CSE 11: Lecture 12

- ✔ Layout Managers
- ✔ Event-driven programming
- ✔ Listeners and events
- ✔ ActionEvent and WindowEvent
- ✔ ActionListener, WindowListener, and WindowAdapter

(Reading: Savitch, Ch. 12)

Midterm Exam #2 will be Tues Nov 18, during lecture time.

Place: here

Closed-book, closed-notes, no calculators. Bring picture ID!

Coverage: Chapters 1-5, 7-9, 12 and 14, Lectures 1-13, Assignments P1-P6

A practice midterm is available online (PDF format)

Lecture notes are available online

Midterm review will be: Monday Nov 17 6-7pm Center 105

Midterm #2 exams will be returned: Monday Nov 24 6-7pm place TBA

Layout managers

- ✔ If you add a Component to a Container, how do they get arranged in the Container?
- ✔ In Java GUI programming, this is the job of a `java.awt.LayoutManager` object
- ✔ There are three main types of `LayoutManager` we'll talk about (there are others):
  - ✗ `FlowLayout`
  - ✗ `GridLayout`
  - ✗ `BorderLayout`
- ✔ Every type of Container has a default layout manager, which will be used if you don't specify another one...
- ✔ ...or, you can tell a Container to use a particular layout manager of your choice
- ✔ ...or, you can tell a Container to use no layout manager at all

Setting a LayoutManager

- ✔ Of the Containers we will talk about:
  - ✗ A `JFrame`'s content pane uses `BorderLayout` as its default layout manager
  - ✗ A `JPanel` uses `FlowLayout` as its default layout manager
- ✔ To associate another LayoutManager with a Container so that it governs where Components are placed in the Container, create a LayoutManager of the appropriate type and pass it to the Container's `setLayout` method:

```
JFrame f = new JFrame();
Container c = f.getContentPane();
c.setLayout(new FlowLayout());  //use FlowLayout, not BorderLayout

JPanel p = new JPanel();
p.setLayout(new BorderLayout());//use BorderLayout, not FlowLayout
```

- ✔ Different kinds of LayoutManager work differently, as we will soon see.

On not using a Layout manager

- ✔ It is possible to size and place Components in a Container without using a LayoutManager
- ✔ To do this, call `setLayout(null)` for the Container, and then explicily call the setSize() and setLocation() methods of each Component you add, to size it an place it within the Container
- ✔ HOWEVER, this is usually not such a good idea
- ✔ The explicit sizing and positioning of Components you choose may work well for the initial size of the Container, but...
  - ✗ if the Container is resized by the user, you probably have to recompute all the Component locations and sizes to get the result to look right
- ✔ The LayoutManagers take care of all this repositioning and resizing for you, and so you should try to use them if possible
- ✔ If an existing LayoutManager does not meet your needs, you could also customize it by subclassing and overriding methods to do what you want

FlowLayout

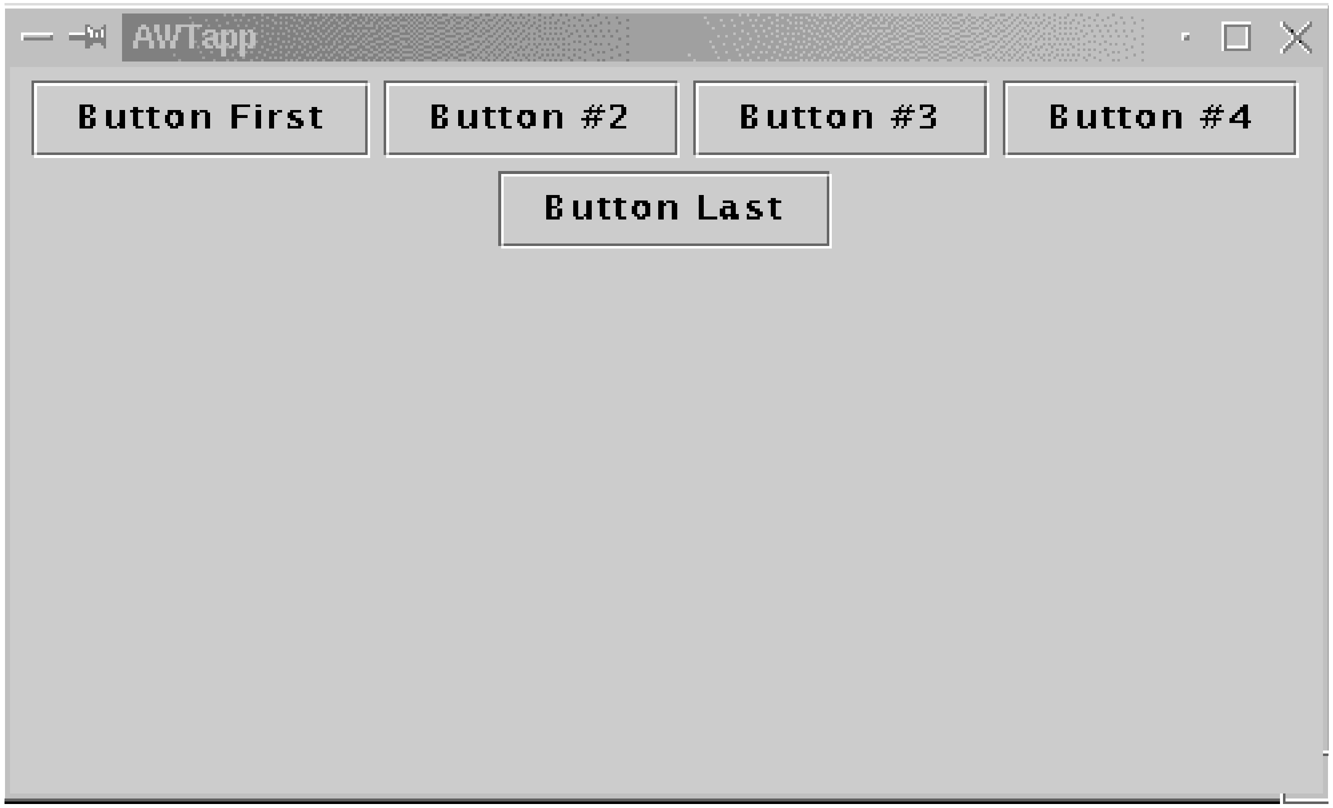
- A FlowLayout manager arranges components in a container left to right and top to bottom, as they are added. Here we set up a JFrame to use a FlowLayout, and add five Buttons to it:

```
JFrame w = new JFrame();  // create a JFrame object
w.setSize(500,300);     // make it 500 pixels wide, 300 high
Container p = w.getContentPane();
p.setLayout(new FlowLayout()); // make it use FlowLayout
// create some JButtons
JButton b1 = new JButton("Button First");
JButton b2 = new JButton("Button #2");
JButton b3 = new JButton("Button #3");
JButton b4 = new JButton("Button #4");
JButton b5 = new JButton("Button Last");

// add them to the Frame
p.add(b1); p.add(b2); p.add(b3);
p.add(b4); p.add(b5);
w.setVisible(true);
```

- (FlowLayout is default manager for JPanels.) This creates the window shown next...

A window with JButtons using FlowLayout

- When run, the code on the previous slide will display a window like the one shown here



- If a Container is resized, the FlowLayout manager takes care of repositioning the Components in the Container as best it can, *without resizing the Components*

GridLayout

- ✔ A GridLayout manager arranges components into cells in a grid
- ✔ The grid has a number of rows, and a number of columns
- ✔ These dimensions of the grid can be specified as arguments to the constructor
- ✔ The GridLayout manager then arranges components into cells in the grid, left-to-right and top-to-bottom as they are added

  ... that is, the first component added goes in the first row, first column; the second component goes in the first row, second column; etc.

  (... Note: GridLayout will decide for itself to use a different number of rows or columns if you add too many or two few components!)

GridLayout: example

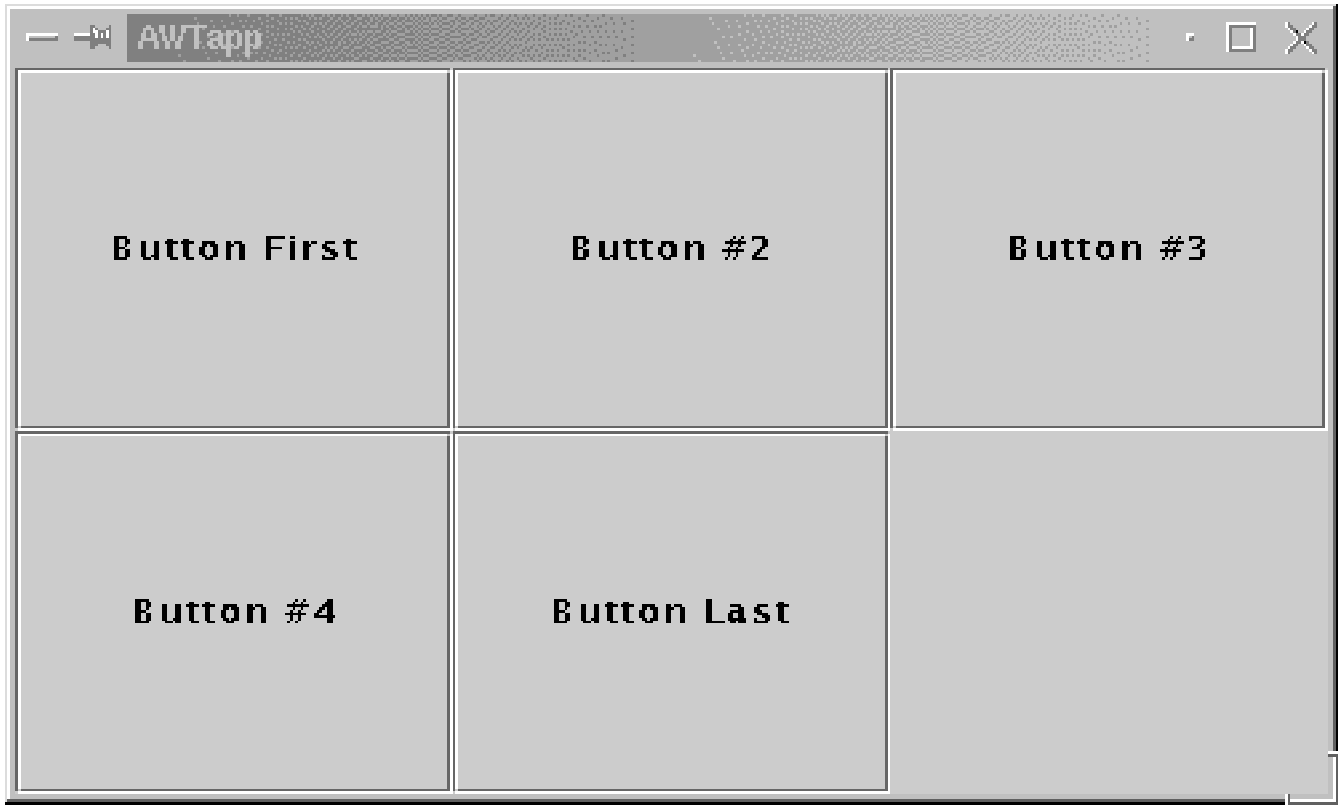
- A GridLayout manager arranges components in a container in a grid with a certain number of rows and columns. Here we set up the Frame to use a 2-by-3 GridLayout, and add five Buttons to it:

```
JFrame w = new JFrame();  // create a Frame object
w.setSize(500,300);     // make it 500 pixels wide, 300 high
// make it use GridLayout, 2 rows and 3 columns
Container c = w.getContentPane();
c.setLayout(new GridLayout(2,3));
// create some JButtons
JButton b1 = new JButton("Button First");
JButton b2 = new JButton("Button #2");
JButton b3 = new JButton("Button #3");
JButton b4 = new JButton("Button #4");
JButton b5 = new JButton("Button Last");

// add them to the Frame
c.add(b1); c.add(b2); c.add(b3);
c.add(b4); c.add(b5);
w.setVisible(true);
```

- This creates the window shown next...

A window using GridLayout

- When run, the code on the previous slide will display a window like the one shown here



- If the window is resized, the GridLayout manager takes care of resizing the components as best it can, but it *keeps them in their relative positions and does not change the number of rows or columns*

BorderLayout

- A BorderLayout manager arranges components into 5 regions of the container: North, South, East, West, and Center
- Using BorderLayout requires using a two-argument form of the add method:
  - the first argument is the Component to add
  - the second argument is a named constant specifying the region: **`BorderLayout.NORTH`**, **`BorderLayout.SOUTH`**, **`BorderLayout.EAST`**, **`BorderLayout.WEST`**, or **`BorderLayout.CENTER`**
- If you use the one-argument form of the add method (not specifying a region) **`BorderLayout.CENTER`** is used
- If a region does not get a component, neighboring regions will expand to fill it
- If a region gets more than one component, only the last one added will appear
- BorderLayout is the default LayoutManager for the JFrame class: If you do not specify one, this is the one you get for your top-level JFrame

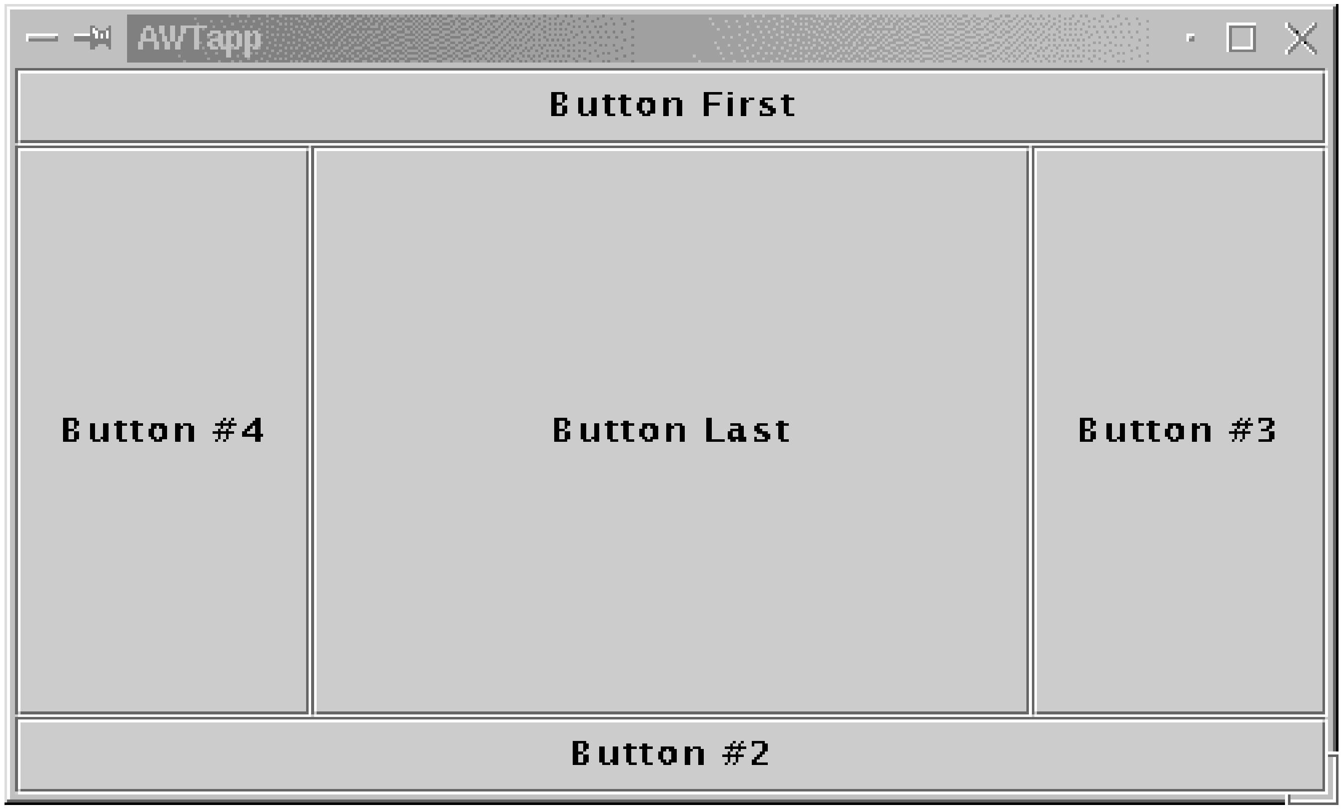
BorderLayout: example with JButtons

```
JFrame w = new JFrame();  // create a Frame object
Container c = w.getContentPane();
w.setSize(500,300);     // make it 500 pixels wide, 300 high
// by default, JFrame content panes use BorderLayout...
// create some Buttons
JButton b1 = new JButton("Button First");
JButton b2 = new JButton("Button #2");
JButton b3 = new JButton("Button #3");
JButton b4 = new JButton("Button #4");
JButton b5 = new JButton("Button Last");
// add them
c.add(b1,BorderLayout.NORTH);
c.add(b2,BorderLayout.SOUTH);
c.add(b3,BorderLayout.EAST);
c.add(b4,BorderLayout.WEST);
c.add(b5);  // same as saying BorderLayout.CENTER
w.setVisible(true);
```

✔ This creates the window shown next...

A window with Buttons using BorderLayout

- When run, the code on the previous slide will display a window like the one shown here



- If the window is resized, the BorderLayout manager takes care of resizing the components as best it can, *keeping them in their relative regions*

Absolute positioning of Components

- Sometimes you may want to do your own sizing and placing of Components in a Container...
  - usually an existing layout manager will do what you want, but not always

- Every Component has these methods for setting its location or size
  - Locations are specified in the coordinate system of the Container

```
// Moves this component to a new location.
public void setLocation(int x, int y)

// Resizes this component so that it has width width and height.
public void setSize(int width, int height)

// Moves and resizes this component.
public void setBounds(int x, int y, int width, int height)
```

- For these to be effective, you will want to disable the layout manager for the Container the Component will be added to:

```
theContainer.setLayout(null)
```

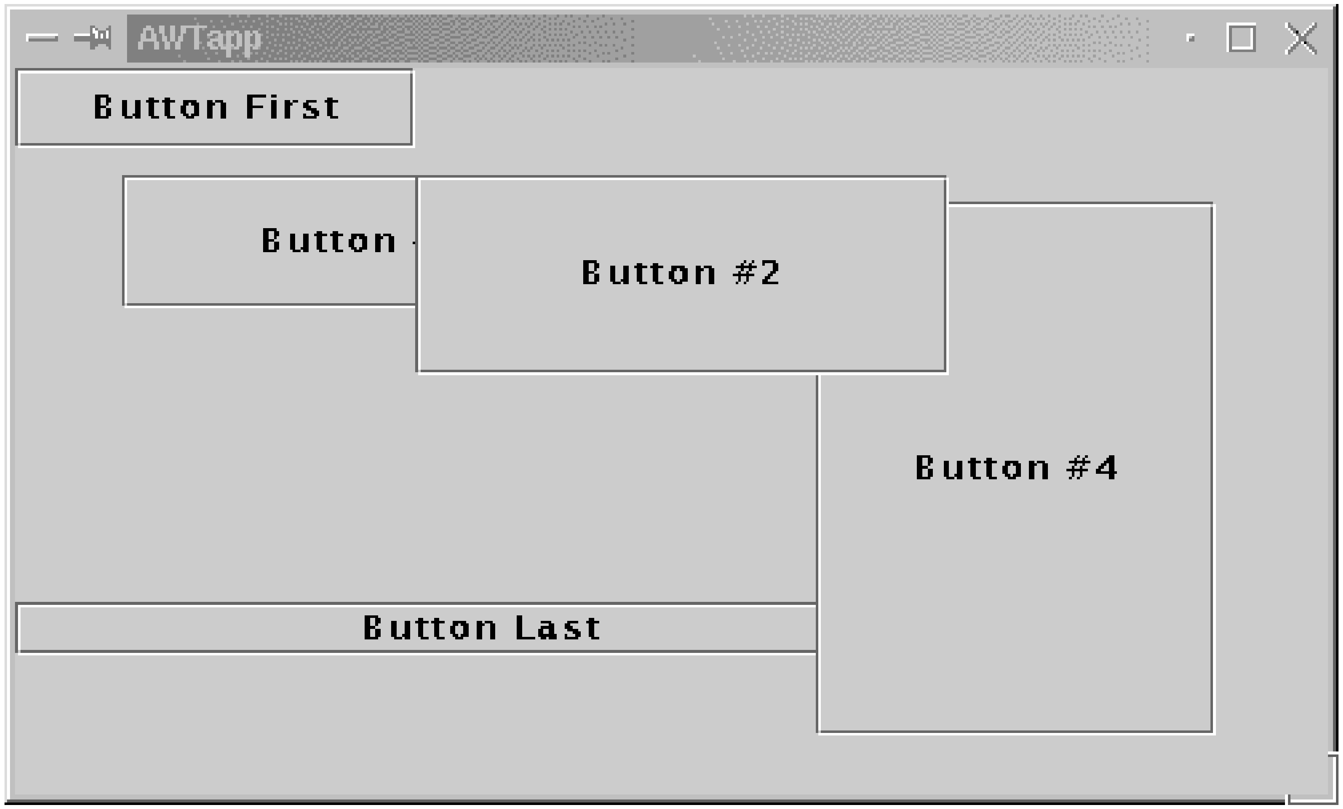
No layout manager: example with JButtons

```
JFrame w = new JFrame();  // create a Frame object
Container c = w.getContentPane();
w.setSize(500,300);      // make it 500 pixels wide, 300 high
c.setLayout(null);       // do it myself, thanks
JButton b1 = new JButton("Button First");
JButton b2 = new JButton("Button #2");
JButton b3 = new JButton("Button #3");
JButton b4 = new JButton("Button #4");
JButton b5 = new JButton("Button Last");

b1.setLocation(0,0); b1.setSize(150,30);
b2.setBounds(150,40,200,75);
c.add(b1);    c.add(b2);      c.add(b3);

b3.setBounds(40,40,180,50);
b4.setSize(150,200);
c.add(b4);  b4.setLocation(300,50);
c.add(b5);  b5.setBounds(0,200,350,20);
w.setVisible(true);
```

✔ This creates the window shown next...

A window with Buttons using no layout manager

- When run, the code on the previous slide will display a window like the one shown here



- Since there is no layout manager, if the window is resized, the components will not be of resized or repositioned

Flow of control in programs

- In the programs we’ve seen so far, the statements in the program determine the order in which things occur -- user input plays a relatively small role in this sequencing
  - The program starts executing with the first statement in the main() method
  - Statements are executed in sequence, subject to the semantics of the statement
  - Some statements involve interesting flow of control:
    - if, if-else, switch statements cause conditional branching
    - while, do-while, for statements cause iterative looping
    - method calls cause execution of statements in the bodies of other methods
    - return, break, continue, and throw statements cause jumps in flow
  - If user input is needed, it is prompted for, and the program waits until it gets the input
  - Conditional branching and iteration can depend on user input
- Typical GUI application programming takes a somewhat different approach...

Event-driven programming

- In typical GUI programming, events generated by the user or by GUI objects themselves play a large role in the sequence of execution. This is called *event-driven* programming... The flow of control is along these lines:
  - The program starts executing with the first statement in the main() method
  - The main() method can be quite simple: An object of a class derived from JFrame is created, and made visible
  - The constructor of the JFrame initializes it, placing components within the JFrame and registering *event listener* objects with the components or the JFrame itself
  - The application is then basically dormant, until the user interacts with the JFrame itself, or its components (e.g. text fields, buttons, menus)
  - Interaction with the JFrame or a component (such as pushing a button) generates or fires an *event*
  - When an event fires, the event listeners registered with that component for that event are *notified* of that event so they can handle it
    - notification involves: calling certain methods of the registered listeners
  - When the listeners are done handling the event, the application waits for more user-generated events

Structure of a typical Java GUI application

✔ ... can look something like this:

```
public class MyApp extends JFrame {
   public MyApp()  {  // constructor
      setSize(...); // set size of the application JFrame
      Container c = getContentPane();
      c.setLayout(...);  // specify which layout manager to use
      // create components that will be contained in the JFrame
      ...
      // add components to the frame
      c.add(...);
      c.add(...);
      ...
      // register listener objects with the components or JFrame
      ...
}
   public static void main(String args[]) {
      // create the application JFrame and make it visible
      (new MyApp()).setVisible(true);
   }
}
```

Registering listeners with components

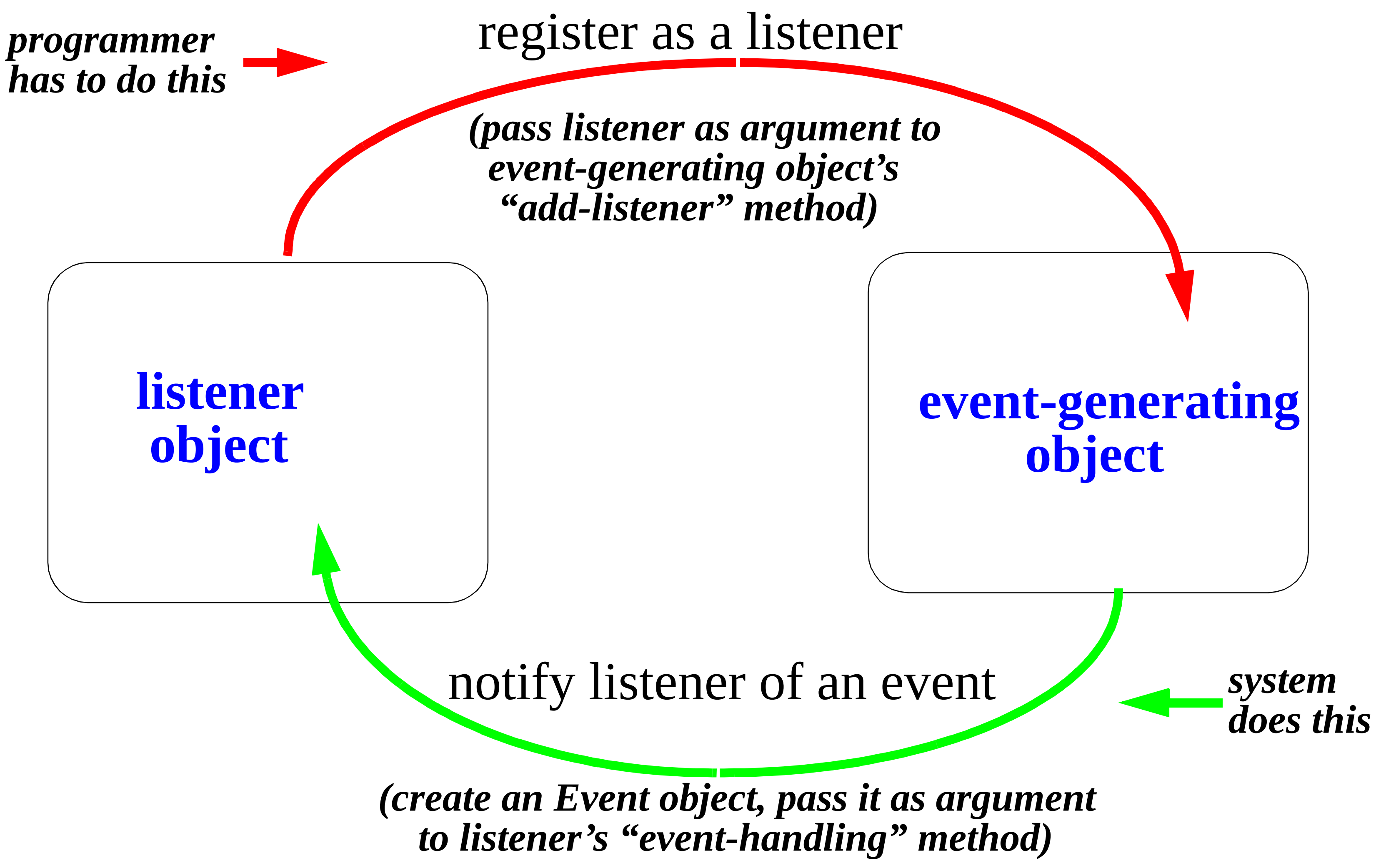
- All of that we have covered already...
  - ... except: registering listener objects!

- Once a listener object is registered with an event-generating object, the listener object will be sutomatically notified of certain events that happen to (are “fired by”) the event-generating object

- This “notification” consists of calling a certain method of the listener object
  - this method is passed an event object, which represents the event that occurs
  - you need to define these methods so that they do the right thing (handle the event appropriately), whatever that means for your application

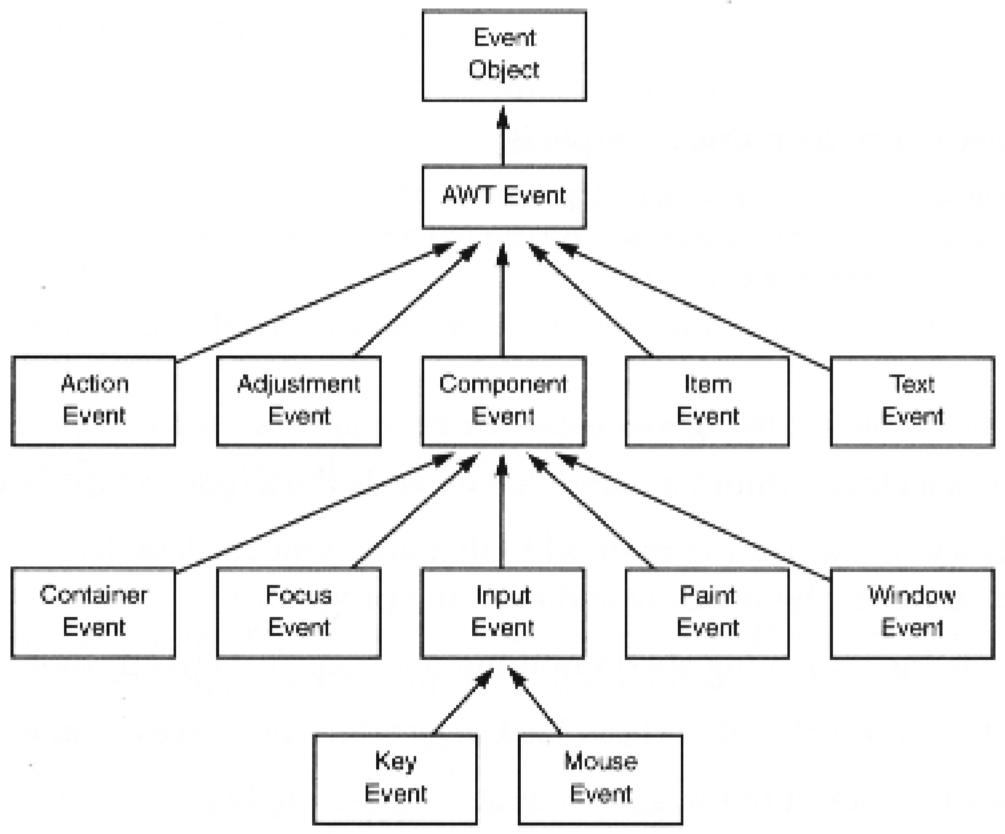
- So we need to look at:
  - what kinds of event objects are there?
  - what kinds of listener objects are there? and what methods must they have?
  - what kinds of event-generating objects can you register a listener with? and how do you do that?

Registration, notification, etc.: a picture


programmer
has to do this
register as a listener
(pass listener as argument to
event-generating object's
"add-listener" method)
listener
object
event-generating
object
notify listener of an event
system
does this
(create an Event object, pass it as argument
to listener's "event-handling" method)

What kinds of events are there?

- ✔ In Java, events are objects (what did you expect?)
- ✔ Here is the event class hierarchy in the `java.awt.event` package:



- ✔ The ones we will talk about are: `ActionEvent` and `WindowEvent`

ActionEvent

- ✔ An ActionEvent is an important, “high-level” kind of GUI event in the AWT
- ✔ An ActionEvent can be generated by several types of JFC object:
  - ✗ a JButton, when it is is pushed (i.e., is clicked on with the mouse)
  - ✗ a JTextField, when someone types <enter> in it
  - ✗ a MenuItem, when it is selected (we haven’t talked about Menus yet)
  - ✗ a List item, when it is double-clicked (we haven’t talked about Lists yet either)
- ✔ Every ActionEvent object has the instance method **`getActionCommand()`**that returns a String which is:
  - ✗ the label of the JButton, if the event came from a Button push
  - ✗ the current text in the JTextField, if it was a JTextField that generated it
  - ✗ the label of the MenuItem, if the event came from a MenuItem
  - ✗ the text of the List item, if that’s what caused it
- ✔ Every EventObject object (ActionEvent is a subclass) has the instance method **`getSource()`** that returns an Object pointer to the object that generated the event

ActionListener

- ✔ An object that is to be registered with a component to listen for ActionEvents fired by the component must be an instance of a class that implements the **`ActionListener`** interface
- ✔ A Java interface is like a class -- it defines a type -- except that an interface has no definitions for any of its methods! It has only method headers
- ✔ A class that implements an interface declares itself to do so, and *must* have a definition for *every* method in the interface
- ✔ The ActionListener interface specifies one method with this header:

```
public interface ActionListener {
   public void actionPerformed(ActionEvent e);
}
```
... so any object registered to listen for ActionEvents must define this method

- ✔ When a component generates an ActionEvent...
  - ✗ each ActionListener object that has been registered with the component will have its **`actionPerformed`** method called
  - ✗ this method will be passed the ActionEvent object correpsonding to the event
  - ✗ in the body of the method, the **`getActionCommand`** and **`getSource`** methods of the ActionEvent object can be called to get useful information

Registering an ActionListener with a Component

- Suppose Foo is a class that implements the ActionListener interface:

```
public class Foo implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      System.out.println("Got " + e.getActionCommand());
   }
}
```

- Now a Foo object is-A ActionListener, and we can use a Foo object to listen for ActionEvents that happen to a Button, TextField, MenuItem, or List. To do register the Foo object to do that, we use the `addActionListener` method of the component:

```
Button b = new Button("Ouch!");
b.addActionListener(new Foo());
// add b to a Frame, make the Frame appear
...
```

- ... now when the button labeled Ouch! is clicked on, the actionPerformed method of its ActionListener is called, and... ??

WindowEvent

- A WindowEvent represents an important event that happens to a Window as a whole (recall that in the AWT hierarchy, JFrame is a subclass of Window)

- These things will fire a WindowEvent:
  - the window has opened (i.e. been created)
  - the window has closed (i.e. been destroyed)
  - the user is trying to close the window (i.e. by hitting the “Close” button on the titlebar)
  - the window has become active (i.e. been made visible)
  - the window has become inactive (i.e. been made invisible)
  - the window becomes iconified (i.e. minimized)
  - the window becomes de-iconified (i.e. restored to original size)

- WindowEvent objects contain information but (unlike ActionEvent objects) it is not often necessary to access it. We’ll see why...

WindowListener

- ✔ An object that is to be registered with a component to listen for WindowEvents must be an instance of a class that implements the `WindowListener` interface
- ✔ The WindowListener interface has seven methods, one for each sort of thing that can fire a WindowEvent:

```
public interface WindowListener {
   public  void windowActivated(WindowEvent e);
   public  void windowClosed(WindowEvent e);
   public  void windowClosing(WindowEvent e);
   public  void windowDeactivated(WindowEvent e);
   public  void windowDeiconified(WindowEvent e);
   public  void windowIconified(WindowEvent e);
   public  void windowOpened(WindowEvent e);
}
```

... so any object registered to listen for WindowEvents must implement *all* these methods

- ✔ When a Window generates a WindowEvent..
  - ✗ each WindowListener object that has been registered with the Window will have one of these methods called, depending on what happened
  - ✗ this method will be passed the WindowEvent object correpsonding to the event

The WindowAdapter class

- ✔ You can define a class that implements WindowListener, but this requires defining 7 methods!
- ✔ Pretty often, you only really want to deal with one of them: `windowClosing`
- ✔ To save some typing, there is an AWT class called WindowAdapter which is defined this way:

```
public abstract class WindowAdapter implements WindowListener {
 public void windowActivated(WindowEvent e) {}
 public void windowClosed(WindowEvent e) {}
 public void windowClosing(WindowEvent e) {}
 public void windowDeactivated(WindowEvent e) {}
 public void windowDeiconified(WindowEvent e) {}
 public void windowIconified(WindowEvent e) {}
 public void windowOpened(WindowEvent e) {}
}
```

- ✔ ... All the methods are stubs!
- ✔ The point is: If you only want to define one or two of these methods, it is easier to subclass WindowAdapter and override those one or two methods, than it is to implement WindowListener

Registering a WindowListener with a component

- ✔ Suppose Baz is a class that extends WindowAdapter (and so implements WindowListener):

```
public class Baz extends WindowAdapter {
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}
```

- ✔ Now we can use a Baz object to listen for WindowEvents that are fired by a JFrame. To register the Baz object to do that, we use the `addWindowListener` method:

```
public class MyApp extends JFrame {
   public MyApp()  {  // constructor
      ...
      addWindowListener(new Baz());
      ...
   }
```

- ✔ ... now when the user tries to close the application Frame, the windowClosing method of its WindowListener is called, and... ??

Implementing listener interfaces

- ✔ In a typical Java GUI application, there will be a user-defined class that extends JFrame

```
public class MyApp extends JFrame {
```

... creating an instance of this class and making it visible will launch the application

- ✔ For the application to be interesting, at least some components contained in the JFrame should have ActionListeners registered with them to handle events

- ✔ For the application to be well-behaved, the JFrame itself should have a WindowListener registered with it to kill the application when the window is being closed

- ✔ Q: Do you NEED to create separate classes for these listeners?

- ✔ A: No; the application class itself can implement ActionListener and WindowListener, and "be its own listener"

.... (What you CANNOT do is have the application class extend WindowAdapter, because it already extends JFrame, and Java has only single inheritance...!)

A JFrame as its own listener

```
public class MyApp extends JFrame
      implements ActionListener, WindowListener {

   // implement actionPerformed, from ActionListener
   public void actionPerformed (ActionEvent e) {
      if(e.getActionCommand().equals("Ouch!")) {
         // it was the Ouch! button... handle it
      } else
      if(e.getActionCommand().equals("Start")) {
         // it was the Start button... handle it
      } else
       ...   // etc.
   }
```

A Frame as its own listener, cont'd

```
// implement windowClosing, from WindowListener
public void windowClosing(WindowEvent e) {
   System.exit(0);
}
// need to implement the other 6 WindowListener methods too!
// they can just be stubs, with empty bodies...

public MyApp()  {  // constructor, does everything
   ...
   // create a button
   JButton ouch = new JButton("Ouch!");
   // give it an ActionListener
   ouch.addActionListener(this);
   // create another button
   JButton start = new JButton("Start");
   // give it an ActionListener
   start.addActionListener(this);
   ...
   // give the whole JFrame a WindowListener
   addWindowListener(this);
```

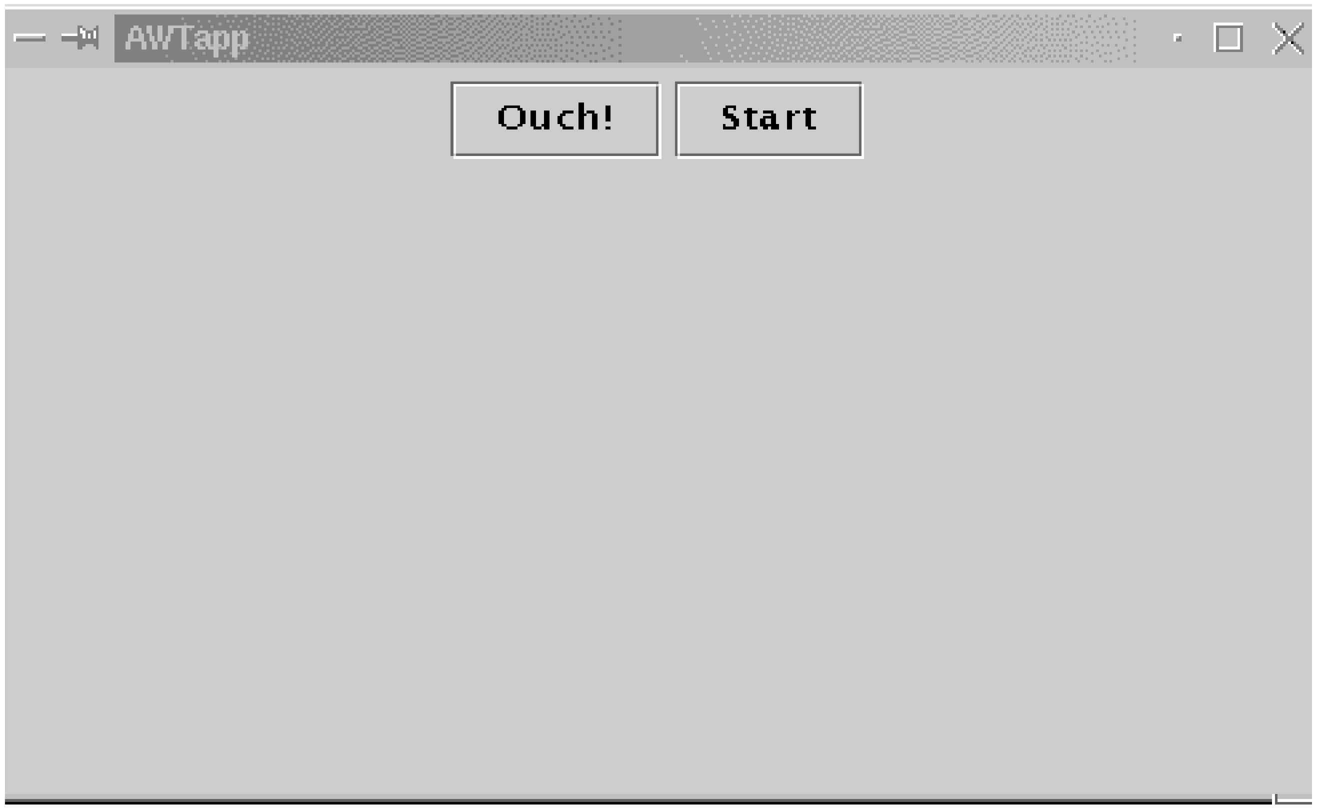
A JFrame as its own listener, still cont’d

```
      // continuing the ctor
      ...
      // set size, choose layout manager
      Container p = getContentPane();
      setSize(500,300); p.setLayout(new FlowLayout());
      // add the buttons
      p.add(ouch); p.add(start);
   }

   // the app’s main
   public static void main(String args[]) {
      // create the application frame
      MyApp app = new MyApp();
      // make it appear
      app.setVisible(true);
   }
```

- ✔ When this application runs, it creates the window shown on the next slide.
  ... and because we have event listeners working, the application actually does something

An application window

- ... what happens when the Ouch! or Start buttons are clicked? Or the X close-window icon on the title bar?



- A neater way to create handlers uses inner classes ...

Next time

- Inner classes
- Anonymous inner classes as event handlers
- MouseEvent, MouseListener, MouseAdapter

(Reading: Savitch, Ch. 14)