Drawing 2

Inner/Nested Classes

private class ("inner class")

A class definition inside an "outer" class
Use as a private utility class -- declare private and clients can't see it
Access style
    The outer and inner classes can access each other's state, even if it is private. Stylistically, they are essentially one class/implementation, that is superficially divided into two parts. Access back and forth is not terrible.
Inner class is always created in the context of an "owning" outer object
Inner class has a pointer to its owner -- can access ivars of owner automatically
"Outer.this" -- syntax to refer to the this pointer of outer object
The inner class can have any superclass -- it is a new class, however the inner class also has access to its outer object.
Why use an inner class? You have a class already built, and you need to create a new object with a new superclass that can access the state of the original object. This is not a common situation, but it happens to work well for Swing "listener" objects.

private static class ("nested class")

Like an inner class, but does not have a pointer to the outer object and so does not automatically access the ivars of the outer object. Use a nested class if the nested class does not need to access the state or send messages to the outer object.

Inner/Nested example

```
// Outer.java
/*
 Demonstrates inner/outer classes.
 Outer has an ivar 'a'.
 Inner has an ivar 'b'.

 Main points:

 -Each inner object is created in the context of a
 single, "owning", outer object. At runtime, the inner
 object has a pointer to its outer object which allows
 access to the outer object.

 -Each inner object can access the ivars/methods
 of its outer object. Can refer to the outer object
 as "Outer.this".
```

```
 -The inner/outer classes can access each other's ivars
 and methods, even if they are "private". Stylistically,
 the inner/outer classes operate as a single class
 that is superficially divided into two.

*/

public class Outer {
   private int a;

   private void increment() {
      a++;
   }

   private class Inner extends Object {
      private int b;
      private Inner(int initB) {
         b = initB;
      }

      private void demo() {
         // access our own ivar
         System.out.println("b: " + b);

         // access the ivar of our outer object
         System.out.println("a: " + a);

         // message send can also go to the outer object
         increment();

         /*
          Outer.this refers to the outer object, so could say
          Outer.this.a or Outer.this.increment()
         */
      }
   }

   public void test() {
      a = 10;
      Inner i1 = new Inner(1);
      Inner i2 = new Inner(2);

      i1.demo();
      i2.demo();

      /*
       Output:
       b: 1
       a: 10
       b: 2
       a: 11
      */
   }


   public static void main(String[] args) {
      Outer outer = new Outer();
```

```
      outer.test();
   }

}

public class Outer {
   private int ivar;

   private static class Nested { // a class known only to Outer
      void foo() {
         // ivar -- does not compile
         // no automatic access to outer ivars
      }
   }

   public void test() {
      Nested nested = new Nested();
      nested.foo();
      ...
   }
}
```

Anonymous inner class

An "anonymous" inner class is a type of inner class created with a quick-and-dirty syntax.

Does not have a name

Can be created on the fly inside a method

Cryptic, but convenient

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

Our 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.

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 void method() {
      int sum;       // ordinary method body statements
      sum = 1 + 1;
      // create new inner class object on the fly
      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
         }
      };
      s.foo();
   }
...
```

Interface

Java "interface"

Method Prototypes
- Defines a set of methods prototypes.
- Does not provide code for implementation -- just the prototypes.
- Can also define read-only constants.

Class implements interface
- A class that implements an interface must implement all the methods in the interface. The compiler checks this at compile time.
- A Java class can only have one superclass, but it may implement any number of interfaces.
- Use "implements" keyword for an interface, vs the "extends" keyword for a superclass.

"Responds To"
- The interface is a "responds to" claim about a set of methods.
- If a class implements the Foo interface, I know it responds to all the messages in the Foo interface.
- In this sense, an interface is very similar to a superclass.
- If an object implements the Foo interface, a pointer to the object may be stored in a Foo variable. (Just like storing a Grad pointer in a Student variable.)

vs. Multiple Inheritance
- C++ multiple inheritance is more capable -- multiple superclasses -- but it introduces a lot of compiler and language complexity, so maybe it wasn't worth it. Interfaces provide 80% of the benefit for 10% of the effort.
- Interfaces allow us to inherit the responds to claim. However, an interface does not allow us to inherit the actual code -- the subclass must still implement the method code.

Irritable Interface

```
// Irritable.java
/*
 Simple example of an interface.
 Defines the method isIrate() --
  subclasses must implement this method.
*/
public interface Irritable {
   public boolean isIrate();  // currently angry?
   public final int A_CONSTANT = 100;
}
```

Irritable Student

```
//IrateStudent.java
/*
 A subclass of Student that also implements the Irritable interface.
 Like a student, but we also respond to isIrate().
*/
public class IrateStudent extends Student implements Irritable {
   // Our ctor just chains up to the Student ctor
   public IrateStudent(int units) {
      super(units);
   }
   // Implement the method from the Irritable interface
   public boolean isIrate() {
      return (getStress() > 200);
   }

   // sample client code
   public static void main(String[] args) {
      IrateStudent s = new IrateStudent(10);

      // ok -- Irritable is a "superclass" of IrateStudent
      Irritable i = s;

      // ok -- pops down to IrateStudent.isIrate()
      boolean irate = i.isIrate();
   }
}
```

Controls and Listeners

Listener Style

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

e.g. ActionListener
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

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

3. source.addXXX(listener)

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 everyone uses technique (3) below...

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.

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 pass it button addXXX(listener)
This works fine, but is rarely done.

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 {
   JLabel label;

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

      JComponent content = (JComponent) getContentPane();
      content.setLayout(new FlowLayout());
      content.setBackground(Color.white);

      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);


      // 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);


      // More realistic example -- add the listener in 1 step.

      // This listener adds a "!" to the label.
      button2.addActionListener(
         new ActionListener() {
            public void actionPerformed(ActionEvent e) {
               // note: we have access to "label" of outer class
               String text = label.getText();
               label.setText(text + "!");
            }
         }
      );


      pack();
      setVisible(true);
   }

}
```