Advanced OOP Concepts in Java

Michael B. Spring
Department of Information Science and Telecommunications
University of Pittsburgh
spring@imap.pitt.edu
http://www.sis.pitt.edu/~spring

09/28/2001
Overview of Part 1 of the Course

- Demystifying Java: Simple Code
- Introduction to Java
- An Example of OOP in practice
- Object Oriented Programming Concepts
- OOP Concepts -- Advanced
- Hints and for Java
- I/O (Streams) in Java
- Graphical User Interface Coding in Java
- Exceptions and Exception handling
Overview of this Slide Set

- Nested Classes
- Inner Classes
- Member Classes
- Local Classes
- Anonymous Classes
Nested classes and Inner classes

- Introduction: a taxonomy
- Nested top-level classes (and interfaces)
- `.class files and JVM
- Inner classes
- Member class
- Local class
- Containment hierarchy
- Anonymous class
- Visibility / Access
Introduction: nested classes

- Java 1.0 allowed class declaration only at the package level. Class names are therefore organized into packages, each having its own name space. These are called top-level classes.
- Java 1.1 allows class (and interface) declaration within the class scope, nested inside the definition of another class. These are called nested classes, and nested interfaces.
- Nested interfaces are always static. Nested classes may or may not be static. Static nested classes and interfaces are functionally top-level classes, but non-static nested classes are inner classes.
Introduction: a taxonomy

- **top-level classes and interfaces:**
  1) **package** level class or interface
  2) **static** class or interface *nested* in class scope.
- **inner classes:**
  - **member** class – declared as a *member* of the containing class;
  - **local** class – declared as a *local variable* within a method body of the containing class;
  - **anonymous** class – *local class* with no declaration (and therefore no name), but an instance can be created.
Nested top-level class & interface

• Nested top-level class must be *declared static*.
• Nested top-level interface is *always static*.
• Being *static*, it is *functionally* same as *top-level*.
• It is *conveniently nested* so that it does not clutter the package level name space, often used in a *helper* class such as an iterator designed for some data structure.
Nested top-level class & interface

interface Link {
    public Link getNext();
    public void setNext(Link node);
}

class Node implements Link {
    int i;
    private Link next;
    public Node(int i) { this.i = i; }
    public Link getNext() { return next; }
    public void setNext(Link node) { next = node; }
}

public class LinkedList {
    private Link head;
    public void insert(Link node) { ... }
    public void remove(Link node) { ... }
}

◆ Link is an interface for nodes of a linked list, let us define it inside the class scope of LinkedList.
Nested top-level class & interface

```java
public class LinkedList {
    public interface Link {
        public Link getNext();
        public void setNext(Link node);
    }
    private Link head;
    public void insert(Link node) { ... };
    public void remove(Link node) { ... };
}

class Node implements LinkedList.Link {
    int i;
    private LinkedList.Link next;
    public Node(int i) { this.i = i; }
    public LinkedList.Link getNext()
    { return next; }
    public void setNext(LinkedList.Link node)
    { next = node; }
}
```
Nested top-level class & interface

• Note how we may import a static nested class…

import LinkedList.*;  // Import nested classes.
class Node implements Link {
    private int i;
    private Link next;
    public Node(int i) { this.i = i; }
    public Link getNext() { return next; }
    public void setNext(Link node) { next = node; }
}
-class files and JVM

• When we compile a Java source (-.java) file with nested class defined, note the class (-.class) files generated.

• When we compile LinkedList.java, we generate
  
  `LinkedList.class`
  
  `LinkedList$Link.class`

• The *Java Virtual Machine* (JVM) knows nothing about nested classes. But the Java compiler uses the “$” insertion to control the class name space so that JVM would interpret the -.class files correctly.
Inner Classes

- Nested classes which are not static are called inner classes. They appear in these ways:
  - member class – declared as a member of the containing class;
  - local class – declared as a local variable within a method body of the containing class;
  - anonymous class – local class with no declaration (and therefore no name), but an instance can be created.
- An inner class is therefore associated with the containing class in which it is defined.
Inner Classes Examples

class A { … };
class B {
  class MC { … }; // Example of Member Class: MC
  public void meth( )
  {
    class LC { … }; // Example of Local Class: LC
    // ... creating an object of an Anonymous Class
    // ... which is a subclass of A.
    A a = new A() { void meth( ) { … } };
    …
  }
}
// ... An inner class is associated with a containing class.
// ... Each inner class object is also associated with an
// ... object of the containing class.
Member Class

- Use a member class (instead of a nested top-level class) when the member class *needs access to the instance fields* of the containing class.
- Consider the LinkedList class we had before. If we have a linked-list (a LinkedList object), and we want to have an enumerator (an Enumerator object) to iterate through the elements in the linked-list, the Enumerator object must be associated with the LinkedList object.
- Let us first define the LinkedListEnumerator as a helper class at the top-level, and then make it into a *Member Class* within the LinkedList class.
public class LinkedList {
    public interface Link {
        public Link getNext();
        public void setNext(Link node);
    }
    private Link head; // ...helper class cannot get to head.
    public Link getHead() { return head; } // ...added.
    public void insert(Link node) { ... };
    public void remove(Link node) { ... };
}

class Node implements LinkedList.Link {
    int i;
    private LinkedList.Link next;
    public Node(int i) { this.i = i; }
    public LinkedList.Link getNext() { return next; }
    public void setNext(LinkedList.Link node) {
        next = node;
    }
}
LinkedList and Enumerator

class LinkedList Enumerator {
    private LinkedList list;
    private LinkedList.Link current;
    public LinkedList Enumerator (LinkedList ll) {
        list = ll;
        current = list.gethead();
    }
    public boolean hasMoreElements() {
        return (current != null);
    }
    public LinkedList.Link nextElement() {
        LinkedList.Link node = current;
        current = current.getNext();
        return node;
    }
}

• Observe that LinkedList Enumerator is a helper class; each of its object is associated with a LinkedList object (ref: constructor); we then want to make it into a Member Class within the LinkedList Class...
public class LinkedList {
    public interface Link {
        public Link getNext();
        public void setNext(Link node);
    }
    private Link head; // helper class can get to head now...
    public void insert(Link node) { ... };
    public void remove(Link node) { ... };
    public class Enumerator {
        private Link current;
        public Enumerator() { current = head; }
        public boolean hasMoreElements() {
            return (current != null);
        }
        public Link nextElement() {
            Link node = current;
            current = current.getNext();
            return node;
        }
    }
}
Member Class

- Member class methods have *access to all fields and methods* of the containing class.
- In the Member class method definition, a field/method name is resolved first in the **local scope**, and then the **class scopes** – first the **inherited** classes and then the **containment classes**, unless there is explicit scope specified.
- **Explicit** access: `<ClassName>.this.<FieldName>`
- A Member Class **cannot be named the same** as one of its containing classes.
- A Member Class **cannot have static** fields/methods.
Member Class: accessing fields

```java
public class A {
    public String name = "A";
    public class B {
        public String name = "B";
        public class C {
            public String name = "C";
            public void print_names() {
                System.out.println(name);
                System.out.println(this.name);
                System.out.println(C.this.name);
                System.out.println(B.this.name);
                System.out.println(A.this.name);
            }
        }
    }
}
...
// ...prints out: C
A a = new A();  // C
A.B b = a.new B();  // C
A.B.C c = b.new C();  // B
c.print_names();  // A
```
Local Class

• A **Local Class** is a class declared and defined within the local scope (like a local variable) of a method definition of the containing class. The class name is *only visible within the local scope*.

• A **Local Class** is similar to a Member Class – and must obey all the restriction of a Member Class – but the methods of a local class **cannot** access other **local variables** within the block **except when they are final**.

• A **Local Class** cannot be declared **public**, **private**, **protected**, or **static**.

• Common use of local classes is for **event listeners** in Java 1.1, using the new AWT event model.
Local Class example

class A { protected char a = 'A'; }
class B { protected char b = 'B'; }
public class C extends A {
    private char c = 'C';
    public static char d = 'D';
    public void createLocalObject(final char e) {
        final char f = 'F'; int i = 0;
        class LocalClass extends B {
            char g = 'G';
            public void printVars() {
                System.out.print(g); // (this.g) … of this
                System.out.print(f); // f … final local
                System.out.print(e); // e … final local
                System.out.print(d); // (C.this.d) containing
                System.out.print(c); // (C.this.c) containing
                System.out.print(b); // b … inherited from B
                System.out.print(a); // a … inherited from A
            }
        }
        LocalClass lc = new LocalClass();
        lc.printVars();
    }
    public static void main(String[] as) {
        C c = new C();
        c.createLocalObject('E');
    } // …prints out: GFEDCBA
Anonymous Class

- An *Anonymous Class* is essentially a *local class without a name*.
- Instead of *defining a local class* and then *create an instance* (probably the only object) of the class for use, Anonymous Class does that in just one step.
- An Anonymous Class must then also obey all the restrictions of a local class, except that in using new syntax, an *Anonymous Class* is *defined* at the point an instance of the class is *created*.
- Common use of anonymous classes is in the *adapter classes* used as event listeners in GUI programming using the AWT event model.
Anonymous Class: an example

```java
import java.io.*;
public class Lister {
    public static void main(String[] arg) {
        File f = new File(arg[0]);
        Class JavaFilter extends FilenameFilter {
            public boolean accept(File f, String s) {
                return( s.endsWith(".java");)
            }
        }
        JavaFilter jfilter = new JavaFilter();
        String[] list = f.list(jfilter);
        for (int i=0; i < list.length; i++)
            System.out.println(list[i]);
    }
}
```
Anonymous Class: an example

```java
import java.io.*;
public class Lister {
    public static void main(String[] arg) {
        File f = new File(arg[0]);
        String[] list = f.list(
            new FilenameFilter() {
                public boolean accept(File f, String s) {
                    return (s.endsWith(".java"));
                }
            }
        );
        for (int i=0; i < list.length; i++)
            System.out.println(list[i]);
    }
}
```
Anonymous Class

Some **style guidelines** for using anonymous class:

- The class has a very short body.
- Only one instance of the class is needed at a time.
- The class is used immediately after it is defined.
- A name for the class does not make the code easier to understand.

Since an anonymous class has *no name*, it is not possible to define a constructor. Java 1.1 has a new feature – **instance initializer** – to conveniently initialize the object created for an anonymous class. But the feature applies to **all** classes.
Instance Initializer

• An instance initializer is a block of code (inside braces) embedded in a class definition, where we normally have definition of fields and methods.

```java
public class InitializerDemo {
    public int[] array;
    {
        array = new int[10];
        for (int i=0; i<10; ++i) array[i] = i;
    }
}
```

• There can be more than one instance initializer in the class definition.

• The instance initializers are executed in order, after the superclass constructor has returned, before the constructor of the current class is called.
Exercise

- Provided: an abstract which defines a banking account with its abstract class.
- To do: write a program that subclasses the abstract class, defines the abstract methods, and provides some additional functionality.
08/23/2000

Introduction to Java