Java Abstract Classes and Interfaces

Table of Contents

Abstract Class in java

Java Abstract classes are used to declare common characteristics of subclasses. An abstract class cannot be instantiated. It can only be used as a superclass for other classes that extend the abstract class. Abstract classes are declared with the abstract keyword. Abstract classes are used to provide a template or design for concrete subclasses down the inheritance tree.

Like any other class, an abstract class can contain fields that describe the characteristics and methods that describe the actions that a class can perform. An abstract class can include methods that contain no implementation. These are called abstract methods. The abstract method declaration must then end with a semicolon rather than a block. If a class has any abstract methods, whether declared or inherited, the entire class must be declared abstract. Abstract methods are used to provide a template for the classes that inherit the abstract methods.

Abstract classes cannot be instantiated; they must be subclassed, and actual implementations must be provided for the abstract methods. Any implementation specified can, of course, be overridden by additional subclasses. An object must have an implementation for all of its methods. You need to create a subclass that provides an implementation for the abstract method.

A class abstract Vehicle might be specified as abstract to represent the general abstraction of a vehicle, as creating instances of the class would not be meaningful.

Example of a shape class as an abstract class

```
abstract class Shape {

        public String color;
        public Shape() {
        }
        public void setColor(String c) {
                color = c;
        }
        public String getColor() {
                return color;
        }
        abstract public double area();
}
```

We can also implement the generic shapes class as an abstract class so that we can draw lines, circles, triangles etc. All shapes have some common fields and methods, but each can, of course, add more fields and methods. The abstract class guarantees that each shape will have the same set of basic properties. We declare this class abstract because there is no such thing as a generic shape. There can only be concrete shapes such as squares, circles, triangles etc.

```
public class Point extends Shape {

        static int x, y;
        public Point() {
                x = 0;
                y = 0;
        }
        public double area() {
                return 0;
        }
        public double perimeter() {
                return 0;
        }
        public static void print() {
                System.out.println("point: " + x + "," + y);
        }
        public static void main(String args[]) {
                Point p = new Point();
                p.print();
        }
}
```

Output

point: 0, 0

Notice that, in order to create a Point object, its class cannot be abstract. This means that all of the abstract methods of the Shape class must be implemented by the Point class.

The subclass must define an implementation for every abstract method of the abstract superclass, or the subclass itself will also be abstract. Similarly other shape objects can be created using the generic Shape Abstract class.

A big Disadvantage of using abstract classes is not able to use multiple inheritance. In the sense, when a class extends an abstract class, it can't extend any other class.

Interfaces in java

In Java, this multiple inheritance problem is solved with a powerful construct called **interfaces**. Interface can be used to define a generic template and then one or more abstract classes to define partial implementations of the interface. Interfaces just specify the method declaration (implicitly public and abstract) and can only contain fields (which are implicitly public static final). Interface definition begins with a keyword interface. An interface like that of an abstract class cannot be instantiated.

Multiple Inheritance is allowed when extending interfaces i.e. one interface can extend none, one or more interfaces. Java does not support multiple inheritance, but it allows you to extend one class and implement many interfaces.

If a class that implements an interface does not define all the methods of the interface, then it must be declared abstract and the method definitions must be provided by the subclass that extends the abstract class.

Example 1: Below is an example of a Shape interface

```
interface Shape {

        public double area();
        public double volume();
}
```

Below is a Point class that implements the Shape interface.

```
public class Point implements Shape {

        static int x, y;
        public Point() {
                x = 0;
                y = 0;
        }
        public double area() {
                return 0;
        }
        public double volume() {
```

```
                return 0;
        }
        public static void print() {
                System.out.println("point: " + x + "," + y);
        }
        public static void main(String args[]) {
                Point p = new Point();
                p.print();
        }
}
```

Similarly, other shape objects can be created by interface programming by implementing generic Shape Interface

Example 2: Below is a java interfaces program showing the power of interface programming in java

Listing below shows 2 interfaces and 4 classes one being an abstract class. Note: The method *toString* in class *A1* is an overridden version of the method defined in the class named **Object**. The classes *B1* and *C1* satisfy the interface contract. But since the class **D1** does not define all the methods of the implemented interface *I2*, the class D1 is declared abstract.
Also,
i1.methodI2() produces a compilation error as the method is not declared in *I1* or any of its super interfaces if present. Hence a downcast of interface reference I1 solves the problem as shown in the program. The same problem applies to i1.methodA1(), which is again resolved by a downcast.

When we invoke the toString() method which is a method of an Object, there does not seem to be any problem as every interface or class extends Object and any class can override the default toString() to suit your application needs. ((C1)o1).methodI1() compiles successfully, but produces a ClassCastException at runtime. This is because B1 does not have any relationship with C1 except they are "siblings". You can't cast siblings into one another.

When a given interface method is invoked on a given reference, the behavior that results will be appropriate to the class from which that particular object was instantiated. This is runtime polymorphism based on interfaces and overridden methods.

```
interface I1 {

        void methodI1(); // public static by default
}

interface I2 extends I1 {

        void methodI2(); // public static by default
}
```

```
class A1 {

	public String methodA1() {
		String strA1 = "I am in methodC1 of class A1";
		return strA1;
	}
	public String toString() {
		return "toString() method of class A1";
	}
}

class B1 extends A1 implements I2 {

	public void methodI1() {
		System.out.println("I am in methodI1 of class B1");
	}
	public void methodI2() {
		System.out.println("I am in methodI2 of class B1");
	}
}

class C1 implements I2 {

	public void methodI1() {
		System.out.println("I am in methodI1 of class C1");
	}
	public void methodI2() {
		System.out.println("I am in methodI2 of class C1");
	}
}

// Note that the class is declared as abstract as it does not
// satisfy the interface contract
abstract class D1 implements I2 {

	public void methodI1() {
	}
	// This class does not implement methodI2() hence declared
abstract.
}

public class InterFaceEx {

	public static void main(String[] args) {
		I1 i1 = new B1();
		i1.methodI1(); // OK as methodI1 is present in B1
		// i1.methodI2(); Compilation error as methodI2 not
present in I1
		// Casting to convert the type of the reference from type
I1 to type I2
		((I2) i1).methodI2();
		I2 i2 = new B1();
		i2.methodI1(); // OK
		i2.methodI2(); // OK
		// Does not Compile as methodA1() not present in interface
reference I1
```

```
                // String var = i1.methodA1();
                // Hence I1 requires a cast to invoke methodA1
                String var2 = ((A1) i1).methodA1();
                System.out.println("var2 : " + var2);
                String var3 = ((B1) i1).methodA1();
                System.out.println("var3 : " + var3);
                String var4 = i1.toString();
                System.out.println("var4 : " + var4);
                String var5 = i2.toString();
                System.out.println("var5 : " + var5);
                I1 i3 = new C1();
                String var6 = i3.toString();
                System.out.println("var6 : " + var6); // It prints the
Object toString() method
                Object o1 = new B1();
                // o1.methodI1(); does not compile as Object class does
not define
                // methodI1()
                // To solve the probelm we need to downcast o1 reference.
We can do it
                // in the following 4 ways
                ((I1) o1).methodI1(); // 1
                ((I2) o1).methodI1(); // 2
                ((B1) o1).methodI1(); // 3
                /*
                 *
                 * B1 does not have any relationship with C1 except they
are "siblings".
                 *
                 * Well, you can't cast siblings into one another.
                 *
                 */
                // ((C1)o1).methodI1(); Produces a ClassCastException
        }
}
```

Output

```
I am in methodI1  of class B1
I am in methodI2  of class B1
I am in methodI1  of class B1
I am in methodI2  of class B1
var2 : I am in methodC1  of class A1
var3 : I am in methodC1  of class A1
var4 : toString()  method  of class A1
var5 : toString()  method  of class A1
var6 : C1@190d11
I am in methodI1  of class B1
I am in methodI1  of class B1
I am in methodI1  of class B1
```

Polymorphism

Polymorphism means one name, many forms. There are 3 distinct forms of Java Polymorphism;

- Method overloading (Compile time polymorphism)
- Method overriding through inheritance (Run time polymorphism)
- Method overriding through the Java interface (Run time polymorphism)

Polymorphism allows a reference to denote objects of different types at different times during execution. A super type reference exhibits polymorphic behavior, since it can denote objects of its subtypes.

```
interface Shape {

        public double area();
        public double volume();
}

class Cube implements Shape {

        int x = 10;
        public double area( ) {

            return (6 * x * x);
        }

        public double volume() {
                return (x * x * x);
        }

}

class Circle implements Shape {

        int radius = 10;
        public double area() {
                return (Math.PI * radius * radius);
        }
        public double volume() {
                return 0;
        }
}

public class PolymorphismTest {

        public static void main(String args[]) {
                Shape[] s = { new Cube(), new Circle() };
                for (int i = 0; i < s.length; i++) {
                        System.out.println("The area and volume of " +
s[i].getClass()
                                        + " is " + s[i].area() + " , " +
s[i].volume());
```

```
                    }
            }
}
```

Output

The area and volume of class Cube is 600.0 , 1000.0
The area and volume of class Circle is 314.1592653589793 , 0.0

The methods area() and volume() are overridden in the implementing classes. The invocation of the both methods area and volume is determined based on run time polymorphism of the current object as shown in the output.

Interfaces versus Abstract Classes

1. Abstract class is a class which contain one or more abstract methods, which has to be implemented by sub classes. An abstract class can contain no abstract methods also i.e. abstract class may contain concrete methods. A Java Interface can contain only method declarations and public static final constants and doesn't contain their implementation. The classes which implement the Interface must provide the method definition for all the methods present.

2. Abstract class definition begins with the keyword "abstract" keyword followed by Class definition. An Interface definition begins with the keyword "interface".

3. Abstract classes are useful in a situation when some general methods should be implemented and specialization behavior should be implemented by subclasses. Interfaces are useful in a situation when all its properties need to be implemented by subclasses

4. All variables in an Interface are by default – public static final while an abstract class can have instance variables.

5. An interface is also used in situations when a class needs to extend another class apart from the abstract class. In such situations it's not possible to have multiple inheritance of classes. An interface on the other hand can be used when it is required to implement one or more interfaces. Abstract class does not support Multiple Inheritance whereas an Interface supports multiple Inheritance.

6. An Interface can only have public members whereas an abstract class can contain private as well as protected members.

7. A class implementing an interface must implement all of the methods defined in the interface, while a class extending an abstract class need not implement any of the methods defined in the abstract class.

8. The problem with an interface is, if you want to add a new feature (method) in its contract, then you MUST implement those method in all of the classes which implement that interface. However, in the case of an abstract class, the method can be simply implemented in the abstract class and the same can be called by its subclass

9. Interfaces are slow as it requires extra indirection to to find corresponding method in in the actual class. Abstract classes are fast

10. Interfaces are often used to describe the peripheral abilities of a class, and not its central identity, E.g. an Automobile class might implement the Recyclable interface, which could apply to many otherwise totally unrelated objects.

Note: There is no difference between a fully abstract class (all methods declared as abstract and all fields are public static final) and an interface.

Note: If the various objects are all of-a-kind, and share a common state and behavior, then tend towards a common base class. If all they share is a set of method signatures, then tend towards an interface.

Similarities:

Neither Abstract classes nor Interface can be instantiated.