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 (for actionListeners).
An anonymous class is defined as part of a `new` expression and must be a subclass or implement an interface (without stating “extends” or “implements”).

The class body can define methods but cannot define any constructors.

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

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; } 
    public int compareTo(Object o) throws ClassCastException {
        Dog other = (Dog)o; 
        int retVal = breed.compareTo( other.getBreed() );
        if (retVal == 0 ) 
        {retVal = name.compareTo( other.getName() );
        return retVal;
        }   
    }  // Dog
Using Anonymous Classes

```java
public void PrintDogsByName( List dogs ) {
    List sorted = dogs;
    Collections.sort( sorted,
        new Comparator () {
            public int compare( Object o1, Object o2) {
                Dog d1 = (Dog)o1;
                Dog d2 = (Dog)o2;
                return d1.getName().compareTo( d2.getName() );
            }
        });
    Iterator i = sorted.iterator();
    while ( i.hasNext() )
        System.out.println( i.next() );
}
```

The Job of a Window Manager

![Diagram of the job of a window manager](image)
What is Event Driven Programming?

- Java's GUI design is based on **event driven programming**

- An **event** is a signal to the program that some external action has occurred (outside the control of the program)
  - A button was clicked
  - The mouse was moved
  - A key was pressed
  - A CD is removed from the CD drive
  - A timer in the operating system expired

- When an event is triggered, a special piece of code can run to respond to the event
  - The left mouse button was pressed so fire the current weapon
  - The mouse was moved so update the players look direction
  - The forward key was pressed so update the players position

- Event driven programming involves writing the handlers and arranging for the handler to be notified when certain events occur

Events and Event Source

- The component which generated the event is the **source object**
  - A button is the source of a button clicking action

- The event generated is an object of **EventObject**
A *listener* is an object who is interested in receiving events.

When this button is pressed, I want the button (the source object) to call me (the listener object) with the event and any pertinent information.

For an object to be a listener it must do two things:
- Implement the corresponding *event-listener* interface
- Register with the source object who generates the event
• For example, the corresponding listener interface for an ActionEvent is ActionListener

• The ActionListener interface requires the listener to implement the following **handler**:

```
void actionPerformed(ActionEvent event);
```

When this button is pressed, the button will call actionPerformed() with the ActionEvent to all registered listeners.

---

• A listener **registers** with the source object by invoking a registration method in the source object

• The JButton object has an addActionListener method which takes the listener object

```
ListenerClass listener = new ListenerClass();
JButton button = new JButton("OK");
button.addActionListener(listener);
```

Every button push will cause button to call the actionPerformed method in listener.
The **event object** contains information pertinent to the event type:

- `java.util.EventObject` returns the object on which the event initially occurred.
- `java.awt.event.ActionEvent` returns the command string associated with the action (i.e., the button text).
- `java.awt.event.AWTEvent` returns the timestamp when the event occurred.

### Action Events

- Write a program that displays two buttons, **OK** and **Cancel**, in the window. A message is displayed on the console to indicate which button was pressed and when.

```bash
% java TestActionEvent
The OK button was clicked at Mon Jan 24 20:13:58 EST 2005
The Cancel button was clicked at Mon Jan 24 20:14:03 EST 2005
```

- `/usr/local/pub/sps/courses/cs2/events/ActionEvent`
• Write a program that demonstrates handling of window events:
  - Window opened
  - Window closing/closed
  - Window activated
  - Window deactivated
  - Window iconified
  - Window deiconified

• /usr/local/pub/sps/courses/cs2/events/WindowEvent

• The WindowAdapter class is a class that implements the WindowListener interface
  - The methods in this class are empty.

• 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 a Window using the window's addWindowListener() method.
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

- Write a program which changes values by 1's and 2's using multiple listeners

Listener 1 created: 0
Listener 2 created: 0
Listener 2 inc: 2
Listener 1 inc: 1
Listener 2 inc: 4
Listener 1 inc: 2
Listener 2 dec: 2
Listener 1 dec: 1
Listener 2 inc: 4
Listener 1 inc: 2

- /usr/local/pub/sps/courses/cs2/events/MultipleListeners
Mouse Events

- A **mouse event** is generated whenever a mouse is pressed, released, clicked, moved or dragged on a component.

<table>
<thead>
<tr>
<th>java.awt.event.InputEvent</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ getWhen() : long</td>
</tr>
<tr>
<td>+ isAltDown() : boolean</td>
</tr>
<tr>
<td>+ isControlDown() : boolean</td>
</tr>
<tr>
<td>+ isMetaDown() : boolean</td>
</tr>
<tr>
<td>+ isShiftDown() : boolean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.awt.event.MouseEvent</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ getButton() : int</td>
</tr>
<tr>
<td>+ getClickCount() : int</td>
</tr>
<tr>
<td>+ getPoint() : java.awt.Point</td>
</tr>
<tr>
<td>+ getX() : int</td>
</tr>
<tr>
<td>+ getY() : int</td>
</tr>
</tbody>
</table>

- Which mouse button was clicked?
- How many times was it clicked?
- Get the coordinates for the mouse point

Mouse Listeners

- There are two listener interfaces to handle mouse events.

<table>
<thead>
<tr>
<th>java.awt.event.MouseListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ mousePressed(e : MouseEvent) : void</td>
</tr>
<tr>
<td>+ mouseReleased(e : MouseEvent) : void</td>
</tr>
<tr>
<td>+ mouseClicked(e : MouseEvent) : void</td>
</tr>
<tr>
<td>+ mouseEntered(e : MouseEvent) : void</td>
</tr>
<tr>
<td>+ mouseExited(e : MouseEvent) : void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>java.awt.event.MouseMotionListener</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ mouseDragged(e : MouseEvent) : void</td>
</tr>
<tr>
<td>+ mouseMoved(e : MouseEvent) : void</td>
</tr>
</tbody>
</table>

- Pressed and released
- Moved while pressed
• Write a program which uses the mouse for scribbling. You can draw with the left mouse button and erase with the right mouse button

• /usr/local/pub/sps/courses/cs2/events/ScribbleDemo

• A **key event** is generated whenever a key is pressed, released, or typed on a component

```java
java.awt.event.InputEvent

java.awt.event.KeyEvent
+ getKeyChar() : char
+ getKeyCode() : int  
character associated with the key
integer associated with the key

java.awt.event.KeyListener
+ keyPressed(e : KeyEvent) : void
+ keyReleased(e : KeyEvent) : void
+ keyTyped(e : KeyEvent) : void  
Key pressed + released
```
• Write a program that displays a user input character which can be moved around with the arrow keys and changed

```
KeyEvent.VK_RIGHT
```

```
KeyEvent.VK_RIGHT
```

```
KeyEvent.VK_RIGHT
```

```
KeyEvent.VK_DOWN
```

• /usr/local/pub/sps/courses/cs2/events/KeyboardDemo

---

• A **timer** is a source object which can trigger an **ActionEvent** at a predefined rate
  - It's not a visible GUI component

```
javax.swing.Timer
```

```
+ Timer(delay : int, listener : ActionListener)
+ addActionListener(listener : ActionListener) : void
+ start() : void
+ stop() : void
+ setDelay(delay : int) : void
```

Create a timer with a specified delay and a listener
Add a listener to the timer
Start, stop, or set a new delay on the timer
• Write a program which modifies the keyboard demo to automatically move the character around the screen using a timer

• The character should move in the direction of the last arrow press (initial = right), every $1/10^{\text{th}}$ of a second

• The character should wrap around the screen edges

• The animation should pause if the space key is pressed

• /usr/local/pub/sps/courses/cs2/events/TimerDemo

• Write a program which draws line segments using the arrow keys. The line starts from the center of the frame and draws towards east, north, west, or south when the arrow keys are pressed
GUI Program Design

- The GUI provides a view of the program, it is clearly not the program.
- Making the GUI code 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 can be done separately and 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
The model passes its data to the view for rendering.

The controller updates the model based on the events received.

The view determines which events are passed to the controller.

The model passes its data to the view for rendering.

The controller updates the model based on the events received.

The view determines which events are passed to the controller.

The Application

Program Logic

The GUI
### A Simple 4 Function Calculator

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
</tr>
</thead>
</table>
| CalcGui | - `labels`: String  
|        | - `NUMROWS`: int  
|        | - `NUMCOLS`: int  
|        | - `display`: JLabel  
|        | + `CalcGui`  
|        | + `setDisplay`: String  
|        | + `setDisplay`: void  
|        | + `actionPerformed`: void |

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
</table>
|        | - `nyCalc`  
|        | - `gui`  
|        | + `Calculator`  
|        | - `DIGITS`: String  
|        | - `firstDigit`: boolean  
|        | - `lOperand`: int  
|        | - `operator`: String  
|        | + `Calculator`  
|        | + `handleButton`: void  
|        | - `compute`: void  
|        | + `main`: void |