# **Advanced Programming (Java)**

# **Abstract Classes & Interfaces**

**Week 6 – Reading Material**

# **Abstract class in Java**

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

### Abstraction in Java

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

### Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

### Abstract class in Java

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

### Example abstract class

**abstract** **class** A{}

### abstract method

|  |
| --- |
| A method that is declared as abstract and does not have implementation is known as abstract method. |

### Example abstract method

**abstract** **void** printStatus();//no body and abstract

### Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

**abstract** **class** Bike{

**abstract** **void** run();

}

**class** Honda4 **extends** Bike{

**void** run(){System.out.println("running safely..");}

**public** **static** **void** main(String args[]){

 Bike obj = **new** Honda4();

 obj.run();

}

}

Output: running safely..

Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods. Let's look at an example of an abstract class, and an abstract method.

Suppose we were modelling the behaviour of animals, by creating a class hierachy that started with a base class called Animal. Animals are capable of doing different things like flying, digging and walking, but there are some common operations as well like eating and sleeping. Some common operations are performed by all animals, but in a different way as well. When an operation is performed in a different way, it is a good candidate for an abstract method (forcing subclasses to provide a custom implementation). Let's look at a very primitive Animal base class, which defines an abstract method for making a sound (such as a dog barking, a cow mooing, or a pig oinking).

public abstract Animal

{

public void eat(Food food)

{

// do something with food....

}

public void sleep(int hours)

{

//All sleep at different time for different times and in a different way

}

public abstract void makeNoise();

}

Note that the abstract keyword is used to denote both an abstract method, and an abstract class. Now, any animal that wants to be instantiated (like a dog or cow) must implement the makeNoise method - otherwise it is impossible to create an instance of that class. Let's look at a Dog and Cow subclass that extends the Animal class.

public Dog extends Animal

{

public void makeNoise() { System.out.println ("Bark! Bark!"); }

}

public Cow extends Animal

{

public void makeNoise() { System.out.println ("Moo! Moo!"); }

}

Now you may be wondering why not declare an abstract class as an interface, and have the Dog and Cow implement the interface. Sure you could - but you'd also need to implement the eat and sleep methods. By using abstract classes, you can inherit the implementation of other (non-abstract) methods. You can't do that with interfaces - an interface cannot provide any method implementations.

**Interfaces in Java**

**Definition:** An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.

Or

An interface is a collection of abstract methods

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

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

However, an interface is different from a class in several ways, including:

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* All of the methods in an interface are abstract.
* An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces.
* Implementing an interface allows a class to become more formal about the behavior it promises to provide.
* Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler.
* If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile

Declaring Interfaces:

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

public interface NameOfInterface

{

//Any number of final, static fields

public static final int a=10;

public static final int b=30;

public static final int c=100;

//Any number of abstract method declarations\

void show();

}

Interfaces have the following properties:

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

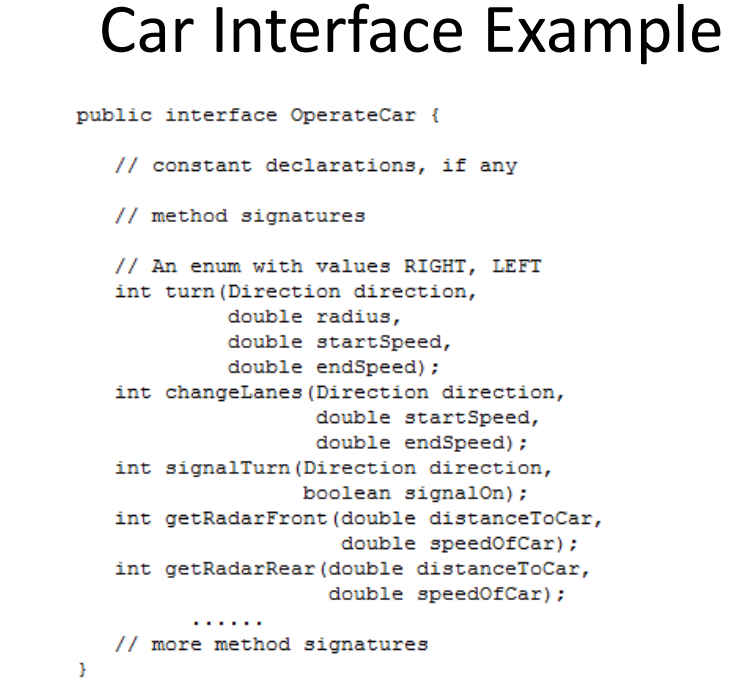
# **Difference between abstract class and interface**

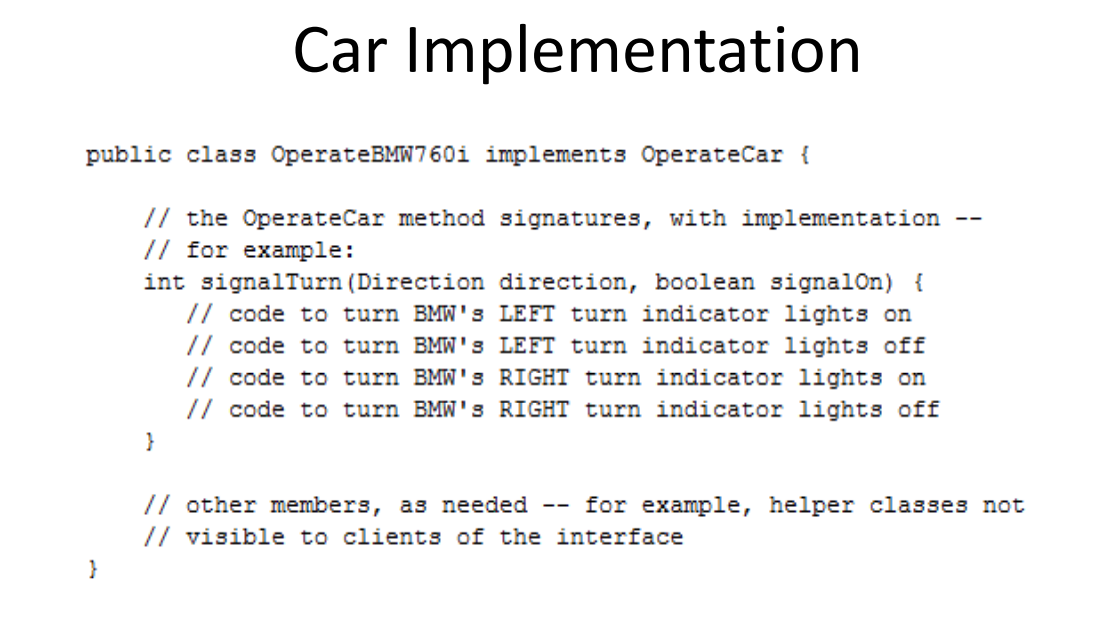
Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

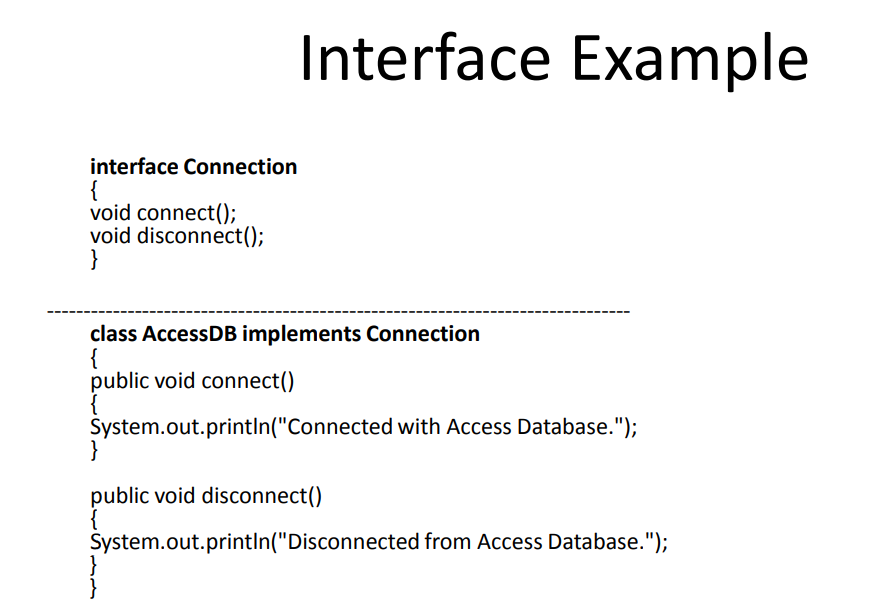
But there are many differences between abstract class and interface that are given below.

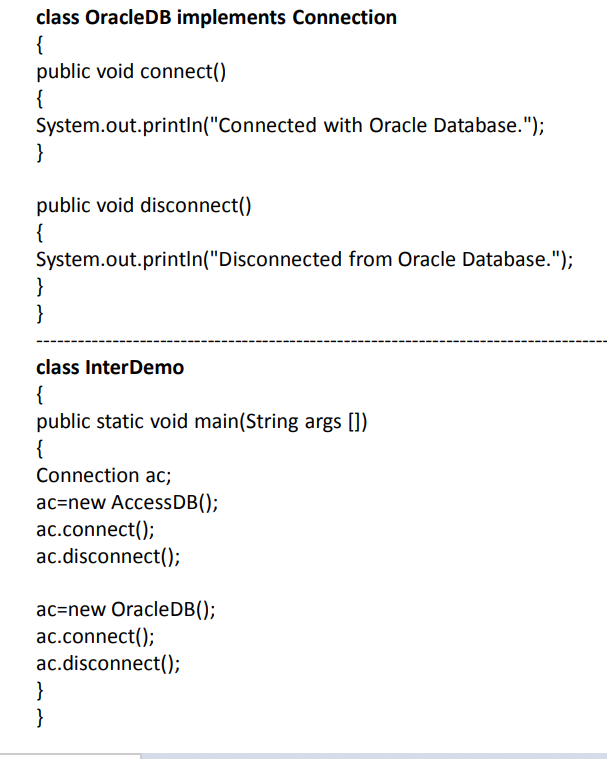
|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods or defender methods. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can have static methods, main method and constructor**. | Interface **can't have static methods, main method or**  **constructor**. |
| 5) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of**  **abstract class**. |
| 6) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 7) **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

****

****

****

****

### Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

//Creating interface that has 4 methods

**interface** A{

**void** a();//bydefault, public and abstract

**void** b();

**void** c();

**void** d();

}

//Creating abstract class that provides the implementation of one method of A interface

**abstract** **class** B **implements** A{

**public** **void** c(){System.out.println("I am c");}

}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods

**class** M **extends** B{

**public** **void** a(){System.out.println("I am a");}

**public** **void** b(){System.out.println("I am b");}

**public** **void** d(){System.out.println("I am d");}

}

//Creating a test class that calls the methods of A interface

**class** Test5{

**public** **static** **void** main(String args[]){

A a=**new** M();

a.a();

a.b();

a.c();

a.d();

}}

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Test5)

Output:

I am a

I am b

I am c

I am d

# Default Methods in Interfaces

The default methods in an interface in Java are also known as defender methods or, virtual methods.

The defender/virtual methods are those that will have a default implementation in an interface. You can define defender/virtual methods using the default keyword as −

default void display() {

   System.out.println("This is a default method");

}

There is no need to implement these defender/virtual methods in the implementing classes you can call them directly.

If you have an interface which is implemented by some classes and if you want to add a new method int it.

Then, you need to implement this newly added method in all the exiString classes that implement this interface. Which is a lot of work.

To resolve this, you can write a default/defender/virtual method for all the newly implemented methods.

## Example

Following Java Example demonstrates the usage of the default method in Java.

interface sampleInterface{

   public void demo();//abstract

   default void display() {

      System.out.println("This is a default method");

   }

}

public class DefaultMethodExample implements sampleInterface{

   public void demo() {

      System.out.println("This is the implementation of the demo method");

   }

   public static void main(String args[]) {

      DefaultMethodExample obj = new DefaultMethodExample();

      obj.demo();

      obj.display();

   }

}

## Output

This is the implementation of the demo method

This is a default method

<https://www.tutorialspoint.com/what-are-defender-methods-or-virtual-methods-in-java>

<https://www.java67.com/2017/08/java-8-default-methods-on-interface-example.html>

# Marker Interfaces

A marker interface is an interface that has no methods or constants inside it. It **provides run-time type information about objects**, so the compiler and JVM have additional information about the object. A marker interface is also called a tagging interface.

<https://www.geeksforgeeks.org/marker-interface-java/>

# Functional Interfaces in java

A functional interface in Java is **an interface that contains only a single abstract (unimplemented) method**. A functional interface can contain default and static methods which do have an implementation, in addition to the single unimplemented method.

<https://www.geeksforgeeks.org/functional-interfaces-java/>

# References:

Abstract Classes: [**https://www.youtube.com/watch?v=HvPlEJ3LHgE**](https://www.youtube.com/watch?v=HvPlEJ3LHgE)

**Interfaces:** <https://www.youtube.com/watch?v=fX1xNMBTMfg>

[**https://www.w3schools.com/java/java\_interface.asp**](https://www.w3schools.com/java/java_interface.asp)

<https://beginnersbook.com/2013/05/java-interface/>