Java Inner Classes

David I. Schwartz

COMS/ENGRD 211

Step 1: Class Declarations

1.1 Non-Generic

```
modifiers class classname extends-clause implements-clause {

   fields
   enums
   initializers
   constructors
   methods
   classes
   interfaces
}
```

Members:

- fields
- methods
- enums
- classes
- interfaces

Note that members can be `static`.

1.2 New Concepts

What you need to know:

- ***Inner classes***: classes that you can write inside another class. Common applications include iterators and GUIs.
- ***Enums***: define named constants (e.g., a type called `Color` that has values `BLUE`, `RED`, …). We will save enums for another document.

What you don't really need to know:

- ***Inner interfaces***: Yes, you can really write an interface inside a class. The rules get complex. Save for a really, really rainy day.
- ***Initializers***: We tend not to cover them, but they're actually rather useful and help to hint at anonymous classes. Imagine using a method body without a header. Why bother? You might wish to set data when creating an object for the first time. Rather than calling a method, you can use a statement block to set the data.

Initializer example:

```
public class Initializers {
   public static void main(String[] args) {
      new Test().print1(); // output: 0123456789
      new Test().print2(); // output: 01234
   }
}

class Test {
   public final int N=10;
   private int[] x=new int[N];
   { for (int i=0; i<N; i++) x[i]=i; }
   public static final int L=5;
   private static int[] y=new int[L];;
   static { for (int i=0; i<L; i++) y[i]=i; }

   public void print1() {
      for (int i=0; i< x.length; i++)
         System.out.print(x[i]);
      System.out.println();
   }

    public void print2() {
   for (int i=0; i< y.length; i++)
       System.out.print(y[i]);
   System.out.println();
    }

}
```

1.3 Generic Classes and Interfaces

You can write a class or interface that serves as a template to make other classes.

Generic class syntax:

```
modifiers class classname<Type1, …, TypeN> baseclause {
   classbody
}
```

We will not deal with generic classes at this point.

Step 2: Levels of Classes

2.1 Top-Level (or Outer) Class

- You can put a class inside an another class.
- A class that contains other classes is a ***TLC***.
- The classes you have seen up until now are TLCs.

2.2 Nested Class

Nested class:

- Class declared inside another class.

Two kinds of nested classes:

- ***Member class***: class declared at the member-level of a TLC.
- ***Local class***: class declared inside a method, constructor, or initializer block.

2.3 Inner Class

Inner class (IC) refers to two special kinds of nested class:

- Non-static member class (member class with no `static` modifier).
- Local class inside a non-static member of a TLC.

Why called inner class?

- Because an object made from the class will contain a reference to the TLC.
- Use ***`TLC.this.member`*** from inside inner class to access member of TLC.

Restrictions:

- Inner class fields can be `static`, but then must also be `final`.
- No `static` methods or other inner classes (same for other members?)
- See language references for even more details.

Handy way to think of inner classes inside a TLC:

- At the member level:
  - just like a variable or method.
  - called ***member class***.
- At the statement level:
  - just like a statement in a method
  - called ***local class***
- At the expression level:
  - just like an expression
  - called ***anonymous class***

Step 3: Member Class (Member Level)

3.1 Rules

Structure:

```
public class OuterClass {
   tlc_members

   public class InnerClass {
      mc_members
   }

}
```

When to use?

- The inner class generates objects used specifically by TLC.
- The inner class is associated with, or "connected to," the TLC.

Example:

```
class List {

   class Node {
   }

}
```

How does visibility work?

- The inner class can be **public**, **private**, **protected**, or package.
- Instances of the inner class type have access to all members of the outer class (including private and static members).

Some restrictions:

- Cannot have same name as TLC or package (not that you would want to!).
- Cannot contain **static** members; can have **static final** fields (constants).

How do you use a member class?

- Every member class is associated with instance of TLC.
- Valid:
```
OuterClass oref = new OuterClass();
OuterClass.InnerClass iref = oref.new InnerClass()
iref.doSomething();
new OuterClass().new InnerClass();
```
- Not valid:
```
InnerClass iref = new InnerClass();
iref.doSomething();
```


Internal references with **this**:

- Inside inner class, the **this** refers to current instance of the inner class.
- To get to current instance of TLC, save the TLC's **this** as field in the TLC or simply use ***TLC*.this**.

Some inheritance:

- Be careful to distinguish between class and containment hierarchies!
- Inner classes do inherit.
- Can use ***TLC*.super.*member*** to access TLC's ***member***.

3.2 Example

```
public class MemberClass {
   public static void main(String[] args) {

   // one way:
   OC a = new OC();
   OC.IC b = a.new IC();
   b.print(); // outputs 3

   // another way:
   new OC().new IC().print(); // outputs 3

   }
}

class OC {

   private int x = 1;

   public class IC {
      private int y = 2;
      public void print() {System.out.println(x+y);}
   }

}
```

3.3 Example

```
public class MemberClass2 {
   public static void main(String[] args) {

      new OC().new IC().print();

   }
}
```

```
class OC {

   private int x = 1;
   private int y = 2;

   public class IC {
      private int x = 3;
      private int y = 4;

      public void print() {

         // 3 + 1 -> 4
         System.out.println(this.x+OC.this.x);
         // 4 + 4 -> 8
         System.out.println(y+this.y);

      } // method print

   } // class IC

} // class OC
```

3.4 Example

```
public class Memberclass3 {
   public static void main(String[] args) {
      new OC().new IC().print(); // Output: IC, OC

   }
}

class OC {

   public class IC {
      public String toString() { return "IC"; }
      public void print() {
         System.out.println(this);
         System.out.println(OC.this);
      }
   }

   public String toString() { return "OC"; }

}
```


3.5 Example

```
public class Memberclass4 {

   public static void main(String[] args) {

      new OC2().new IC().print(); // output: 2

   }

}

class OC {

   public class IC {

      private int x = 2;

      public void print() { System.out.println(x); }

   }

}

class OC2 extends OC { }
```

Step 4: Local Classes (Statement Level)

4.1 Rules

Local class location:

- Statement level declaration.
- Usually written in methods. See also constructors and initializers.

Scope:

- Local to block.
- Can access all members of the TLC.
- Actually, things can get confusing here!
  - An object of local class might persist after method ends.
  - Java does have rules for dealing with the matter.

Example structure:

```
public class TLC {
   tlc_members

   methodheader {
      statements

      public class InnerClass {
         ic_members
      }

      statements
   }

   moreTLCmethods

}
```

More restrictions:

- Cannot be used outside of block.
- No modifiers.
- Enclosing block's variables must be `final` for local class to access.
- No `static`, but can have `static final` (constants).
- Terminate with a semicolon! The class is effectively an expression statement.
- Cannot have same name of TLC.

4.2 Example

```
public class LocalClass {
   public static void main(String[] args) {
      new OC().print();
   }
}

class OC {

   public void print() {

      final String s = "test: ";

      class Point {
         private int x;
         private int y;
         public Point(int x,int y) { this.x=x; this.y=y; }
         public String toString() { return s+"("+x+","+y+")"; }
      };

      System.out.println(new Point(1,2));

   } // method print

} // class OC
```

Step 5: Anonymous Class

5.1 Rules

Location and structure:

- Defined and created at *expression* level.
- So, has no name and no modifiers.
- Syntax:

```
new classname ( argumentlist ) { classbody }
new interfacename ( argumentlist ) { classbody }
```

Adapter class:

- Adapter class defines code that another object invokes.
- Common in GUIs and iterators.

Some restrictions:

- No modifiers.
- No `static`, but can have `static final` (constants).
- No constructors, but can use initializers for same purpose! (See Section 1.2.)

When to use?

- Class has very short body.
- Only one instance of class needed.
- Class used right after defined; no need to create new class.

5.2 Example

How to create an array "on the spot" with values? Use initializer list:

```
int[] = { 1 , 2 , 3 } ;
```

Can you return an initializer list?

```
int[] doStuff() {
   return { 1 , 2 , 3 } ;
}
```

Looks good, but it won't work! To "return an array of data" (a reference to a newly created array with assigned values), use an anonymous array, which is effectively an anonymous class!

```
return new int[] { 1 , 2 , 3 };
```

The pattern is identical: `new classname { stuff } ;`. Note also that the anonymous array is the expression of the return statement and is thus expression-level!

5.3 Example

In example below, we print a `Point` again. But, we cannot say new `Point`, because we have not defined a `Point` class. Instead, I use a placeholder, class `Object`. You will often find yourself using interface names instead.

```
public class AnonymousClass {
   public static void main(String[] args) {

      new OC().print();

    }
}

class OC {

   public void print() {

      final String s = "test: ";

      System.out.println(new Object() {
         private int x=1;
         private int y=2;
         public String toString() { return s+"("+x+","+y+")"; }
      } );

    }

}
```