Object Oriented Programing 1

Lecture 12: Java inner Classes, Abstraction and Java interfaces

By

Elubu Joseph

MSci.IS

Email: josebulinda@gmail.com

or

jose@kumiuniversity.ac.ug

Agenda

1. Java inner Classes,
2. Abstraction and
3. Java interfaces

Java Inner class

Java Inner/Nested class

an inner class is also known as nested class. Inner classes are part of nested classes. When a non-static class is defined in nested class then it is known as an inner class.

It is defined inside the class or an interface. Inner classes are mostly **used to logically group** all the classes and the interface in one place, which makes the code more readable and manageable.

Inner classes can access members of the outer class including all the private data members and methods.

Inner class Syntax

```
class OuterClass {
    //code
    class InnerClass {
    //code
    }
}
```

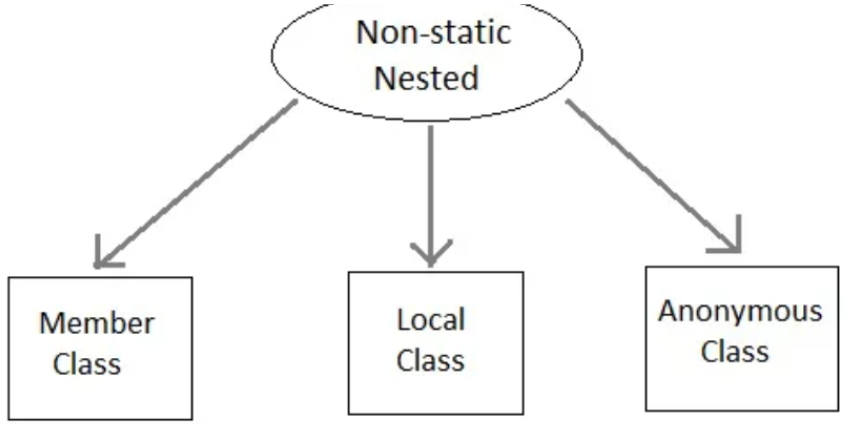
Advantages of Nested Class

1. It is a way of logically grouping classes that are only used in one place.
2. It increases encapsulation.
3. It can lead to more readable and maintainable code.

If you want to create a class which is to be used only by the enclosing class, then it is not necessary to create a separate file for that. Instead, you can add it as **"Inner Class"**

Types of Nested classes

1. Static nested class – a nested class whose definitions has static modifier
2. Non-static nested class a nested class whose definitions dose not involve static modifier
   i. Member inner class - Nested class created outside a method
   ii. Anonymous inner class- A class without any name
   iii. Local inner class- a class inside a method



Static Nested Class

Is a nested class defined within another, with static modifier applied in it. Static nested classes can **access only** static members of its outer class i.e it cannot refer to non-static members of its enclosing class directly.

Because of this restriction, static nested class is rarely used.

Non-static Nested class

Non-static Nested class is the most important type of nested class. It is also known as **Inner class**.

NOTE:

It has **access to all** variables and methods of Outer class including its private data members and methods and may refer to them directly.

However, the **Outer class cannot directly** access members of Inner class. Inner class can be declared private, public, protected, or with default access whereas an Outer class can have only public or default access.

Inner class can be created only within the scope of Outer class. Java compiler generates an error if any code outside Outer class attempts to instantiate Inner class directly.

Example of Inner class(Member class)Outer-inner

```
class Outer {
    public void display(){
        Inner in=new Inner();
        in.show();
    }
    class Inner {
        public void show(){
        System.out.println("Inner class in charge");
        }
    }
}
```

Example of Inner class(Member class)-Test

```
class OCTest {
    public static void main(String[] args){
        Outer ob = new Outer();
        ob.display();
    }
}
```

Output: `Inner class in charge`

Example of Inner class inside a method(local inner class)

```
class Outer {
     int count;
     public void display(){
          for(int i=0;i<3;i++){
            class inner{
                public void show(){
                System.out.println("Inside class "+(count++));
                }
            }
          Inner in=new Inner();
          in.show();
          }
     }
}
```

Example of Inner class(Member class)-Test

```
class OCTest1 {
    public static void main(String[] args){
        Outer ob = new Outer();
        ob.display();
    }
}
```

Output:

```
Inner class 0
Inner class 1
Inner class 2
```

Annonymous class –in the Bird interface

A class without any name is called **Annonymous class**.

```
interface Bird{
   void type();
}
```

```
public class BTest{
     public static void main(String args[]){
     //Annonymous class created
          Bird b = new Bird(){
               public void type(){
               System.out.println("Annonymous Bird");
               }
          };
          b.type();
     }
}
```

Explanation

Here a class is created which implements **Bird** interace and its name will be decided by the compiler.

This annonymous class will provide implementation of **type()** method.

Abstraction

Abstraction

is a process of hiding the complexity/implementation details from the user, but only providing the functionality to the user. In other words, the user will have the information on what the object does instead of how it does it.

To achieve abstraction one has to use the abstract class.

Abstract Class

is a class declared using **abstract keyword**. An abstract class may or may not have abstract methods. Some of the key notes we need to take about abstract class is that: -

1. It is used to achieve abstraction but it does not provide 100% abstraction because it can have concrete methods(fully defined methods with ability to be overridden).
2. An abstract class must be declared with an abstract keyword.
3. It can have abstract and non-abstract methods.
4. It cannot be instantiated.

Syntax

```
abstract class class_name {

}
```

Abstract method

Methods that are declared without any body within an abstract class are called **abstract methods**.

The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods.

Syntax:

```
abstract return_type method_name (); //No definition
```

Remember about Abstract Class and Method

Here are some useful points to remember:

1. Abstract classes are not Interfaces. They are different, we will study this when we will study Interfaces.
2. An abstract class may or may not have an abstract method. But if any class has even a single abstract method, then it must be declared abstract.
3. Abstract classes can have Constructors, Member variables and Normal methods.
4. Abstract classes are never instantiated.
5. When you extend Abstract class which has abstract method, you must define the abstract method in the child class, or make the child class abstract.

Example of Abstract class

To understand how abstruct class can be used lets create one.

In this example below, we created an abstract class A that contains a method callOne() and Using class B, we are extending the abstract class.

Normal class B inherits Abstract

```
abstract class A {
        abstract void callOne();
    }

class B extends A {
    void callOne() {
    System.out.println("Calling...");
    }
    public static void main(String[] args){
    B b = new B();
    b.callOne();
} }
```

OUTPUT: Calling…

Abstract class with non-abstract method

Abstract classes can also have non abstract methods along with abstract methods. But remember while extending the class, one has to provide definition for the abstract method.

```
abstract class A1{
    abstract void callOne();
        public void show() {
    System.out.println("Non-abstract method");
    }
}
```

Cass B1 Defines super class method callOne()

```
class B1 extends A1 {
    void callOne() {
    System.out.println("Abstract method Calling...");
    }

    public static void main(String[] args){
    B1 b = new B1();
    b.callOne();
    b.show(); }
}
```

OUTPUT

```
Abstract method Calling...
Non-abstract method
```

When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where:-

1. two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.
2. Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

Java Interfaces

Java Interfaces

Interface is a concept used to achieve abstraction in Java. This is the only way by which we can achieve full abstraction.

Interfaces are syntactically similar to classes, but you cannot create instance of an **Interface** and their methods are declared without any body. when you create an interface it defines what a class can do without saying anything about how the class will do it.

NOTE:

1. Interface use to have only abstract methods and static fields. However, from **Java 8**, interface can have **default** and **static methods** and from **Java 9**, it can have **private methods** as well.
2. When an interface inherits another interface the **extends** keyword is used whereas class use **implements** keyword to inherit an interface.

An interface

is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

NOTE

1. Along with **abstract methods**, an interface may also contain;- **constants, default methods, static methods**, and **nested types**. Method bodies exist only for default methods and static methods.
2. Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.
3. **Unless the class** that implements the interface **is abstract**, all the methods of the interface need to be defined in the class.

Advantages of Interface

1. It Support multiple inheritance
2. It helps to achieve abstraction
3. It can be used to achieve loose coupling.

Syntax:

```
interface interface_name {
      // fields
      //   abstract/private/default methods
}
```

Similarities between a class and Interface

An interface is similar to a class in the following ways —

1. An interface can contain any number of methods.
2. An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
3. The byte code of an interface appears in a **.class** file.
4. Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

Differences between a class and Interface

An interface is different from a class in several ways, including —

1. You cannot instantiate an interface.
2. An interface does not contain any constructors.
3. All of the methods in an interface are abstract.
4. An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
5. An interface is not extended by a class; it is implemented by a class.
6. An interface can extend multiple interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements
public interface NameOfInterface {
      // Any number of final, static fields
      // Any number of abstract method declarations
      // abstract/private/default methods
}
```

Implicit properties of interface

Interfaces have the following properties —

1. An **interface is implicitly abstract**. You do not need to use the **abstract** keyword while declaring an interface.
2. **Each method** in an interface is also **implicitly abstract**, so the abstract keyword is not needed.
3. **Methods** in an interface are **implicitly public**.

Example

```
/* File name : Animal.java */
 interface Animal {

    public void eat();
    public void travel();

}
```

Note.

Even if you do not add the key word public to your method declaration, the compiler adds it for you.

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Implementing Interfaces+ Example

```
/* File name : Ani1.java */
public class Ani1 implements Animal{
      public void eat(){
            System.out.println("Mammal eats");
      }
      public void move(){
            System.out.println("Mammal Moves");
      }
      public int noOfLegs(){
            return 0;
      }
      public static void main(String args[]){
            Ani1 a = new Ani1();
            a.eat();
            a.move(); }
}
```

What will be the output here?

Rules for Overriding methods defined in interfaces

When overriding methods defined in interfaces, there are several rules to be followed —

1. Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
2. The **signature** of the interface method and the same return type or subtype should be maintained when overriding the methods.
3. An implementation class itself can be abstract and if so, interface methods need not to be implemented.

Rules for implementing interface

When implementing interfaces, there are several rules -

1. A class can **implement more than one** interface at a time.
2. A class can **extend only one class**, but implement many interfaces.
3. An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class.

The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Games interface is extended by Netball and Football interfaces.

Example

Games - super Interface extended by Football

```
// Filename: Games.java
public interface Games{
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}
// Filename: Football.java
public interface Football extends Games{
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
}
```

Netball interface extends Games

```
// Filename: Netball.java

public interface Netball extends Games{
      public void homeGoalScored();
      public void visitingGoalScored();
      public void endOfPeriod(int period);
      public void overtimePeriod(int ot);
}
```

NOTE:

The **Netball** interface has **four methods**, but it **inherits two** from Games; **thus**, a class that implements Netball needs to implement all six methods.

Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed for a java class. Interfaces are not classes, therefore, an **interface can extend more than one parent interface.**

1. The **extends keyword** is used once, and the parent interfaces are declared in a comma-separated list.
2. For example, if the Netball interface extended both Games and Event, it would be declared as

```
public interface Netball extends Games, Event{

}
```

Tagging Interface

Is an interface with **no methods** inside it . The most common use of tagging interfaces occurs when the parent interface does not contain any methods.

For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as :-

```
package java.util;
public interface EventListener {    }
```

Purpose of tagging interface

There are two basic design purposes of tagging interfaces —

1. **Creates a common parent** : As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces.

   For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

2. **Adds a data type to a class** :This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

Summary

1. Java inner/Nested Classes

   (Types of inner classes,advantages and disadvantages of Inner Classes)

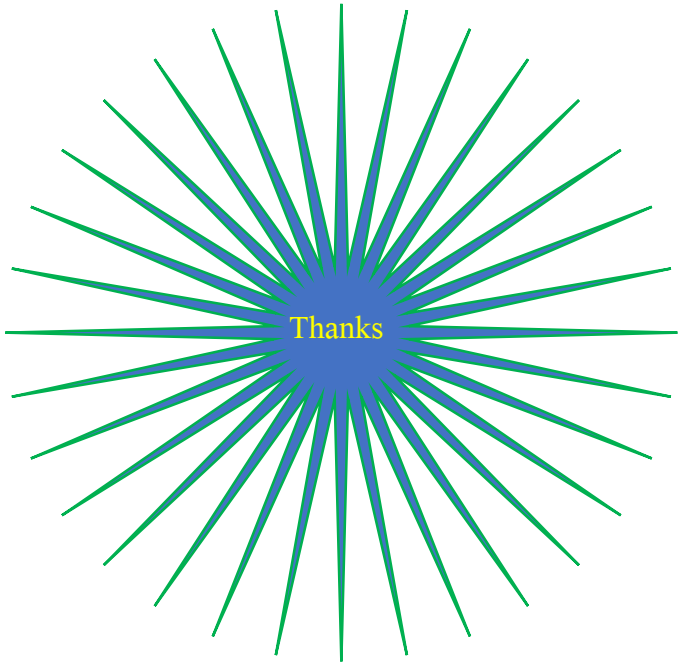
2) Abstraction

   (abstract classes and abstract methods, types of abstract methods etc.)

3. Java interfaces

   (Declaraion, implémentation, interface Vs Class etc.)

Thank you for Listening



Reference

Java abstract class and methods. Studytonight.com. (n.d.). Retrieved May 10, 2022, from https://www.studytonight.com/java/abstract-class.php

Java - Interfaces. Tutorials Point. (n.d.). Retrieved May 01 4, 2022, from https://www.tutorialspoint.com/java/java_interfaces.htm

Java interfaces. Studytonight.com. (n.d.). Retrieved April 19, 2022, from https://www.studytonight.com/java/java-interface.php