Anonymous Classes

• An anonymous class is a local class that does not have a name.
• An anonymous class allows an object to be created using an expression that combines object creation with the declaration of the class.
• This avoids naming a class, at the cost of only ever being able to create one instance of that anonymous class.
• This is handy in the AWT.

Anonymous Class Syntax

• An anonymous class is defined as part of a `new` expression and `must` be a subclass or implement an interface

  ```java
  new className( argumentList ) { classBody }
  new interfaceName() { classBody }
  ```

• The class body can define methods but cannot define any constructors.
• The restrictions imposed on local classes also apply

Using Anonymous Classes

```java
public class Dog { 
  private String breed;
  private String name;

  public Dog( String theBreed, String theName ) { 
    breed = theBreed;
    name = theName;
  }

  public String getBreed() { 
    return breed;
  }

  public String getName() { 
    return name;
  }
} // Dog
```
Using Anonymous Classes

```java
public void PrintDogsByName( List<Dog> dogs ) {
    // Don't change the input list - so make a copy
    List<Dog> sorted = new ArrayList<Dog>( dogs );
    // Sort the dogs using an anonymous class
    Collections.sort( sorted,
        new Comparator<Dog>() {
            public int compare( Dog d1, Dog d2 ) {
                return d1.getName().compareTo( d2.getName() );
            }
        });
    // Print out the results
    Iterator<Dog> i = sorted.iterator();
    while ( i.hasNext() ) {
        System.out.println( i.next() );
    }
}
```

The Job of a Window Manager

Event Driven Programming

- Programs respond to events that are generated outside the control of the program
  - User types a key
  - The left mouse button is pressed
  - A CD is removed from the CD drive
- When an event occurs, it is handled by an event handler
- Event driven programming involves writing the handlers and arranging for the handler to be notified when certain events occur
Event Handling

• Events are represented by objects that give information about the event and identifies the event source
  – Event sources are typically components, but other kinds of objects can also be event sources
• A listener is an object that wants to be notified when a particular event occurs
  – An event source can have multiple listeners registered on it
  – A single listener can register with multiple event sources

Listeners

• In order for an object to be notified when a particular event occurs, the object
  – Must implement the appropriate Listener interface
  – Be registered as an event listener on the appropriate event source

Swing Listeners

<table>
<thead>
<tr>
<th>Action</th>
<th>Listener Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>User clicks a button, presses return while typing in a text file, or chooses a menu item</td>
<td>ActionListener</td>
</tr>
<tr>
<td>Users closes a frame (main window)</td>
<td>WindowListener</td>
</tr>
<tr>
<td>User presses a mouse button while the cursor is over a component</td>
<td>MouseListener</td>
</tr>
<tr>
<td>User moves the mouse over a component</td>
<td>MouseMotionListener</td>
</tr>
<tr>
<td>A component becomes visible</td>
<td>ComponentListener</td>
</tr>
<tr>
<td>A component gets the keyboard focus</td>
<td>FocusListener</td>
</tr>
<tr>
<td>A table of list selection changes</td>
<td>ListSelectionListener</td>
</tr>
</tbody>
</table>
Window Closing

- A very common event directed towards a window is a close event
  - The default behavior is to simply hide the JFrame when the user closes the window
- Normally we would want the program to terminate when the user closes the main window
- Two steps required to accomplish this
  - Write an event handler for the close event that will terminate the program
  - Register the handler with the appropriate event source

WindowListener

- The WindowListener interface
  - void windowActivated(WindowEvent e);
  - void windowClosed(WindowEvent e);
  - void windowClosing(WindowEvent e);
  - void windowDeactivated(WindowEvent e);
  - void windowDeiconified(WindowEvent e);
  - void windowIconified(WindowEvent e);
  - void windowOpened(WindowEvent e);
- A class that implements WindowListener must implement all of these methods!

WindowAdapter

- A class that implements WindowListener
  - The methods in this class are empty. The class exists as convenience for creating listener objects.
- To use the WindowAdapter class:
  - Extend this class to create a WindowEvent listener
  - Override the methods for the events of interest
  - Create a listener object using the extended class and then register it with the Window.
- When an event occurs the appropriate method in the listener is invoked.
The Result

import javax.swing.*;
import java.awt.event.*;

public class SwingFrame {
    public static void main(String[] args) {
        JFrame win = new JFrame("My First GUI Program");
        win.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        win.setSize(250, 150);
        win.setVisible(true);
    }
} // SwingFrame

Closing a Window

- Java 1.4 added the setDefaultCloseOperation() method to the JFrame class
  - EXIT_ON_CLOSE
  - DO_NOTHING_ON_CLOSE (default)
  - HIDE_ON_CLOSE
  - DISPOSE_ON_CLOSE

The Result

import javax.swing.*;
import java.awt.event.*;

public class SwingFrame {
    public static void main(String[] args) {
        JFrame win = new JFrame("My First GUI Program");
        win.setDefaultCloseOperation(EXIT_ON_CLOSE);
        win.setSize(250, 150);
        win.setVisible(true);
    }
} // SwingFrame
Buttons

- Buttons generate action events
- The ActionListener interface
  - void actionPerformed(ActionEvent e);
  - Note that there is no need for an ActionAdapter class
- Generally one ActionListener will be responsible for handling the events generated by a group of buttons
  - You can tell which button got pressed using the event’s getActionCommand() method

Example

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

public class SwingFrame implements ActionListener {
    private JFrame win;
    public SwingFrame( String title ) {
        win = new JFrame( title );
        win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        win.getContentPane().setLayout( new FlowLayout() );
        for ( int i = 0; i < 10; i++ ) {
            JButton b = new JButton( String.valueOf( i ) );
            b.addActionListener( this );
            win.getContentPane().add( b );
        }
        win.pack();
        win.setVisible( true );
    }

    public void actionPerformed( ActionEvent e ) {
        System.out.println( "Button " +
            e.getActionCommand() +
            " was pressed" );
    }

    public static void main( String args[] ) {
        SwingFrame f = new SwingFrame( "My First GUI" );
    }
}
```

Example

```java
public void actionPerformed( ActionEvent e ) {
    System.out.println( "Button " +
        e.getActionCommand() +
        " was pressed" );
}
```

```java
public static void main( String args[] ) {
    SwingFrame f = new SwingFrame( "My First GUI" );
}
```
GUI Program Design

• The GUI provides a view of the program, it is clearly not the program
• Making the GUI code as independent of the program code is a good strategy
  – Changes in the program do not necessarily change the GUI
  – Different GUIs can be developed for the same program
  – Debugging and maintaining both the GUI and the program code is easier
Model-View-Controller

- The MVC pattern is commonly used to develop applications that have a GUI component
- Consists of three parts
  - Model
    - The program
  - View
    - The GUI
  - Controller
    - The event handling mechanism

MVC

- The model passes its data to the view for rendering
- The view determines which events are passed to the controller
- The controller updates the model Based on the events received

MVC in Swing

- The model passes its data to the view for rendering
- The view determines which events are passed to the controller
- The controller updates the model Based on the events received
A Simple 4 Function Calculator

```
Calculator
- Label: String
- Number: int
- Display: Label
+ CalcAdd
+ getDisplay String
++ setDisplay void
+ setPerformed void

Calculator
- Result: String
- firstDigit: boolean
- operand: int
- operator: String
+ calcResult:
+ handleButton void
++ expect void
++ main void
```