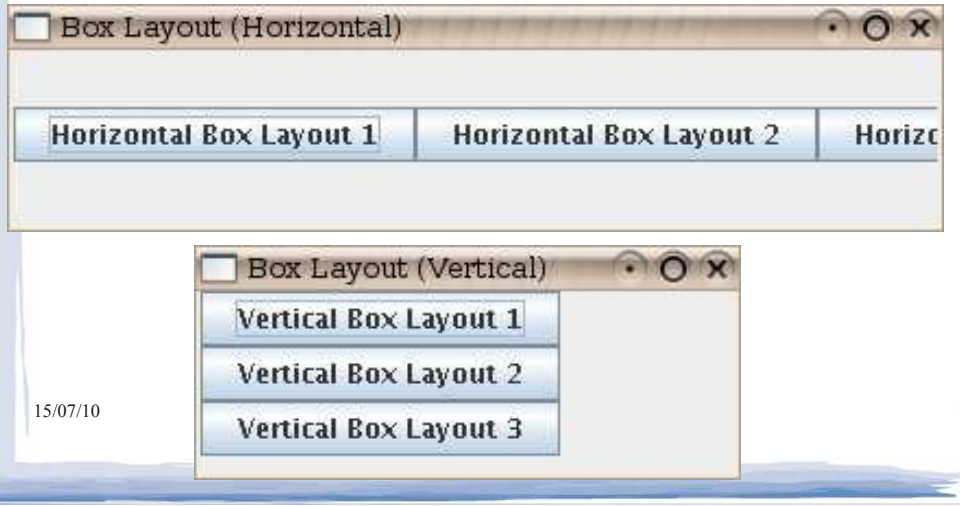
```
public class Border extends JFrame {
  public Border(String title) {
    super(title);
    Container cp = getContentPane();
    cp.setLayout(new BorderLayout());
    cp.add(new JButton("Center"), BorderLayout.CENTER);
    cp.add(new JButton("North"), BorderLayout.NORTH);
    cp.add(new JButton("South"), BorderLayout.SOUTH);
    cp.add(new JButton("East"), BorderLayout.EAST);
    cp.add(new JButton("West"), BorderLayout.WEST);
    pack();
    setVisible(true);
  }
}
```


Layout Managers

- Flow Layout:
  - places components from left to right, with their centres aligned
  - can specify row alignment (LEFT, CENTER, RIGHT)
  - default for JPanel

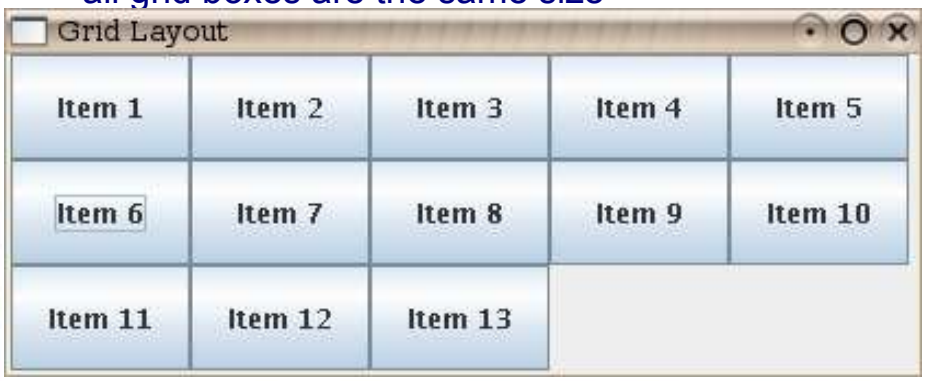
Layout Managers

- Box Layout:
  - places components in a horizontal box from left to right, or in a vertical box from top to bottom.



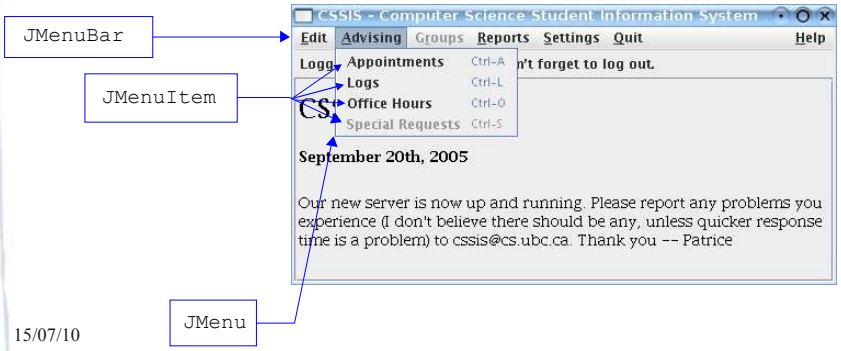
Layout Managers

- Grid Layout:
  - places components in a grid
  - number of rows and columns in grid is specified by client
  - all grid boxes are the same size



Menus

- A menu consists of:
  - A `JMenuBar`, containing one or more
    - `JMenu` objects, each of which contains one or more
      - `JMenuItem` objects



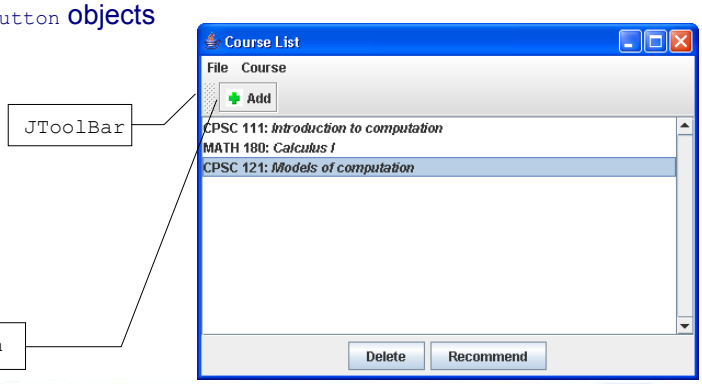
Menus

- **Here is part of the code used to create the menu on the previous page:**

```
JMenuBar menubar = new JMenuBar();
JMenu advisingMenu = new JMenu("Advising");
advisingMenu.setMnemonic(KeyEvent.VK_A);
menubar.add(advisingMenu);

JMenuItem apptMenuItem = new JMenuItem("Appointments");
apptMenuItem.setAccelerator(
   KeyStroke.getKeyStroke(KeyEvent.VK_A,ActionEvent.CTRL_MASK));
advisingMenu.add(apptMenuItem);

...

JMenuItem specMenuItem = new JMenuItem("Special Requests");
specMenuItem.setAccelerator(...);
specMenuItem.setEnabled(false);
advisingMenu.add(specMenuItem);

myFrame.setJMenuBar(menubar);
```

Toolbars

- A toolbar provides the user with easy access to frequently used actions and often duplicates actions available via menus.
- A toolbar consists of:
  - A `JToolBar`, containing one or more
    - `JButton` objects



Dialogs

- A dialog window is used to present data to the user or to get data from the user.
- Dialog windows can either be *modal* or *non-modal*
  - When a *modal* dialog opens
    - It blocks input to other windows associated with the same application.
    - The user must respond to the dialog before doing anything else within that application.
  - When a *non-modal* dialog opens
    - input to other windows is not blocked.
    - The user can continue to access other windows associated with the same application.

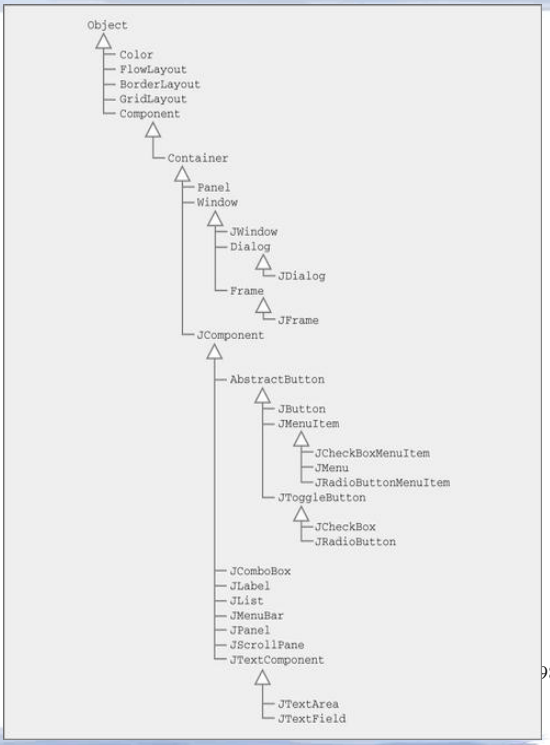
Dialogs

- `JDialog` objects work more or less the same way as `JFrame` objects.
- The `JOptionPane` class provides several methods to create standard dialogs. For instance:

```
JOptionPane.showMessageDialog(
                null,
                "You love CPSC 211",
                "Message",
                JOptionPane.INFORMATION_MESSAGE);


int result = JOptionPane.showConfirmDialog(
               null,
               "Do you like Swing?",
               "Confirm",
               JOptionPane.YES_NO_OPTION,
               JOptionPane.QUESTION_MESSAGE);
```


Swing and AWT classes

For your reference:



Adapter Classes

- Implementing all the methods of a listener can be tedious if we only care about one of them.
  - The `MouseListener` interface, for example, defines five methods.
  - We don't want to have to provide implementations for all of them if we're interested only in the `mouseClicked` event.
- Swing provides adapter classes for some listeners: the adapter provides a default implementation (an empty body) for all the methods.
- The adapter can be subclassed to override the methods that we care about, without having to implement the other ones.

Adapter Classes

- Examples:
  - **`MouseAdapter` implements `MouseListener` interface**
  - **`WindowAdapter` implements `WindowListener` interface**
- Code Example:

```
frame.addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
          confirmExitIfUnsavedData();
       }});
```

Note the anonymous inner class in use here!

Focus Events

- At any point in time, exactly one component will receive key events (events generated when a key is pressed or released).
- This component is said to have the *keyboard focus*.
- Focus events occur when:
  - The user clicks on a component that can acquire keyboard focus, for instance a text field (`focusGained` event).
  - The user clicks somewhere else, causing the component to lose keyboard focus (`focusLost` event).
  - The user hits the `tab` key to move from one text field to another.
  - And more. . .

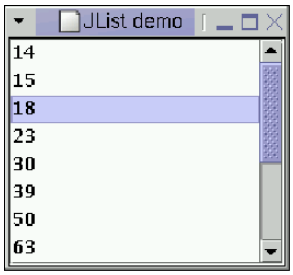
Example: Focus Listener

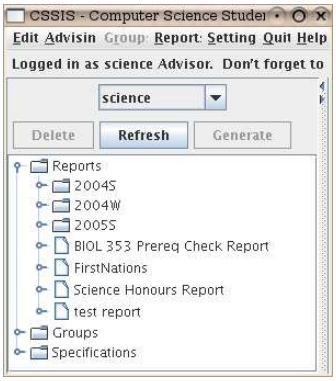
In an application, the following focus listener might be registered with the text fields in a dialog so that when a text field gains focus, all the text within that field is selected for the user. This makes it easy for a user to replace all the text in a field if they revisit it.

```
private class TextFieldListener implements FocusListener
{
    public void focusGained(FocusEvent event)
    {
      JtextField tf = (JtextField) event.getSource();
      tf.selectAll();
    }

    public void focusLost(FocusEvent event)
    {
       // Do nothing when we lose focus.
    }
}
```

Components and Models

- We use a Component to display data in a particular way. For example, the `JList` and a `JTree` classes are used to display a list (or tree) of items that can be selected by the user.





Components and Models (cont'd)

- In general, we want to separate the data (the model) from the way the data is displayed (the view).
  - This makes it easy to change the way the data is presented to the user (the view) without making any changes to the way the data itself is represented (the model).
  - It allows the application to manipulate the data without regard to the way the data is displayed.
- The more complicated Swing components such as `JList`, `JTree` and `JTable` do not store the data that they display. Instead, it is stored in another object called a model.

Components and Models (cont'd)

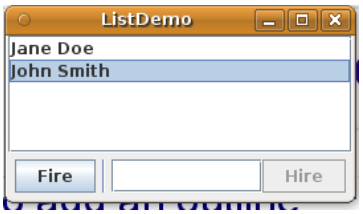
- How do these components handle different types of data?
  - Swing defines an interface that defines the behaviour (methods) that the model must support.
  - The Component accesses the data *only* through the model's interface.
  - The class to be displayed in the component implements this interface, in addition to its "own functionality".

Example: Components and Models

For example, Java provides an implementation of the ListModel interface, called DefaultListModel
DefaultListModel provides implementations for methods like addElement(), removeElement() and getElementAt()
When we create a GUI using lists, we can store our data in the DefaultListModel
For example...

ListDemo

- We have a list of employees
- We can select people to fire
- Or type in names of people to hire



A snippet of ListDemo

```
public class ListDemo extends JPanel implements ListSelectionListener {
    private JList list;
    private DefaultListModel listModel;
    . . .
    public ListDemo() {
        super(new BorderLayout());

        listModel = new DefaultListModel();
        listModel.addElement("Jane Doe");
        listModel.addElement("John Smith");
        listModel.addElement("Kathy Green");

        //Create the list and put it in a scroll pane.
        list = new JList(listModel);
```

Create a new DefaultListModel

A snippet of ListDemo

```
public class ListDemo extends JPanel implements ListSelectionListener {
    private JList list;
    private DefaultListModel listModel;
    . . .
    public ListDemo() {
        super(new BorderLayout());

        listModel = new DefaultListModel();
        listModel.addElement("Jane Doe");
        listModel.addElement("John Smith");
        listModel.addElement("Kathy Green");

        //Create the list and put it in a scroll pane.
        list = new JList(listModel);
```

Add some data to the model

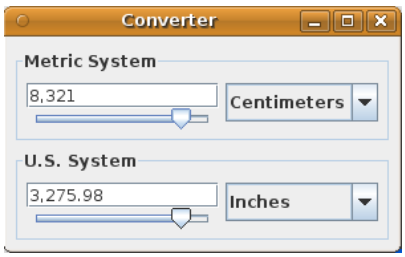
A snippet of ListDemo

```
public class ListDemo extends JPanel implements ListSelectionListener {
    private JList list;
    private DefaultListModel listModel;
    . . .
    public ListDemo() {
        super(new BorderLayout());

        listModel = new DefaultListModel();
        listModel.addElement("Jane Doe");
        listModel.addElement("John Smith");
        listModel.addElement("Kathy Green");

        //Create the list and put it in a scroll pane.
        list = new JList(listModel);
```

Create a list associated with the data in that model

Another Model Example

- Two sliders
- Only one model controls the value of the data



Radio Buttons

```
public class RadioTester {

JFrame myframe;

JRadioButton orangeButton;

JRadioButton blueButton;

ButtonGroup group;

JPanel panel;


public void go(){

myframe = new JFrame();

final int F_WIDTH = 200;

final int F_HEIGHT = 200;

myframe.setSize(F_WIDTH, F_HEIGHT);

myframe.setTitle("Radio Button
Tester");

myframe.setDefaultCloseOperation(JFram
e.EXIT_ON_CLOSE);


// continued -->
```

- Let's create a simple GUI with some radio buttons
- Radio buttons are mutually exclusive (you can only select one)
- First we create the frame and set its size

Radio Buttons

```
orangeButton = new JRadioButton("orange");
blueButton = new JRadioButton("blue");
group = new ButtonGroup();
group.add(orangeButton);
group.add(blueButton);
panel = new JPanel();
panel.add(orangeButton);
panel.add(blueButton);
myframe.add(panel);
myframe.setVisible(true);
}
public static void main(String[] args) {
RadioTester rt = new RadioTester();
rt.go();
}
```

Two new buttons

Radio Buttons

```
orangeButton = new JRadioButton("orange");
blueButton = new JRadioButton("blue");
group = new ButtonGroup();
group.add(orangeButton);
group.add(blueButton);
panel = new JPanel();
panel.add(orangeButton);
panel.add(blueButton);
myframe.add(panel);
myframe.setVisible(true);
}
public static void main(String[] args) {
RadioTester rt = new RadioTester();
rt.go();
}
```

We need to associate the buttons with one another so that only one can be selected at a time

Radio Buttons

```
orangeButton = new JRadioButton("orange");
blueButton = new JRadioButton("blue");
group = new ButtonGroup();
group.add(orangeButton);
group.add(blueButton);
panel = new JPanel();
panel.add(orangeButton);
panel.add(blueButton);
myframe.add(panel);
myframe.setVisible(true);
}
public static void main(String[] args) {
RadioTester rt = new RadioTester();
rt.go();
}
```

Create a new panel, add each button to the panel, and then add the whole panel to the frame, and make it visible.

Radio Buttons

```
orangeButton = new JRadioButton("orange");
blueButton = new JRadioButton("blue");
group = new ButtonGroup();
group.add(orangeButton);
group.add(blueButton);
panel = new JPanel();
panel.add(orangeButton);
panel.add(blueButton);
myframe.add(panel);
myframe.setVisible(true);
}
public static void main(String[] args) {
RadioTester rt = new RadioTester();
rt.go();
}
```

In-Class Exercise II

- Edit this code in the following ways:
  - Add another radio button “green”
  - Associate each button with an action listener
  - When somebody selects one of the buttons, the entire background of the panel should change to that color
  - Hint: JPanel has a setBackground() method that can be called like `setBackground(Color.orange)`

Exercises

2nd Ed:

Chapter 14

Questions P14.1, P14.4

3rd & 4th Eds:

Chapter 18

Questions P18.1, P18.4

Learning Goals Review

- describe the event driven model
- describe and apply scoping rules that apply to the use of inner classes
- write code that uses inner classes (including anonymous inner classes) to handle events raised by GUI elements
- define and use appropriate data models for certain components like JLists to separate data from their display