

SUPSI

Nested Classes and Interfaces

Object Oriented Programming

Tiziano Leidi

08.11.2021

Nested Classes and Interfaces

In Java, it is possible to define a class or an interface **within another class or interface**.

Nested classes and interfaces allow to:

- group classes/interfaces for more **modularity**
- increase the use of **encapsulation**,
- develop more **readable and maintainable code**.

Nested Classes

When a class is defined inside another class or interface, it is a nested class.

Nested classes inside classes are divided into two categories:

- static classes are called: **static-nested classes**
- instance classes are called: **inner classes**

In addition, **nested classes can be defined inside interfaces.**

Nested Class in Class

```
class OuterClass {  
    // ...  
  
    static class StaticNestedClass {  
        // ...  
    }  
  
    class InnerClass {  
        // ...  
    }  
}
```

Nested Class in Class

The scope of a class nested in another class is **bounded by the scope of its enclosing class**.

Like fields and methods, a nested class **is a member of its enclosing class**. As a consequence, classes inside other classes **can be private, package-private, protected and public**.

Nested Class in Class

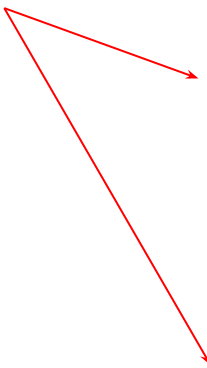
Rules of nested classes are as follow:

- **A static-nested class** has access only to the static fields and methods of the class that contains it (including private ones).
- **An inner class** has access to all the fields and methods of the class that contains it (both static and non-static, including private ones).

However, an inner class can never contain a static field or a static method (except constants: static final)!

```
public class OuterClass {  
    private static void externalStaticMethod() {  
        System.out.println("Execute operation");  
    }  
  
    private void externalMethod() {  
        System.out.println("Execute operation");  
    }  
  
    private static class StaticNestedClass {  
        private static void internalStaticMethod() {  
            externalStaticMethod();  
            externalMethod();  
        }  
  
        private void internalMethod() {  
            externalStaticMethod();  
            externalMethod();  
        }  
    }  
}
```

Not allowed,
the method is
not static.




```
public class OuterClass {  
    private static void externalStaticMethod() {  
        System.out.println("Execute operation");  
    }  
}
```

```
private void externalMethod() {  
    System.out.println("Execute operation");  
}
```

```
private class InnerClass {  
    private static void internalStaticMethod() {  
        externalStaticMethod();  
        externalMethod();  
    }  
}
```

Not allowed,
inner classes
can't have static
methods.



```
private void internalMethod() {  
    externalStaticMethod();  
    externalMethod();  
}  
}  
}
```


Static Nested Classes

A static-nested class cannot directly access instance fields or methods defined in the class that contains it. For such an access a reference to the object of the outer class is needed.

The contained class is nested with the goal of packing it together with the outer class.

To call the nested class, use the syntax with the dot.

For example, to instantiate an object of the nested class:

```
OuterClass.StaticNestedClass nestedObject  
    = new OuterClass.StaticNestedClass();
```

```
class Calculator {  
    private static class Sum implements Operation {  
        @Override  
        public double execute(double a, double b) {  
            return a + b;  
        }  
    }  
}  
  
private static class Subtract implements Operation {  
    @Override  
    public double execute(double a, double b) {  
        return a - b;  
    }  
}  
  
private static class Multiply implements Operation {  
    @Override  
    public double execute(double a, double b) {  
        return a * b;  
    }  
}  
// ...
```

```
interface Operation {  
    double execute(double a, double b);  
}
```

```
// ...  
  
private Operation[] operations = new Operation[3];  
private int cnt = 0;  
  
public Calculator() {  
    addOperation(new Sum());  
    addOperation(new Subtract());  
    addOperation(new Multiply());  
}  
  
private void addOperation(Operation operation) {  
    operations[cnt++] = operation;  
}  
  
public void showResults(double a, double b) {  
    for (Operation operation : operations)  
        System.out.println("Result: " + operation.execute(a, b));  
}  
}  
  
public class CalculatorTest {  
    public static void main(String[] args) {  
        new Calculator(). showResults(42.42, 666.666);  
    }  
}
```

Inner Classes

An inner class is associated with an instance of the class that contains it.

Each instance of the inner class exists within an instance of the outer class.

In order to instantiate an object of the inner class, **an object of the outer class needs first to be instantiated:**

```
OuterClass outerObject = new OuterClass();  
OuterClass.InnerClass innerObject =  
    outerObject.new InnerClass();
```

```
class Outer {
    private int x = 5;

    public Outer(int x) {
        this.x = x;
    }

    class Inner {
        private int y = 7;

        public Inner(int y) {
            this.y = y;
        }

        public void show() {
            int sum = x + y;
            System.out.println("Sum: " + sum);
        }
    }
}

class Main {
    public static void main(String[] args) {
        Outer.Inner in = new Outer(5).new Inner(7);
        in.show();
    }
}
```

Nested Class in Interface

One of Java's stranger language features is the **ability to nest a class inside an interface**.

Despite its strangeness, being able to nest a class inside an interface can be useful, especially **when there is a tight relationship** between the enclosing interface and the enclosed class.

Nested classes defined inside interfaces are **implicitly public and static**.

```
interface Addressable {  
    class Address {  
        private String boxNumber;  
        private String street;  
        private String city;  
  
        public Address(String boxNumber, String street, String city) {  
            this.boxNumber = boxNumber;  
            this.street = street;  
            this.city = city;  
        }  
  
        public String toString() {  
            return boxNumber + " - " + street + " - " + city;  
        }  
    }  
  
    Address getAddress();  
}
```

Nested Interfaces

An interface which is **declared inside another interface or class** is called a nested interface (or inner interface).

The main purpose is to **resolve the namespace by grouping related interfaces** (and classes) together.

Nested Interfaces

Nested interfaces **are static by default**. You don't have to mark them static explicitly as it would be redundant.

Nested interfaces declared **inside classes** can take **any access modifier** (private, package-private, protected or public). however nested interfaces declared **inside interfaces are public implicitly**.

```
interface EnclosingInterface {  
    interface EnclosedInterface1 {  
    }  
  
    interface EnclosedInterface2 {  
    }  
}
```

Nested interfaces have to be called by using the outer class or outer interface name followed by a dot, followed by the interface name:

EnclosingInterface.EnclosedInterface

Inner Classes: Additional Variants

There are two additional variants of inner classes

- local classes,
- anonymous classes.

Local Classes

A local class is **declared locally within a block of Java code** (typically a method), rather than as a member of a class. Because all blocks of Java code appear within class definitions, **all local classes are nested** within containing classes.

However, it is usually appropriate to think of them as an **entirely separate kind of nested class**. A local class has approximately the same relationship to a member class as a local variable has to an instance variable of a class.

Local Classes

Like a local variable, a local class is **valid only within the scope defined by its enclosing block.**

If a member class is used only within a single method of its containing class, there is usually no reason it cannot be coded as a local class, rather than a member class.

```
interface Linkable {
    Linkable getNext();
}

public java.util.Enumeration enumerate() {
    class Enumerator implements java.util.Enumeration {
        Linkable current;

        public Enumerator(Linkable head) {
            current = head;
        }

        public boolean hasMoreElements() {
            return (current != null);
        }

        public Object nextElement() {
            if (current == null) throw new java.util.NoSuchElementException();
            Linkable value = current;
            current = current.getNext();
            return value;
        }
    }
    return new Enumerator(head);
}
```

Local Classes

Like inner classes, local classes can access **any members, including private members**, of the containing class.

In addition, local classes can access any local variables, method parameters, or exception parameters that are **in the scope of the method and are final or effectively final**.

This is because the lifetime of an instance of a local class can be much longer than the execution of the method. Private internal copies of all local variables are thus automatically generated by the compiler.

Effectively Final Variables

Java 8 introduced a new term: **effectively final variables**.

A variable which is not declared as final but whose value is never changed after initialization is effectively final.

Local Classes

Local classes are subject to the **following restrictions**:

- A local class is visible only within the block that defines it
- Like local variables, local classes cannot be declared public, protected, private, or static
- Local classes **cannot contain static fields, methods, or classes**. The only exception is for constants (static and final)
- **Interfaces cannot be defined locally**

Anonymous Classes

Anonymous classes are a special type of local class.


Anonymous classes allow to simultaneously declare and instantiate classes.

Anonymous classes don't have a name.

When an anonymous class is declared and instantiated, a parent class is always extended (or an interface is implemented).

```
class Calculator {  
    private Operation[] operations = new Operation[3];  
    private int cnt = 0;  
  
    public Calculator() {  
        addOperation(new Operation() {  
            @Override  
            public double execute(double a, double b) {  
                return a + b;  
            }  
        });  
        addOperation(new Operation() {  
            @Override  
            public double execute(double a, double b) {  
                return a - b;  
            }  
        });  
        addOperation(new Operation() {  
            @Override  
            public double execute(double a, double b) {  
                return a * b;  
            }  
        });  
    }  
    // ...  
}
```

It is not directly instantiating the Operation interface, but an anonymous class that implements the Operation interface



```
interface Operation {  
    double execute(double a, double b);  
}
```

```
// ...  
    private void addOperation(Operation operation) {  
        operations[cnt++] = operation;  
    }  
  
    public void showResults(double a, double b) {  
        for (Operation operation : operations)  
            System.out.println("Result: " + operation.execute(a, b));  
    }  
}
```

```
public class CalculatorTest {  
    public static void main(String[] args) {  
        new Calculator(). showResults(42.42, 666.666);  
    }  
}
```

Anonymous Classes

An anonymous class:

- is not allowed to have constructors.
- can access the fields and methods of the object that contains it.
- can access the local variables of the method in which it is declared, but only if they are declared final or if they are effectively final.

```
public class TestAnonymous {  
    private static String[] fruits = new String[5];  
  
    public static void main(String args[]) {  
        fruits[0] = "apple";  
        fruits[1] = "orange";  
        fruits[2] = "pineapple";  
        fruits[3] = "mango";  
        fruits[4] = "banana";  
        int currentFruit = 2;  
  
        Person p = new Person() {  
            @Override  
            public void eat() {  
                System.out.println("Eating " + fruits[currentFruit]);  
            }  
        };  
  
        p.eat();  
    }  
}
```

```
interface Person {  
    void eat();  
}
```

Summary

- Introduction to nested classes and interfaces
- Static-nested classes: packing classes together
- Inner classes: associating class instances
- Nested interfaces: grouping interfaces together
- Local classes: classes inside methods
- Anonymous classes: on the flight instantiation of classes