Lecture 7:
Nested Classes

Wholeness of the Lesson

Nested classes allow classes to play the roles of instance variable, static variable and local variable, providing more expressive power to the Java language. Likewise, it is the hidden, unmanifest dynamics of consciousness that are responsible for the huge variety of expressions in the manifest world.

Outline of Topics

1. Definitions of Nested and Inner Class
2. Four Types of Nested Classes: Member, Static, Local, and Anonymous
3. Using Lambda Expressions in Place of Anonymous Inner Classes
   - Syntax of lambda expressions
   - Lambdas as implementers of functional interfaces

Outline of Topics

1. Definitions of Nested and Inner Class
2. Four Types of Nested Classes: Member, Static, Local, and Anonymous
3. Using Lambda Expressions in Place of Anonymous Inner Classes
   - Syntax of lambda expressions
   - Lambdas as implementers of functional interfaces

Definition and Types of Nested Classes

- A class is a *nested class* if it is defined *inside* another class. (Note: This is different from having multiple classes defined in the same file.)
- Four kind of nested classes:
  - static
  - member
  - local
  - Anonymous
- Of these, member, local, and anonymous are all called *inner* classes.
- An alternative, but equivalent, definition of *inner class* is “any non-static nested class”
- *Demos for nested classes are in package* `lesson7.nestedclasses`

Definition and Types of Nested Classes

- An *inner* has access to all members (variables and methods) of its enclosing class.
- It is possible to make inner classes `private` and also `static` (see below) – these keywords cannot be used with ordinary classes.

Outline of Topics

1. Definitions of Nested and Inner Class
2. Four Types of Nested Classes: Member, Static, Local, and Anonymous
3. Using Lambda Expressions in Place of Anonymous Inner Classes
   - Syntax of lambda expressions
   - Lambdas as implementers of functional interfaces

Member Inner Classes

```
public class Classroom {
        private String roomNo;
        // member inner class
        class Projector {
                private double price;
                Projector(double price) {
                        this.price = price;
                }
                public String locate() {
                        return roomNo;
                }
        }

        public void aMethod() {
                Projector p = new Projector(800.8);
                System.out.println(p.price);
        }
}
```

Member Inner Class Syntax and Rules

- Member inner classes, like other members of the class, can be declared public, private, protected , or may have package level access. In the example, they have package level access.

- Member inner classes have access to all fields and methods of the enclosing class, including private fields and methods. No explicit reference to an enclosing class instance is needed.

  In the example, notice how the `roomNo` are accessed.

- Likewise, the outer class can access private variables and methods in the inner class, but only with reference to an inner class instance that has already been created.

- To access the methods and variables of a member inner class, it is necessary to explicitly instantiate it – it is *not* instantiated automatically when the enclosing class is instantiated.

```
public class MyClass {
   private String s = "hello";
   MyInnerClass inner;
   public static void main(String[] args){
       new MyClass();
   }
   MyClass() {
       System.out.println(inner.anInt);//NullPointerException
       inner = new MyInnerClass();
       System.out.println(inner.anInt); //OK
   }
   class MyInnerClass{
       private int anInt = 3;
       void innerMethod(){
           System.out.println(s);
        }
   }
}
```

- In a member inner class, no variable or method may be declared static. (static final variables are allowed.) (By contrast, static members are allowed in a static nested class.)

- Like ordinary classes, when a member inner class is instantiated, it has an implicit parameter `'this'`. The `'this’` of the enclosing class is accessible from within the inner class . The `‘this’` of the inner class is accessible from within itself, but *not* from the enclosing class.
- Demo: `lesson7.innerthiskeyword`

Example

```
Class MyOuterClass {
    MyInnerClass inner;
    private String param;
    MyOuterClass(String param) {
        inner = new MyInnerClass("innerStr");
        this.param = param; // the outer class version of this
    }
    void outerMethod() {
        System.out.println(inner.innerParam);
        inner.innerMethod();
        //String t = inner.this.innerParam; //compiler error
    }
    class MyInnerClass{
        private String innerParam;
        MyInnerClass(String innerParam) {
            //the inner class version of 'this'
            this.innerParam = innerParam;
        }
        void innerMethod() {
            //accessing enclosing class's version of this
            System.out.println(MyOuterClass.this.param);
            //same as the following
            System.out.println(param);
        }
  }
  public static void main(String[] args) {
    (new MyOuterClass("outerStr")).outerMethod();
  }
}
```

```
//OUTPUT:
innerStr
outerSt
outerStr
```

- If the member inner class is sufficiently accessible (i.e., not private), it can be instantiated by a class other than the enclosing class, as long as an instance of the enclosing class has already been created.

  Example:

```
class ClassA {
    class InnerClassA {
    }
}

class ClassB {
   ClassB() {
        ClassA a = new ClassA();

        //This is OK
        ClassA.InnerClassA innerA = a.new InnerClassA();


        //This is illegal,'new' requires an enclosing instance
        ClassA.InnerClassA innerA =new ClassA.InnerClassA();
  }
 }
```

- **Best Practice.** A member inner class is typically a small specialized "assistant" that is exclusively owned by its enclosing class. Consequently, it is not good practice to access a member inner class from outside the enclosing class, since this undermines the overall purpose of this type of nested class.
- In the Java API, `AbstractList.Itr` and `ArrayList.Itr` are examples of member inner classes.

Main Point

Inner classes – a special kind of nested class – have access to the private members of their enclosing class. The most commonly used kind of inner class is a *member* inner class. Likewise, when individual awareness is awake to its fully expanded, self-referral state, the memory of its ultimate nature becomes lively.

Static Nested Classes

- If a nested class is defined using the `static` keyword, it becomes a static nested class.
- Static nested classes do not have direct access to instance variables and methods of the enclosing class. A static nested class is in effect a top level class that has been "packaged" differently; it has the same access to the enclosing class variables and methods as another class located in the same package. (Static nested classes have access to private variables and methods of the enclosing class through an instance of enclosing class.)

```
  public class Main {
    public int i = 4;
    public int getInt() {
      return 3;
    }
    static class NestedClass {
      public void innerMethod() {
        int j = i;          //compiler error
        int k = getInt(); //compiler error
      }
    }
}
```

- It is possible to declare static variables and methods in a static nested class; however, static nested classes may also have instance variables and methods.
- As with member inner classes, the enclosing class of a static nested class has access to the nested class's private variables and methods, with reference to an instance.
- Unless a static nested class is declared private, other classes can instantiate it, but the syntax is different from that for member inner classes. For example:

```
MyClass.MyStaticNestedClass cl =
                  new MyClass.MyStaticNestedClass();
```

- demo: `lesson7.staticnested`

```
public class MyClass {
    private String s = "hello";
    public static void main(String[] args){
        new MyClass();
    }
    MyClass() {
        //access static methods in the usual way
        MyStaticNestedClass.myStaticMethod();

        //access instance methods in the usual way too
        //except that now private methods are also accessible
        MyStaticNestedClass cl = new MyStaticNestedClass();
        cl.myOtherMethod();

        //as with inner classes, private instance vbles are accessible
        int y = cl.x;
    }
    static class MyStaticNestedClass {
        private int x = 0;
        static void myStaticMethod() {
            String t = s; //compiler error -- no access
        }
        private void myOtherMethod() {
        }
    }
}
```

```
class AnotherClass {
   public static void main(String[] args){
      MyClass.MyStaticNestedClass cl =
               new MyClass.MyStaticNestedClass(); //OK
      }
}
```

- **Best Practice.** Static nested classes should be thought of as ordinary ("top-level" or "first class") classes that are "privately packaged". Usually not accessed from outside, but it's not necessarily bad practice to do so since static nested classes are "top level" classes.
  - In the Java API, `LinkedList.Node` and `HashMap.Node` are examples of static nested classes.

Local Inner Classes

- Local inner classes are defined entirely within the body of a method.

- Access specifiers (public, private, etc) are not used to affect access of the inner class since access to the inner class is always restricted to the local access within the method body.
- Local inner classes have access to instance variables and methods in the enclosing class; they also have access to *local variables* – variables inside the method body, as well as parameters passed in to the method – these local variables are *effectively final* (this means that local inner class cannot change the values of these variables during execution).

Demo: `lesson7.nestedclasses.local`

Advantages of Local Inner Classes

Two reasons for preferring local inner classes to member inner classes:

1. A local inner class is defined precisely where it is needed. This makes it easier to maintain, and makes code easier to follow

2. A local inner class can be used only for one purpose – the purpose of its enclosing method. Therefore, local inner classes are used to provide *strong encapsulation*.

Issues Concerning Local Inner Classes

- *Local Inner Classes Should Be Small.* When the inner class requires more than a small amount of code, it should not be squeezed inside the body of a method – the method body would become to big, harder to read, and maintenance of the method becomes more difficult.

- *Avoid Including Local Inner Classes Involved in a Loop.* If the method in which the local inner class is defined is called from a loop, the inner class will be instantiated repeatedly. This behavior can be costly – in such cases, it is usually better to move the class out as a member inner class (or even a top-level class) and instantiate it just once; if its values need to take on different values as a loop executes, these can be set using setter methods.

Anonymous Inner Classes

- An inner class declared without a class name is known as an anonymous inner class. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface.
- Sometimes an anonymous inner class is defined – like local inner classes – within a method body. In that case, the code is even more compact, and has the same advantages as a local inner class.
- See package `lesson7.moreanonymous` for examples.

Advantages to Anonymous Inner Classes

- They provide the same strong encapsulation as local inner classes.
- They have wider applicability than local inner classes: Whenever either a subclass of a class or an implementation of an interface is needed, anonymous inner classes can be used – no enclosing method is necessary.

Outline of Topics

1. Definitions of Nested and Inner Class
2. Four Types of Nested Classes: Member, Static, Local, and Anonymous
3. Using Lambda Expressions in Place of Anonymous Inner Classes
   - Syntax of lambda expressions
   - Lambdas as implementers of functional interfaces

Example: Implementing a Comparator with a Lambda Expression (for jse8 and later)

```
Arrays.sort(emps, new Comparator<Employee>() {

   @Override
   public int compare(Employee o1, Employee o2) {
      return o1.getName().compareTo(o2.getName());
   }

});
```

About the Code Sample

- Note: `Comparator` has just one (abstract) method `compare`. (It is a functional interface.)
- The `compare` method takes two arguments of same type (Employee) and maps it to a block of code, representing how to compare two employee objects.
- Lambda Expressions are syntax shortcuts for implementation of functional interfaces. We can use a lambda expression to replace this implementation of `Comparator` in the example.

Anatomy of a Lambda Expression

A lambda expression has three parts:

parameters [zero or more]
->
code block [if more than one statement, enclosed in curly braces { . . . }]
[may contain *free variables*; values for these supplied
by local or instance variables]

Using Lambda Expression

- Recall: Associated with the argument `e1 and e2` was the block of code

```
  {
     return o1.getName().compareTo(o2.getName())
  }
```

- A lambda expression abstracts from the inner class syntax just this functional relationship:

```
     (e1, e2) -> e1.getName().compareTo(e2.getName())
```

- More about `Comparator` in Lesson 8.

Application of Nested Classes: Implementing the Singleton Pattern

Demo: `lesson7.singletons`

There are several ways to implement the singleton pattern:

- Use lazy initialization to instantiate a private static instance variable. (Not threadsafe.)
  - `- SingleThreadedSingleton.java`
- Store instance as a public static constant, constructed when class is loaded.
  - `- SingletonAsPublicStatic.java`
- Implement as an Enum. Also constructed when enum is loaded.
  - `- SingletonAsEnum.java`
- Use Spring's Singleton Holder pattern, whereby the singleton is stored as a static variable of a static nested class. Permits lazy initialization and is threadsafe.
  - `- SingletonAsStaticNestedClass.java`

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Inner classes retain the memory of their "unbounded" context

1. A *nested class* is a class that is defined inside another class.
2. An *inner class* is a nested class that has full access to its context, its enclosing class.

---

3. **Transcendental Consciousness:** TC is the unbounded context for individual awareness.
4. **Wholeness moving within itself**: When individual awareness is permanently and fully established in its transcendental "context" – pure consciousness – every impulse of creation is seen to be an impulse of one's own awareness.