CSE 11: Lecture 13

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

(Reading: Savitch, Ch. 14)

Implementing event listeners

- 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

- Using event-driven programming, components contained in the JFrame, and the JFrame itself, will have event listeners registered with them to handle events
  - ... creating and adding the components and registering the listeners is typically done in MyApp's constructor

- We have seen two approaches to creating listener objects:
  - define separate named classes that implement the appropriate listener interfaces, or that extend the appropriate adapter class
  - declare the top-level application class both to extend JFrame and to implement the appropriate listener interfaces

- Both of these approaches have disadvantages

Toward anonymous inner classes

- ✔ Defining separate event-handler classes...
  - ✗ requires writing separate source code files
  - ✗ requires coming up with names for the separate classes
  - ✗ separate class cannot access private variables or methods of your application class
- ✔ Declaring your application class to implement listener interfaces...
  - ✗ requires implementing all the interface methods, even if you don't need them all
  - ✗ there can be only one event-handler method for each event type (for example, one actionPerformed() method), even though many components may fire the event and need to be handled differently: leads to calling getSource() and doing multiway branching on the result
- ✔ In registering a listener with an event-generating component, all you really want to do is give the component a method to call to handle the event when it occurs
- ✔ ... this job is made easy by the use of anonymous inner classes

Inner classes

- Inner, or nested, classes are an advanced object-oriented feature introduced in JDK 1.1
- An inner class is: a class defined within the body of the definition of another class
- An inner class can have **`public`**, "package", **`protected`**, or **`private`** visibility; and it can be **`static`** or "instance" inner class
- Most interesting for what we want to do here, an inner class can be *anonymous*
  - an anonymous inner class is defined without giving it a name!
  - because it doesn't have a name, you can't refer to it except once as an argument to **`new`**
  - the anonymous inner class will have method definition(s) inside it... which in the case of event handlers is all you want to do
  - an anonymous inner class is an "instance" inner class, and so methods in the inner class can access instance variables and instance methods in its outer class!
- This is very convenient for associating a different ActionListener with each component, associating a WindowAdapter with a Frame, etc.

Defining an anonymous inner class

- Basic syntax for defining and creating an instance of an anonymous class:

  `new <typename> () { <class-body> }`

  where **`<typename>`** is a class or interface name

- This does two things:
  - defines a new anonymous inner class, which either extends **`<typename>`**(if **`<typename>`** is a class name) or implements **`<typename>`** (if **`<typename>`** is an interface name)
  - creates an instance of that class, using the default constructor

- The <class-body> should include definition(s) of method(s) as needed
  - these override definitions in **`<typename>`** (if **`<typename>`** is a class name) or implement methods in **`<typename>`** (if **`<typename>`** is an interface name)

Anonymous inner classes: an example

- AWT/JFC applications often need an event handler to listen for window-closing events
- Savitch shows the example of defining a class, WindowDestroyer, which extends WindowAdapter and overrides the windowClosing() method. An instance of this class is created and registered as a window listener with the application JFrame
- In the last lecture, we showed the example of declaring the application class MyApp to implement WindowListener, which required implementing 7 methods in MyApp; an instance of the application class itself was registered as a window listener with the application
- Instead, we can define an anonymous inner class that extends WindowAdapter, override the windowClosing method (which is the only one we cared about), create an instance of it, and register it as a listener with the JFrame very easily:

```
public class MyApp extends JFrame {
   MyApp() {

      addWindowListener( new WindowAdapter () {
            public void windowClosing(WindowEvent e) {
               System.exit(0);
            }});
      ...
```

An improvement in JDK 1.3

- As we have seen, there are 7 kinds of window events, corresponding to 7 different methods required by the WindowListener interface
- But it is common only to care about one of these, the one handled by the windowClosing() method; and it is common for that handler just to exit the application in response to the event
- We have looked at 3 ways to implement this functionality, but each of them seems overly complicated
- In JDK 1.3, in response to this poor design, an easier approach to this problem was added. There is now an instance method of the JFrame class

```
void setDefaultCloseOperation(int op)
```

- ... which lets you specify what happens when the user initiates a "close" on the JFrame. Useful named constants to pass as argument are:

```
WindowConstants.DO_NOTHING_ON_CLOSE // use the registered
                                    // listener, if any
WindowConstants.HIDE_ON_CLOSE       // hide the window
                                    // (this is the default)
JFrame.EXIT_ON_CLOSE                // exit the application
```

- So `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` will have the effect of what we have been doing in our WindowListener examples...

Anonymous inner classes: another example

- In the last lecture, an example declared the MyApp class to implement ActionListener; this led to having only one actionPerformed method which had to handle events from multiple components
- Instead, we could define anonymous classes that implement ActionListener, each with an actionPerformed method that is suited exactly to the component it is registered with:

```
JButton start = new JButton("Start");
start.addActionListener(new ActionListener () {
      public void actionPerformed(ActionEvent e) {
         // code to handle the start button being pushed

      }});

JButton ouch = new JButton("Ouch!");
ouch.addActionListener(new ActionListener () {
      public void actionPerformed(ActionEvent e) {
         // code to handle the ouch button being pushed

      }});
```

MouseEvents

- A MouseEvent represents event that indicates that a mouse action happened in a component

- These things will fire a MouseEvent:
  - a mouse button is pressed
  - a mouse button is released
  - a mouse button is clicked (pressed and released)
  - the mouse cursor enters a component
  - the mouse cursor leaves a component
  - the mouse is moved (this fires a MouseMotionEvent, a subclass of MouseEvent)
  - the mouse is dragged (this fires a MouseMotionEvent, a subclass of MouseEvent)

- MouseEvent objects contain information accessible with methods shown next...

MouseEvent methods

- Every MouseEvent object has several public instance accessor methods which are useful for finding out information about it. The most generally useful ones are:

```
// Returns the x position of the event relative to the
// source component.
public int getX()

// Returns the y position of the event relative to the
// source component.
public int getY()

// Returns as a java.awt.Point the x,y position of
// the event relative to the source component.
public Point getPoint()

// Returns the number of clicks associated with this event
// (1 for single click, 2 for double click, etc.)
public int getClickCount()

// Returns an int encoding state of mouse and keyboard buttons
public int getModifiers()
```

MouseListener

- An object that is to be registered with a component to listen for MouseEvents must be an instance of a class that implements the **`MouseListener`** interface
- The MouseListener interface has five methods, one for each sort of thing that can fire a nonmotion MouseEvent:

```
public interface MouseListener {
   public  void mouseClicked(MouseEvent e);
   public  void mouseEntered(MouseEvent e);
   public  void mouseExited(MouseEvent e);
   public  void mousePressed(MouseEvent e);
   public  void mouseReleased(MouseEvent e);
}
```

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

- When a component fires a MouseEvent..
  - each MouseListener object that has been registered with the component will have one of these methods called, depending on what happened
  - this method will be passed the MouseEvent object correpsonding to the event

The MouseAdapter class

- You can define a class that implements MouseListener, but this requires implementing 5 methods!
- Often, you only really want to deal with one of them, e.g. **`mouseClicked`**
- To save some typing, there is an AWT class called MouseAdapter which is defined this way:

```
public abstract class MouseAdapter implements MouseListener {
   public  void mouseClicked(MouseEvent e) {}
   public  void mouseEntered(MouseEvent e) {}
   public  void mouseExited(MouseEvent e) {}
   public  void mousePressed(MouseEvent e) {}
   public  void mouseReleased(MouseEvent 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 extend MouseAdapter and override those one or two methods, than it is to implement MouseListener

Registering a MouseListener with a component

- You can use any of the techniques we have talked about (define a separate class, use the application class itself, or use an anonymous inner class) to get an object that implements MouseListener

- Once you have such an object, you can register it as a listener with any component that can fire MouseEvents.
- To do that, pass the listener as argument to the component's **`addMouseListener`** method...

```
public class MyApp extends JFrame {
   public MyApp()  {  // constructor
      ...
      addMouseListener(new MouseAdapter() {
         public void mouseClicked(MouseEvent e) {
               ...
      }});
      ...
   }
```

An example

- Using anonymous inner classes to create listeners... and components!

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

public class MouseApp extends JFrame {

  private String s = "";  // a String to display
  private int x, y;  // coordinates to display String

  public MouseApp()  {  // constructor, does all initialization
    setBounds(50,50,300,200); // place and size the JFrame

    // create a JPanel to do graphics on; override paintComponent()
    JPanel p = new JPanel() {  // anon inner class!
      public void paintComponent(Graphics g) {
        super.paintComponent(g); // this clears the JPanel
        g.drawString(s,x,y);
      }};
```

An example, cont'd

```
// give the JPanel a MouseListener
p.addMouseListener(new MouseAdapter() {  // anon inner class!
 public void mouseClicked (MouseEvent e) {
   x = e.getX();  y = e.getY();  // new coordinates for String
   int n = e.getClickCount();  // set String, depending oncount
   if(n==1) s = "one";
   if(n==2) s = "two";
   if(n>2)  s = "many";
   repaint();  // repaint the JFrame, to reflect the new info
}});

 // add the JPanel to the JFrame's content pane
 getContentPane().add(p);
 }

 // the app's main
 public static void main(String args[]) {
   // create the application frame & make it visible
   (new MouseApp()).setVisible(true);
 }
```

More Java GUI programming

- We have only scratched the surface of JFC/Swing programming
- For other, more sophisticated 2D graphics, there is also the java.awt.geom package which was released in JDK 1.2
- And the Java3D graphics and sound library including classes for creating portable, networkable 3D media has been released in the javax.media.j3d and javax.vecmath packages
- What you learned here about containers, components, and event-driven programming will apply to them as well
- For more information, one good source is the online tutorials on Java GUI programming

Next time

- Midterm exam! But after that:

- Declaring and creating arrays
- Indexing
- Array initialization
- Arrays as arguments and as returned values
- Multidimensional arrays

(Reading: Savitch, Ch. 6)