ICS 111
Event-Oriented Programming

- GUI interaction model
- Events
- Embedded Systems
- The package java.awt.event
- Inner Classes
- Anonymous Inner Classes

GUI Interaction Model

- One of the advantages of a GUI over text I/O is that the user can do things in different orders
  - for example, if multiple text fields and selections are present, the user can fill them out and select in any sequence
  - most of the time, the user may select any menu item
- each such interaction may require interaction with the program, which means that the GUI (the Java runtime system) must be able to call a method in your program
- such a call reflects a user interaction **event**
- a program must be able to handle such events in any order

Event-Oriented Programming

- In most programs so far, the programmer decides the order in which operations are done
- but when a program interacts with the user, the user determines the order of events
  - when a program in a device interacts with the real world, the real world determines the order of external events
- an event-oriented program program must:
  - listen for incoming events
  - respond appropriately to each event
    - this is the part where the programmer can decide which operations should be part of the response
  - keep track of any information needed to respond

Event-Oriented Programming for Embedded Systems

- The Internet of Things (IoT) is the collection of connected devices that operate more or less autonomously
  - this includes smart speakers, vehicle control systems, remote sensors, and everything other “smart” device that operates mostly autonomously
  - another term for such devices is **embedded systems**
- embedded systems must respond to events in their environment
- when idle (not responding) they should save energy by not doing any processing
- events may include changes in the physical world, vocal calls to wake up, and incoming network traffic
- some embedded systems, including vehicle control systems have to respond within a specific time, or risk failure: these are **real-time systems**

The package java.awt.event

- This package provides almost 20 interfaces and classes, each with a name ending in ...Listener
- to handle events, we must first register a Listener, a method that will be called when the event is detected
- ActionListener is an interface that requires the method

  `void actionPerformed(ActionEvent e)`

  - a class that implements ActionListener, provides an actionPerformed method
  - actionPerformed will be called once for every matching event
- to listen for a button event, call the button's addActionListener method, giving as parameter any object that implements the ActionListener interface
- the GUI calls the actionPerformed method when the button is clicked
- an example is on the next slide

ActionListener example

```
public class PrintButtonClicks
  implements java.awt.event.ActionListener {
  public void
    actionPerformed (java.awt.event.ActionEvent e)
  {
    System.out.println("button clicked!");
  }
}

javax.swing.JButton b =
  new javax.swing.JButton("click here");
...
b.addActionListener(new PrintButtonClicks());
```

Unsatisfactory Ways of implementing action listeners

- creating a new class for every button that you listen to can lead to having lots of classes
- also, a button listener should have access to the instance variables of the class (call it **C**) that has the graphics code
- one answer is to have the class C implement ActionListener and provide actionPerformed
- problem: this does not work well when you have multiple buttons
  - each class (such as C) can only provide a single actionPerformed method

Inner Classes

- instead of having one-class-per-button or trying to squeeze all the functionality of all the action listeners into a single method in the main class, it is better to create an inner class for each action listener
- an inner class is a class definition that is inside another class
- we can have as many of these inner classes as we need
- the methods of the inner class have access to the instance variables of the enclosing class

Inner Class Syntax Example

```
public class Outer {
  int v1 = 99;
  String v2 = "hello world";
  Inner1 i1;
  Inner2 i2;
  class Inner1 {
    public void m1() {
      System.out.println("v2 is " + v2);
    }
  }
  class Inner2 {
    ...
  }
  public Outer() {
    i1 = new Inner1();
    i1.m1();
  }
```

Where to define Inner Classes

- inner classes are commonly defined inside top-level classes
- this is not a requirement – inner classes could be defined anywhere, even inside a method
- if defined inside a method, only that method has access to the inner class
- and the inner class only has access to any `final` instance variables
- final instance variables cannot be assigned to, but the object they refer to can change
  - e.g. if the final instance variable refers to an ArrayList, when can add values to the array list

ActionListener example with an Inner Class

```
public class AllMyGraphics {
  javax.swing.JButton b =
    new javax.swing.JButton("click here");
  class PrintButtonClicks
    implements java.awt.event.ActionListener {
    public void
      actionPerformed (java.awt.event.ActionEvent e) {
      System.out.println("button clicked! button " + b +
                         ", event " + e);
    }
  }
  public AllMyGraphics() {
    ...
    b.addActionListener(new PrintButtonClicks());
  }
```

Anonymous Inner Classes

- sometimes an inner class is only used once, such as for an action listener
- then, it is not necessary to give the class a name
- in this case, we can use as a constructor the name of an interface that this anonymous class implements

ActionListener example: Anonymous Inner Class inside a method

```
public class AllMyGraphics {
  final javax.swing.JButton b =
    new javax.swing.JButton("click here");
  public AllMyGraphics() {
    ...
    b.addActionListener(new java.awt.event.ActionListener() {
      public void
        actionPerformed (java.awt.event.ActionEvent e)
        {
          System.out.println("button clicked! button " + b +
                             ", event " + e);
        }  // ends actionPerformed
      });    // ends the anonymous inner class
    }      // ends the no-arguments constructor
  }
```

Summary

- events happen!
- event-handling code is called by the system (the system that implements the GUI) when a user takes a specific action
- different events may call different methods
  - or may all call the same method, and the event would be used to figure out which event happened
- inner classes are convenient when the alternative would be many small classes
- inner classes may be defined inside methods, and then can only access `final` instance variables
- inner classes may be anonymous
- see the code examples on the course web page