# OOP: Abstraction in Java

Abstraction focuses on what an object does instead of how it does it. Abstractionallows us to focus on what an object does instead of how it does. Abstraction is achieved by abstract methods. In java, abstract methods can be created in the abstract classes and interfaces.

1. Abstract class.
2. Interface.

**Abstract class**

* An abstract class is a special class in Java that can have abstract methods.
* We cannot initialize abstract class directly.
* The abstract class can have regular methods and properties as well.
* Abstract class extend another abstract class but implementation of abstract methods are not required.
* The first non-abstract class that extends abstract class must implement all its abstract methods.
* The class can extend only one abstract class by the rules of inherentence to avoid diamond problem.

public abstract class Vehicle{  
 public String name;  
   
 public abstract void move();  
 public abstract void stop();  
}

We have an abstract class Vehicle with two abstract methods move and stop. All vehicles should able to move and stop but the way they do will be different and based on concrete class implementation.

public class Car extends Vehicle {  
   
 @Override  
 public void move() {  
 System.out.println("Car is driving");  
 }  
   
 @Override  
 public void stop() {  
 System.out.println("Car is stopped");  
 }  
}

In above example, Car class is extending the Vehicle abstract class. The Car class is concrete class and it must implement all abstract methods from Vehicle class. It’s kind of agreement for the Car class with abstract class so if Car wants to be a Vehicle. It should have abstract method of Vehicle.

We implement abstract methods by overriding them.

*Overriding rules:*

*1. the method name should be same.*

*2. the number, order, and data type of argument should be same. Data type can be covariant as well.*

*3. access modifier should be same or more accessible.*

*4. if overring method throws exception in the method declaration. Exception should be same or smaller(child exceptions) than parent’s class method.*

Let’s have another concreate class

public class Motorcycle extends Vehicle {  
   
 @Override  
 public void move() {  
 System.out.println("Motorcycle is driving");  
 }  
   
 @Override  
 public void stop() {  
 System.out.println("Motorcycle is stopped");  
 }

}

We have another class that extends our abstract Vehicle class and implements its abstract methods.

Why do we need to create abstract class and then create regular classes that extend it? Why just don’t create two regular classes?

The code should be designed to make changes will be easier and software always changes. And there is one powerful OOP prince — **code to interface, not to concrete classes**. When we say interface we are referring to abstraction. Let’s see an example

public class Main {  
 public static void main(String[] args) {  
 // we cannot assign abstract class directly   
 // Vehicle vehicle = new Vehicle();  
   
 Vehicle vehicle = new Car();  
 vehicle.drive();  
 // a lot of code here ...  
   
 vehicle.stop();  
 }  
}

We cannot initialize abstract class directly so it’s not useful by itself and always needs to be used with concrete classes.

Let’s say we have program that response to drive a vehicle to specific destination. Now business wants to add Motorcycle. If we wouldn’t use abstract class and code them differently with different method names and overall differently. We had to write new program for Motorcycle. Luckily we coded to abstraction and we can just change one line.

public class Main {  
 public static void main(String[] args) {  
 Vehicle vehicle = new Motorcycle();  
 vehicle.drive();  
 // a lot of code here ...  
   
 vehicle.stop();  
 }  
}

**Interface**

* Interface can have only abstract methods except static and default method.
* We cannot initialize interface directly.
* All interface properties are public static final. Basically they are all constants by default.
* public static final is optional to type for properties. It will be injected by default.
* public abstract is optional to type for methods in the interface. It will be injected by default.
* We need to use implements keyword to implement our interface. One class can implement multiple interfaces.
* static and default methods can have bodies in the interface.

Let’s the same example as with abstract class

public interface Vehicle {  
 public static final String str;  
 // private String str1; does not compile

void drive(); // public abstract will be injected by default  
 void stop(); // public abstract will be injected by default

// static method can have a body  
 public static String setStr(String strInput) {  
 str = strInput;  
 }

// default method can have a body  
 public default void printStr() {  
 System.out.println(str);  
 }  
}

Above we have our interface. I have added extra code to show you some details about default public static final properties and two methods that can have bodies.

public class Car implements Vehicle {  
   
 @Override  
 public void move() {  
 System.out.println("Car is driving");  
 }  
   
 @Override  
 public void stop() {  
 System.out.println("Car is stopped");  
 }  
}

We created our first concrete class for our interface. We use implements keyword in order to implement our interface.

public class Motorcycle implements Vehicle {  
   
 @Override  
 public void move() {  
 System.out.println("Motorcycle is driving");  
 }  
   
 @Override  
 public void stop() {  
 System.out.println("Motorcycle is stopped");  
 }

}

And another concreate class.

public class Main {  
 public static void main(String[] args) {  
 // we cannot assign interface directly   
 // Vehicle vehicle = new Vehicle();  
   
 Vehicle vehicle = new Car();  
 vehicle.drive();  
 // a lot of code here ...  
   
 vehicle.stop();  
 }  
}

We cannot initialize interfaces directly so they are not useful at all without concrete classes.