Abstraction in Java

Abstraction

Abstraction is the mechanism of hiding implementation details and displaying only essential functionalities to the user.

Let's consider a real-life example to understand the abstraction.

Consider the following real-life scenario: a man is watching television. The man only knows that by pressing the buttons on the remote, he can change the channel, increase/decrease the volume, increase/decrease the brightness, and so on. Yet he has no idea how, as he presses the buttons on the remote, the volume actually increases; he is unaware of the inner mechanism of the television. This is what abstraction is.

In Java, there are two ways to achieve abstraction:

- Abstract class
- Interface (achieve 100 percent abstraction)

Abstract Class and Abstract Method

- An abstract class is one that has been declared with the abstract keyword.
- An abstract method is one that is declared but does not have a method body. It does not have a complete implementation.
- Both abstract methods may or may not be present in an abstract class. Some of them might be concrete methods.
- Any class that includes one or more abstract methods must also include the abstract keyword in its declaration.
- We cannot use a new keyword to instantiate an abstract class.

When to use abstract class and abstract method?

When we are unsure about the implementation of a method in our program, we declare it abstract. For example, assume we have a class Vehicle with a method **getNoOfWheels()** that returns a set of wheels. But we have no idea how many wheels we have to return since each vehicle has a unique number of wheels. As a result, we abstract this method.

In a program, there are instances where the implementation of the class is incomplete; this sort of partially implemented class is declared as abstract.

Example:

Suppose we have a class **School** with a method **getAdmissionFee()** and subclasses of banks such as DAV, DPS, JNV, etc. Since admission fees vary from school to school, implementing this method in the parent class (**School** class) is meaningless. This is due to the fact that each class must override this method in order to have its own implementation details.

```
// Abstract class
abstract class School {

	// Non-abstract method
	public void show() {
		System.out.println("School Application");
	}

	// Abstract method
	abstract int getAdmissionFee();
}

class DAV extends School {

	// Overriding the method from 'School' class
	int getAdmissionFee() {
		return 25000;
```

```
	}
}

class DPS extends School {

	// Overriding the method from 'School' class
	int getAdmissionFee() {
		return 35000;
	}
}

class JNV extends School {

	// Overriding the method from 'School' class
	int getAdmissionFee() {
		return 1000;
	}
}

public class Main {
	public static void main(String args[]) {

		School s;

		s = new DAV();
		s.show();

		System.out.println("DAV Admission Fee: " + s.getAdmissionFee());

		s = new DPS();
		System.out.println("DPS Admission Fee: " + s.getAdmissionFee());

		s = new JNV();
		System.out.println("JNV Admisssion Fee: " + s.getAdmissionFee());
	}
}
```

Output:

```
School Application
DAV Admission Fee: 25000
DPS Admission Fee: 35000
```

JNV Admisssion Fee: 1000

Interface in Java

An interface is a blueprint of a class. An interface, like a class, may have methods and variables, but the methods declared in an interface are always abstract. Thus, the interface is a way to achieve complete abstraction.

Why we use interface in Java?

- Complete abstraction is achieved by the use of the interface.
- Since Java does not allow multiple inheritance in the case of classes, we can achieve multiple inheritance by using interfaces.
- It can be used for loose coupling.

Why use interfaces when we have abstract classes:

Abstraction is achieved by the use of interfaces and abstract classes. However, the reason for using interfaces is that abstract classes may contain non-final variables, while interface variables are public, final, and static.

Syntax:

```
interface interface_name {
        /*
        Constant Fields and Abstract Methods
         * /
}
```

Here, the **interface** is a keyword used to create an interface.

Example:

```
interface InterfaceExample {

	// public, static and final data member
	int val = 10;

	// public and abstract method
	void show();
}

public class Main implements InterfaceExample {

	// Overriding the abstract method
	public void show() {
		System.out.println("Coding Ninjas");
	}

	public static void main(String args[]) {
		Main t = new Main();
		t.show();
		System.out.println(val);
	}
}
```

Output:

```
Coding Ninjas
10
```

Multiple Inheritance in Java using an Interface

Because of the ambiguity problem, Java does not support multiple inheritance in the case of classes. However, we can use interfaces to implement multiple inheritance in Java since there is no ambiguity in the case of interfaces. It is because of its implementation provided by the implementation class.

Example:

```
interface Interface1 {

	// Default method
	default void show() {
		System.out.println("Default Interface 1");
	}
}

interface Interface2 {

	// Default method
	default void show() {
		System.out.println("Default Interface 2");
	}
}

public class Main implements Interface1, Interface2 {

	// Overriding default method
	public void show() {
		// Using super keyword to call the show() method of interface, 'Interface1'
		Interface1.super.show();

		// Using super keyword to call the show() method of interface, 'Interface2'
		Interface2.super.show();
	}

	public static void main(String args[]) {
		Main obj = new Main();
		obj.show();
	}
```

}

Output:

Default Interface 1
Default Interface 2