Abstraction

In Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In java abstraction can be achieved by two ways.

1. Interface.
2. Abstract class.

Interface

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract in the java interface and not method body. It is used to achieve fully abstraction.

A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, static methods, and nested types. Method bodies exist only for static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviours of an object. And an interface contains behaviours that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

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 byte code 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.
- Interfaces can contains non abstract methods(from JDK 8).

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.

Declaring Interfaces

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

Syntax:

```
Interface InterfaceName

{

}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while 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.

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while 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.

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.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Interface Example:

```
interface MyInterface
{
   public void method1();
   public void method2();
}
class XYZ implements MyInterface
{
  public void method1()
  {
      System.out.println("implementation of method1");
  }
  public void method2()
  {
      System.out.println("implementation of method2");
  }
  public static void main(String arg[])
  {
      MyInterface obj = new XYZ();
```

```
    obj. method1();
  }
}
```

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration.

Syntax:

```
public abstract class Class_name
{
//final / static variables
// non-final / non-static variables
```

//abstract methods

// non abstract methods

}

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract. A method that is declared as abstract and does not have implementation is known as abstract method.

- **abstract** keyword is used to declare the method as abstract.
- You have to place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semoi colon (;) at the end.

Syntax:

public abstract return_type method_name();

Declaring a method as abstract has two consequences −

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Note − Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Abstract Class Example:

Creating a abstract class Person

```
public abstract class Person {

    private String name;
    private String gender;

    public Person(String nm, String gen){
        this.name=nm;
        this.gender=gen;
    }
    //abstract method
    public abstract void work();

    public String toString(){
        return "Name="+this.name+"::Gender="+this.gender;
```

```
	}
	public void changeName(String newName) {
		this.name = newName;
	}
}
```

Now, using the abstract class Person.

```
public class Employee extends Person {

	private int empId;

	public Employee(String nm, String gen, int id) {
		super(nm, gen);
		this.empId=id;
	}

	@Override
	public void work() {
		if(empId == 0){
			System.out.println("Not working");
		}else{
```

```
                System.out.println("Working as employee!!");
        }
    }

    public static void main(String args[]){
        //coding in terms of abstract classes
        Person student = new Employee("Dove","Female",0);
        Person employee = new Employee("Pankaj","Male",123);
        student.work();
        employee.work();
        //using method implemented in abstract class - inheritance
        employee.changeName("Pankaj Kumar");
        System.out.println(employee.toString());
    }

}
```

Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.

Class ---extends□> class

Class ---implements□> interface

Interface ---extends□> interface