Listeners

Anonymous inner class

Before getting to listeners, we will need anonymous inner classes...

An "anonymous" inner class is a type of inner class created on the fly in the code with a quick-and-dirty syntax.

Convenient for creating small inner classes -- essentially these will play the role of call-back function pointers as we'll see below.

As a matter of style, the anonymous inner class is appropriate for small sections of code. If the class requires non-trivial ivars or methods, then a true inner class is a better choice.

When compiled, the inner classes are given names like Outer$1, Outer$2 by the compiler.

An anonymous inner class may not have a ctor. It must rely on the default constructor of its superclass.

An anonymous inner class does not have a name, but it may be stored in a Superclass type pointer. The inner class has access to the outer class ivars, as usual for an inner class.

The anonymous inner class does not have access to local stack vars from where it is declared, unless they are declared final.

Suppose we have a class "Outer". Here we create an anonymous inner class on the fly in a method. The inner class is subclassed off of Superclass...

```
public class Outer {
   int ivar;

   public Superclass method() {
      int sum;       // ordinary stack var
      sum = ivar + 1;
      final int temp = ivar + 1; // stack var, but declared final (constant)

      // Create new anonymous inner class, subclassed off Superclass
      Superclass s = new Superclass() {
         private int x = 0;

         public void foo() {
            x++;     // x of inner class
            ivar++;  // ivar of outer class
            bar();   // inherited from Superclass

            // x = sum; // no, cannot see sum
            x = temp;      // this works, since temp is final
         }
      };
      return(s);     // later on, someone can send s.foo()
   }
...
```

final var trick

Inner classes can see ivars of outer object
Inner classes **cannot** see stack vars from where they are created.
However, inner classes **can see** "**final**" **stack vars** from where they are created -- so declare stack vars as final to communicate their value to an anonymous inner class.
Use "Outer.this" to refer to the this pointer of the outer object. Necessary in some cases if the compiler cannot distinguish that a ivar should be available from the outer object.

Controls and Listeners

Control-Listener Theory

Source
    Buttons, controls, etc.
Listener
    An object that wants to know when the control is operated
Notification message
    A message sent from the source to the listener as a notification that the event has occurred

1. Listener Interface

ActionListener interface
Objects that would like to listen to a JButton must implement ActionListener

```
public interface ActionListener extends EventListener {

    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);

}
```

2. Notification Prototype

The message prototype defined in the ActionListener interface -- the message the button sends.
The ActionEvent parameter includes extra information about the event in case the listener cares -- a pointer to the source object (e.getSource()), when the event happened, modifier keys held down, etc,

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

3. source.addXXX(listener)

To set up the listener relationship, the listener must register with the source
e.g. button.addActionListener(listener)
The listener must implement the ActionListener interface
    i.e. it must respond to the message that the button will send

4. Event -> Notification

When the action happens (button is clicked, etc.) ...
The source iterates through its listeners
Sends each the notification
e.g. JButton sends the actionPerformed() message to each listener

Using a Button and Listener

There are 3 ways, but technique (3) below is the most common...

1. Component implements ActionListener

The component could implement the interface (ActionListener) directly, and register "this" as the listener object. This works, but is rarely done.

```
class MyComponent extends JComponent implements ActionListener {
   ...
   ...
   // in the JComponent ctor
   button.addActionListener(this);
```

2. Create an inner class to be the dest

Like the ChunkIterator strategy.
Create a MyListener inner class that implements ActionListener
Create a new MyListener object and add it via button addXXX(listener)
This works fine, but is rarely done.

```
// in the JComponent ctor
ActionListener listener = new MyActionListener();
button.addActionListener(listener);
```

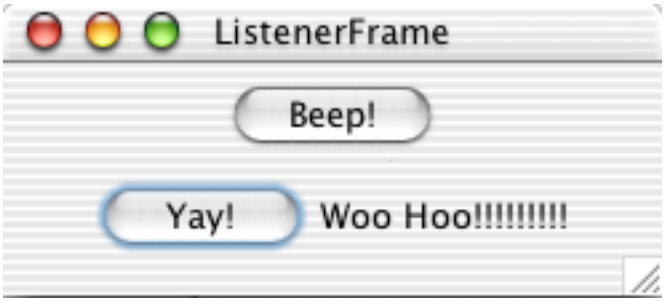
3. Anonymous inner class

Create an "anonymous inner class" that implements the listener interface
Like an inner class (option 2), but does not have a name

Can be created on the fly inside a method

```
button = new JButton("Beep");
panel.add(button);
button.addActionListener(
   new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         Toolkit.getDefaultToolkit().beep();
      }
   }
);
```

Button Listener Example




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

/*
 Demonstrates bringing up a frame with a couple of buttons in it.
 Demonstrates using anonymous inner class listener.
*/
public class ListenerFrame extends JFrame {
   private JLabel label;

   public ListenerFrame() {
      super("ListenerFrame");

      JComponent content = (JComponent) getContentPane();
      content.setLayout(new FlowLayout());

      JButton button = new JButton("Beep!");
      content.add(button);
```

```
   // ----
   // Creating an action listener in 2 steps...

   // 1. Create an inner class subclass of ActionListener
   ActionListener listener =
      new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            Toolkit.getDefaultToolkit().beep();
         }
      };

   // 2. Add the listener to the button
   button.addActionListener(listener);


   // ----
   // Creating a listener in 1 step...

   // Create a little panel to hold a button
   // and a label
   JPanel panel = new JPanel();
   content.add(panel);

   JButton button2 = new JButton("Yay!");
   label = new JLabel("Woo Hoo");
   panel.add(button2);
   panel.add(label);

   // This listener adds a "!" to the label.
   button2.addActionListener(
      new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            String text = label.getText();
            label.setText(text + "!");
            // note: we have access to "label" of outer class
            // we do not have access to local vars like 'panel',
            // unless they are declared final.
         }
      }
   );

   pack();
   setVisible(true);
}
```

Misc Listeners

JCheckBox

Uses ActionListener, like JButton
Responds to boolean isSelected() to see if it's currently checked

JSlider

JSlider -- component with min / max / current int values
JSlider uses the StateChangeListener interface -- the notification is called stateChanged(ChangeEvent e)
Use e.getSource() to get a pointer to the source object
JSlider responds to int getValue() to get its current value

Listener Strategy

The way we've done things so far.
Get notifications from the button, slider, etc. at the time of the change

Poll Strategy

Another technique -- do not listen to the control. Instead, check the control's value at the time of your choosing
e.g. checkbox.isSelected()
Avoid having two copies of the control's state -- just use the one copy in the control itself.
Polling does not work if you need to do something immediately on control change, since you want to hear of the change right when it happens.
Polling is simpler if you can get a way with it.