# Encapsulation in Java:

**Encapsulation in Java** is a process of wrapping code and data together into a single unit, for example, a capsule which is mixed of several medicines.



We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

### **Advantage of Encapsulation in Java**

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

### **Another Example of Encapsulation in Java**

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

*File: Account.java*

1. //A Account class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. **class** Account {
4. //private data members
5. **private** **long** acc\_no;
6. **private** String name,email;
7. **private** **float** amount;
8. //public getter and setter methods
9. **public** **long** getAcc\_no() {
10. **return** acc\_no;
11. }
12. **public** **void** setAcc\_no(**long** acc\_no) {
13. **this**.acc\_no = acc\_no;
14. }
15. **public** String getName() {
16. **return** name;
17. }
18. **public** **void** setName(String name) {
19. **this**.name = name;
20. }
21. **public** String getEmail() {
22. **return** email;
23. }
24. **public** **void** setEmail(String email) {
25. **this**.email = email;
26. }
27. **public** **float** getAmount() {
28. **return** amount;
29. }
30. **public** **void** setAmount(**float** amount) {
31. **this**.amount = amount;
32. }
34. }

*File: TestAccount.java*

1. //A Java class to test the encapsulated class Account.
2. **public** **class** TestEncapsulation {
3. **public** **static** **void** main(String[] args) {
4. //creating instance of Account class
5. Account acc=**new** Account();
6. //setting values through setter methods
7. acc.setAcc\_no(7560504000L);
8. acc.setName("Sonoo Jaiswal");
9. acc.setEmail("sonoojaiswal@javatpoint.com");
10. acc.setAmount(500000f);
11. //getting values through getter methods
12. System.out.println(acc.getAcc\_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.getAmount());
13. }
14. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=TestEncapsulation" \t "_blank)**

Output:

7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0

Abstract class in Java:

A class which is declared with the abstract keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

### **Abstraction in Java**

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

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you 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](https://www.javatpoint.com/object-and-class-in-java) 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 which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### **Points to Remember**

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.



**Example of abstract class**

1. **abstract** **class** A{}

### **Abstract Method in Java**

A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

1. **abstract** **void** printStatus();//no method body and abstract

### **Example of Abstract class that has an abstract method**

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

1. **abstract** **class** Bike{
2. **abstract** **void** run();
3. }
4. **class** Honda4 **extends** Bike{
5. **void** run(){System.out.println("running safely");}
6. **public** **static** **void** main(String args[]){
7. Bike obj = **new** Honda4();
8. obj.run();
9. }
10. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=Honda4)

running safely

### **Understanding the real scenario of Abstract class**

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

*File: TestAbstraction1.java*

1. **abstract** **class** Shape{
2. **abstract** **void** draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. **class** Rectangle **extends** Shape{
6. **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** Circle1 **extends** Shape{
9. **void** draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. **class** TestAbstraction1{
13. **public** **static** **void** main(String args[]){
14. Shape s=**new** Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
15. s.draw();
16. }
17. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction1)

drawing circle

1. **abstract** **class** Bank{
2. **abstract** **int** getRateOfInterest();
3. }
4. **class** SBI **extends** Bank{
5. **int** getRateOfInterest(){**return** 7;}
6. }
7. **class** PNB **extends** Bank{
8. **int** getRateOfInterest(){**return** 8;}
9. }
11. **class** TestBank{
12. **public** **static** **void** main(String args[]){
13. Bank b;
14. b=**new** SBI();
15. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
16. b=**new** PNB();
17. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
18. }}

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

Rate of Interest is: 7 %

Rate of Interest is: 8 %

### **Abstract class having constructor, data member and methods**

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

*File: TestAbstraction2.java*

1. //Example of an abstract class that has abstract and non-abstract methods
2. **abstract** **class** Bike{
3. Bike(){System.out.println("bike is created");}
4. **abstract** **void** run();
5. **void** changeGear(){System.out.println("gear changed");}
6. }
7. //Creating a Child class which inherits Abstract class
8. **class** Honda **extends** Bike{
9. **void** run(){System.out.println("running safely..");}
10. }
11. //Creating a Test class which calls abstract and non-abstract methods
12. **class** TestAbstraction2{
13. **public** **static** **void** main(String args[]){
14. Bike obj = **new** Honda();
15. obj.run();
16. obj.changeGear();
17. }
18. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestAbstraction2)

bike is created

running safely..

gear changed

# Interface in Java

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*[*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

## **Why use Java interface?**

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

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



## **How to declare an interface?**

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

#### **The 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**.

## **Java Interface Example**

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

1. **interface** printable{
2. **void** print();
3. **int id;**
4. }
5. **class** A6 **implements** printable{
6. **public** **void** print(){System.out.println("Hello");}
8. **public** **static** **void** main(String args[]){
9. A6 obj = **new** A6();
10. obj.print();
11. }
12. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=A6)

Output:

Hello

## **Java Interface Example: Drawable**

In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

*File: TestInterface1.java*

1. //Interface declaration: by first user
2. **interface** Drawable{
3. **void** draw();
4. }
5. //Implementation: by second user
6. **class** Rectangle **implements** Drawable{
7. **public** **void** draw(){System.out.println("drawing rectangle");}
8. }
9. **class** Circle **implements** Drawable{
10. **public** **void** draw(){System.out.println("drawing circle");}
11. }
12. //Using interface: by third user
13. **class** TestInterface1{
14. **public** **static** **void** main(String args[]){
15. Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()
16. d.draw();
17. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface1)

drawing circle

## **Java Interface Example: Bank**

Let's see another example of java interface which provides the implementation of Bank interface.

*File: TestInterface2.java*

1. **interface** Bank{
2. **float** rateOfInterest();
3. }
4. **class** SBI **implements** Bank{
5. **public** **float** rateOfInterest(){**return** 9.15f;}
6. }
7. **class** PNB **implements** Bank{
8. **public** **float** rateOfInterest(){**return** 9.7f;}
9. }
10. **class** TestInterface2{
11. **public** **static** **void** main(String[] args){
12. Bank b=**new** SBI();
13. System.out.println("ROI: "+b.rateOfInterest());
14. }}

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestInterface2)

Output:

ROI: 9.15

# 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. Since Java 8, it can have **default and static methods** also. |
| 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 provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**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%).

#### **Notes on Interfaces:**

* Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
* Interface methods do not have a body - the body is provided by the "implement" class
* On implementation of an interface, you must override all of its methods
* Interface methods are by default abstract and public
* Interface attributes are by default public, static and final
* An interface cannot contain a constructor (as it cannot be used to create objects)

#### **Why And When To Use Interfaces?**

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

## **Multiple Interfaces**

To implement multiple interfaces, separate them with a comma:

### **Example**

interface FirstInterface {

public void myMethod(); // interface method

}

interface SecondInterface {

public void myMethod(); // interface method

}

class DemoClass implements FirstInterface, SecondInterface {

public void myMethod() {

System.out.println("Some text..");

}

}

class Main {

public static void main(String[] args) {

DemoClass myObj = new DemoClass();

myObj.myMethod();

}

}