Interfaces in Java

Like a class, an interface can have methods and variables, but the methods declared in 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 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 <interface\_name> {

// declare constant fields

// declare methods that abstract

// by default.

}

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

**Why do we use interface ?**

* It is used to achieve total abstraction.
* Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
* 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 interface are final, public and static.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // A simple interface  interface Player  {      final int id = 10;      int move();  } |

To implement an interface we use keyword: implement

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate working of  // interface.  import java.io.\*;    // A simple interface  interface in1  {      // public, static and final      final int a = 10;        // public and abstract      void display();  }    // A class that implements interface.  class testClass implements in1  {      // Implementing the capabilities of      // interface.      public void display()      {          System.out.println("Geek");      }        // Driver Code      public static void main (String[] args)      {          testClass t = new testClass();          t.display();          System.out.println(a);      }  } |

Output:

Geek

10

**A real world example:**  
Let’s consider the example of vehicles like bicycle, car, bike………,they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicylce, Bike, car ….etc implement all these functionalities in their own class in their own way.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| import java.io.\*;    interface Vehicle {        // all are the abstract methods.      void changeGear(int a);      void speedUp(int a);      void applyBrakes(int a);  }    class Bicycle implements Vehicle{        int speed;      int gear;         // to change gear      @Override      public void changeGear(int newGear){            gear = newGear;      }        // to increase speed      @Override      public void speedUp(int increment){            speed = speed + increment;      }        // to decrease speed      @Override      public void applyBrakes(int decrement){            speed = speed - decrement;      }        public void printStates() {           System.out.println("speed: " + speed                + " gear: " + gear);      }  }    class Bike implements Vehicle {        int speed;      int gear;        // to change gear      @Override      public void changeGear(int newGear){            gear = newGear;      }        // to increase speed      @Override      public void speedUp(int increment){            speed = speed + increment;      }        // to decrease speed      @Override      public void applyBrakes(int decrement){            speed = speed - decrement;      }        public void printStates() {           System.out.println("speed: " + speed               + " gear: " + gear);      }    }  class GFG {        public static void main (String[] args) {            // creating an inatance of Bicycle          // doing some operations          Bicycle bicycle = new Bicycle();          bicycle.changeGear(2);          bicycle.speedUp(3);          bicycle.applyBrakes(1);            System.out.println("Bicycle present state :");          bicycle.printStates();            // creating instance of bike.          Bike bike = new Bike();          bike.changeGear(1);          bike.speedUp(4);          bike.applyBrakes(3);            System.out.println("Bike present state :");          bike.printStates();      }  } |

Output;

Bicycle present state :

speed: 2 gear: 2

Bike present state :

speed: 1 gear: 1

**New features added in interfaces in JDK 8**

1. Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has 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.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // An example to show that interfaces can  // have methods from JDK 1.8 onwards  interface in1  {      final int a = 10;      default void display()      {          System.out.println("hello");      }  }    // A class that implements interface.  class testClass implements in1  {      // Driver Code      public static void main (String[] args)      {          testClass t = new testClass();          t.display();      }  } |

Output :

hello

1. Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // An example to show that interfaces can  // have methods from JDK 1.8 onwards  interface in1  {      final int a = 10;      static void display()      {          System.out.println("hello");      }  }    // A class that implements interface.  class testClass implements in1  {      // Driver Code      public static void main (String[] args)      {          in1.display();      }  } |

Output :

hello

**Important points about interface or summary of article:**

* We can’t create instance(interface can’t be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
* A class can implement more than one interface.
* An interface can extends another interface or interfaces (more than one interface) .
* A class that implements interface must implements all the methods in interface.
* All the methods are public and abstract. And all the fields are public, static, and final.
* It is used to achieve multiple inheritance.
* It is used to achieve loose coupling.

**New features added in interfaces in JDK 9**  
From Java 9 onwards, interfaces can contain following also

1. Static methods
2. Private methods
3. Private Static methods

Functional Interfaces In Java

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, [lambda expressions](https://www.geeksforgeeks.org/lambda-expressions-java-8/) can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. ***Runnable***, ***ActionListener***,***Comparable***are some of the examples of functional interfaces.  
Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate functional interface    class Test  {      public static void main(String args[])      {          // create anonymous inner class object          new Thread(new Runnable()          {              @Override              public void run()              {                  System.out.println("New thread created");              }          }).start();      }  } |

Output:

New thread created

Java 8 onwards, we can assign [lambda expression](https://www.geeksforgeeks.org/lambda-expressions-java-8/) to its functional interface object like this:

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate Implementation of  // functional interface using lambda expressions    class Test  {    public static void main(String args[])    {        // lambda expression to create the object      new Thread(()->         {System.out.println("New thread created");}).start();    }  } |

New thread created

**@FunctionalInterface Annotation**  
@FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method. In case more than one abstract methods are present, the compiler flags an ‘Unexpected @FunctionalInterface annotation’ message. However, it is not mandatory to use this annotation.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java program to demonstrate lamda expressions to implement  // a user defined functional interface.    @FunctionalInterface  interface Square  {      int calculate(int x);  }    class Test  {      public static void main(String args[])      {          int a = 5;            // lambda expression to define the calculate method          Square s = (int x)->x\*x;            // parameter passed and return type must be          // same as defined in the prototype          int ans = s.calculate(a);          System.out.println(ans);      }  } |

Output:

25

**java.util.function Package:**  
The java.util.function package in Java 8 contains many builtin functional interfaces like-

* **Predicate:**The Predicate interface has an abstract method test which gives a Boolean value as a result for the specified argument. Its prototype is
* public Predicate
* {
* public boolean test(T t);

}

* **BinaryOperator:** The BinaryOperator interface has an abstract method apply which takes two argument and returns a result of same type. Its prototype is
* public interface BinaryOperator
* {
* public T apply(T x, T y);

}

* **Function:** The Function interface has an abstract method apply which takes argument of type T and returns a result of type R. Its prototype is
* public interface Function
* {
* public R apply(T t);

}

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // A simple program to demonstrate the use  // of predicate interface  import java.util.\*;  import java.util.function.Predicate;    class Test  {      public static void main(String args[])      {            // create a list of strings          List<String> names =              Arrays.asList("Geek","GeeksQuiz","g1","QA","Geek2");            // declare the predicate type as string and use          // lambda expression to create object          Predicate<String> p = (s)->s.startsWith("G");            // Iterate through the list          for (String st:names)          {              // call the test method              if (p.test(st))                  System.out.println(st);          }      }  } |

Output:

Geek

GeeksQuiz

Geek2

**Important Points/Observations:**

1. A functional interface has only one abstract method but it can have multiple default methods.
2. @FunctionalInterface annotation is used to ensure an interface can’t have more than one abstract method. The use of this annotation is optional.
3. The java.util.function package contains many builtin functional interfaces in Java 8.