# Interfaces in Java

 An interface in Java is a blueprint of a class.

A Java interface contains static constants and abstract methods.

The interface in Java is *a*mechanism to achieve [abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2/). There can be only abstract methods in the Java interface, not the method body.

It is used to achieve abstraction and [multiple inheritance in Java](https://www.geeksforgeeks.org/java-and-multiple-inheritance/). 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**

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

* Interfaces specify what a class must do and not how. It is the blueprint of the class.
* An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
* If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
* A Java library example is [Comparator Interface](https://www.geeksforgeeks.org/comparator-interface-java/). If a class implements this interface, then it can be used to sort a collection.

**Syntax:**

Interface {

// declare constant fields

// declare methods that abstract

// by default.

}

To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use implements keyword.

### Why do we use an Interface?

* It is used to achieve total abstraction.
* Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
* It is also used to achieve loose coupling.
* Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.

// A simple interface

interface Player

{

final int id = 10;

int move();

}

### Difference Between Class and Interface

The major differences between a class and an interface are:

| S. No. | Class | Interface |
| --- | --- | --- |
| 1. | In class, you can instantiate variables and create an object. | In an interface, you can’t instantiate variables and create an object. |
| 2. | Class can contain concrete(with implementation) methods | The interface cannot contain concrete(with implementation) methods |
| 3. | The access specifiers used with classes are private, protected, and public. | In Interface only one specifier is used- Public. |

**Implementation:**To implement an interface we use the keyword **implements**

#### Advantages of Interfaces in Java

The advantages of using interfaces in Java are as follows:

1. Without bothering about the implementation part, we can achieve the security of the implementation.
2. In Java, multiple inheritance is not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

### New Features Added in Interfaces in JDK 8

**1.**Prior to JDK 8, the interface could not define the implementation. We can now add default implementation for interface methods. This default implementation has a special use and does not affect the intention behind interfaces.

Suppose we need to add a new function in an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Lambda expressions:

Lambda expressions basically express instances of functional interfaces (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda expressions implement the only abstract function and therefore implement functional interfaces

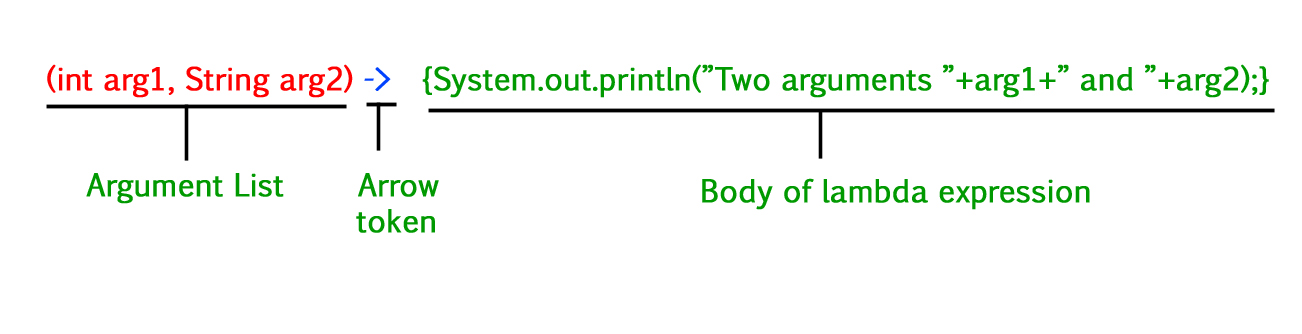
lambda expressions are added in Java 8 and provide below functionalities.

* Enable to treat functionality as a method argument, or code as data.
* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.

|  |
| --- |
| // Java program to demonstrate lambda expressions  // to implement a user defined functional interface.    // A sample functional interface (An interface with  // single abstract method  **interface** FuncInterface  {      // An abstract function  **void** abstractFun(**int** x);        // A non-abstract (or default) function  **default** **void** normalFun()      {         System.out.println("Hello");      }  }    **class** Test  {  **public** **static** **void** main(String args[])      {          // lambda expression to implement above          // functional interface. This interface          // by default implements abstractFun()          FuncInterface fobj = (x)->System.out.println(2\*x);            // This calls above lambda expression and prints 10.          fobj.abstractFun(5);      }  } |

Output:

10

[](https://media.geeksforgeeks.org/wp-content/uploads/lambda-expression.jpg)  
**Syntax:**

lambda operator -> body

where lambda operator can be:

* **Zero parameter:**

() -> System.out.println("Zero parameter lambda");

* **One parameter:**–

(p) -> System.out.println("One parameter: " + p);

It is not mandatory to use parentheses, if the type of that variable can be inferred from the context

* **Multiple parameters :**

(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

Please note: Lambda expressions are just like functions and they accept parameters just like functions.

|  |
| --- |
| // A Java program to demonstrate simple lambda expressions  **import** java.util.ArrayList;  **class** Test  {  **public** **static** **void** main(String args[])      {          // Creating an ArrayList with elements          // {1, 2, 3, 4}          ArrayList<Integer> arrL = **new** ArrayList<Integer>();          arrL.add(1);          arrL.add(2);          arrL.add(3);          arrL.add(4);    for(){  syso(arr[i])  }          // Using lambda expression to print all elements          // of arrL          arrL.forEach(n -> System.out.println(n));            // Using lambda expression to print even elements          // of arrL          arrL.forEach(n -> { **if** (n%2 == 0) System.out.println(n); });      }  } |

Output :

1

2

3

4

2

4

Note that lambda expressions can only be used to implement functional interfaces. In the above example also, the lambda expression implements [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) Functional Interface.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **Final Access Modifier**

[Final access modifier](https://www.geeksforgeeks.org/final-keyword-in-java/) is a modifier applicable to classes, methods, and variables. If we declare a parent class method as final then we can’t override that method in the child class because its implementation is final and if a class is declared as final we can’t extend the functionality of that class i.e we can’t create a child class for that class i.e inheritance is not possible for final classes. Every method present inside the final class is always final y default but every variable present inside the final class need not be final. The main advantage of the final keyword is we can achieve security and we can provide a unique implementation. But the main disadvantage of the final keyword is we are missing key benefits of OOPs like Inheritance(Because of the final class), Polymorphism(Because of the final method)  hence if there are no specific requirements then it is not recommended to use the final keyword.

**Example 1:**

* Java

|  |
| --- |
| // Java Program to illustrate Final keyword  // Where No final keyword Is Used    // Importing required classes  **import** java.io.\*;  **import** java.util.\*;    // Class 1  // Super-class  **class** P {        // Method 1      // To declare first name  **public** **void** firstName()      {            // Passing name and print it          System.out.println("Mayank");      }        // Method 2      // To declare last name  **public** **void** surName()      {            // Passing name and print it          System.out.println("Trivedi");      }  }    // Class 2  // Sub-class  // Extending above class  **class** C **extends** P {        // Method 1      // Trying to override the last name  **public** **void** surName()      {          // Display surname          System.out.println("Sharma");      }        // Method 2      // Main driver method  **public** **static** **void** main(String[] args)      {            // Display message          System.out.println("Exmpl ");      }  } |

**Output**

GFG

### **Static Access Modifier**

[Static access modifier](https://www.geeksforgeeks.org/static-keyword-java/) is an access modifier that is applicable for methods and variables but not for classes. We cant declare top-level class with a static modifier but we can declare the inner class as static (such types of inner classes are known as static nested classes). In the case of instance variable for every object, a separate copy will be created but in the case of static variable, a single copy will be created at class level and shared by every object of that class.

**Example**

* Java

|  |
| --- |
| // Java Program to Illustrate Static Access Modifier    // Importing required classes  **import** java.io.\*;  **import** java.util.\*;    // Main class  **class** Exmpl{        // Creating a static variable and      // initializing a custom value  **static** **int** x = 10;        // Creating a instance variable and      // initializing a custom value  **int** y = 20;        // Main driver method  **public** **static** **void** main(String[] args)      {            // Creating an object of class inside main() method          Exmpl t1 = **new** Exmpl ();            // Accessing and re-initializing the          // static and instance variable          // using t1 reference          t1.x = 88;          t1.y = 99;            // Creating an object of class inside main() method          // again          Exmpl t2 = **new** Exmpl ();            // Accessing the static and instance variable using          // t2 reference as we know that for each object          // there is a separate copy of instance variable          // created. While a same copy of static variable will          // be shared between the objects            // Displaying the value of static and instance          // variable using t2 object reference          System.out.println(              "Value of Static variable x = " + t2.x + "\n"              + "Value of Instance variable y = " + t2.y);      }  } |

**Output**

Value of Static variable x = 88

Value of Instance variable y = 20

| **Final Access Modifier** | **Static Access Modifier** |
| --- | --- |
| This modifier is applicable to both outer and inner classes, variables, methods, and blocks. | This modifier is only applicable to inner classes, methods, and variables. |
| It is not necessary to initialize the final variable at the time of its declaration. | It is necessary to initialize the static variable at the time of its declaration. |
| Final variable cannot be reinitialized. | Static variables can be reinitialized. |
| Final method can’t be inherited. | Static methods can only access the static members of the class and can only be called by other static methods. |
| Final class can’t be inherited by any class. | The static class object can’t be created and it only contains static members only. |
| Final keyword doesn’t support any block for initialization of final variables. | Static block is used to initialize the static variables. |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*