Interfaces in Java
Lambda Expressions in Java

Interfaces

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.

It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods, default methods, static methods and variables.

Note: Default and static methods were introduced from JDK 8, prior to that only public abstract methods and public, static, final variables were used in the interfaces

Why use Java interface?

There are mainly two reasons to use interface.

They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

How to declare an interface?

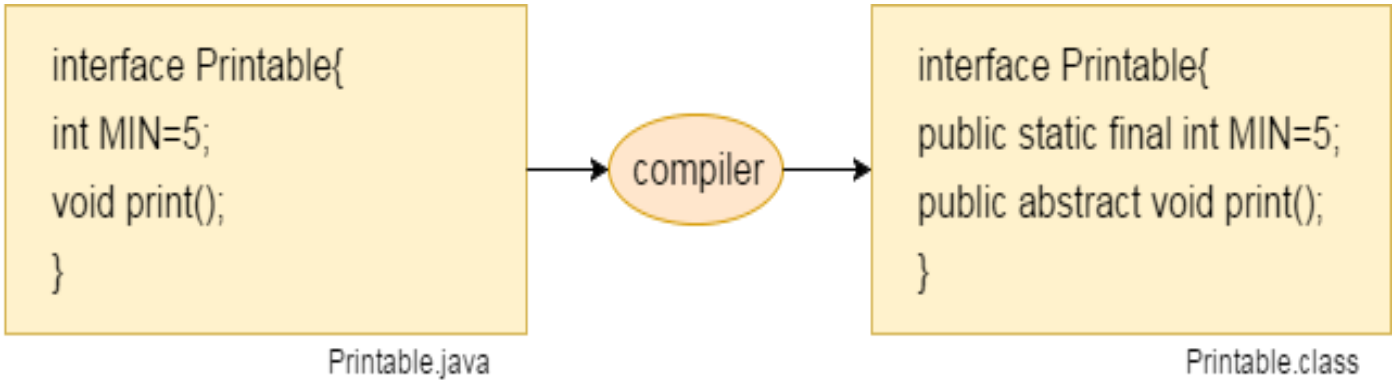
An interface is declared by using the interface keyword. It provides abstraction; means most of the methods in an interface are declared with the empty body[except default and static methods], and all the fields are public, static and final by default. A class that implements an interface must implement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

Syntax :

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // default/ or static methods
    // by default.
}
```

The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.

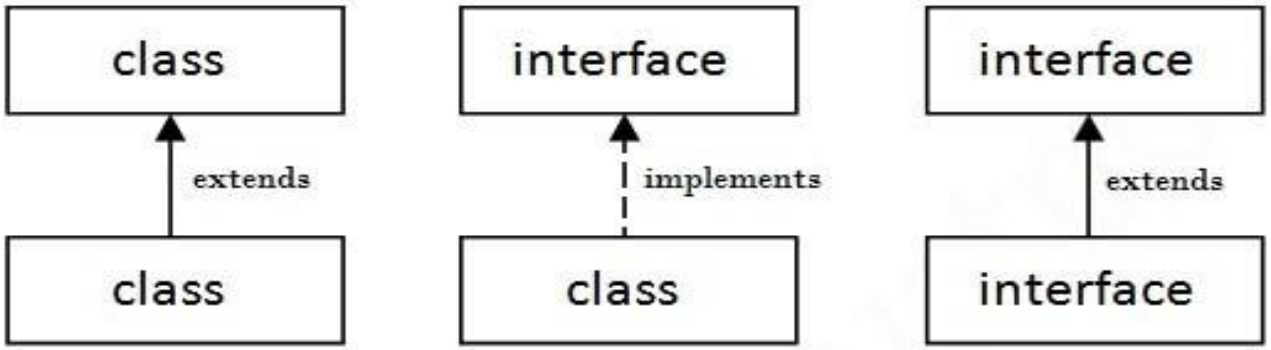


Example of an interface

```
interface My
    {
        int x = 10;                 // by default public static and final
        void demo();                // by default public and abstract
        default void show()         {…}// default method, public
        static void test()          {…}  // static method, public
    }
```

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.



Types of methods in interfaces

We can have three types of methods in an interface:

Abstract methods: Method without body and qualified with abstract keyword

static methods: Method with body and qualified with static keyword

default method: Method with body and qualified with default keyword

But we cannot have concrete methods in interfaces like abstract classes.

Basic Example of interface[Interfaces can also help to show the polymorphic behavior]

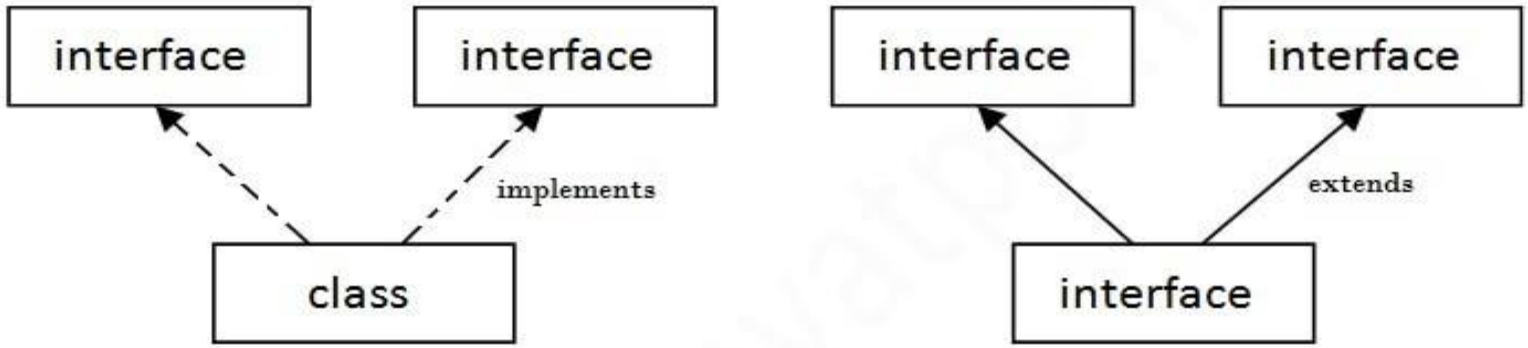

```
interface Shape
{
   int l=12,b=34;
   void area();
   default void msg()
   {
  System.out.println("Welcome to interfaces");
   }
}
class Rectangle implements Shape
{

 public void area()
   {
      System.out.println("Area of rec is "+ l*b);
   }
}
class Circle implements Shape
{
   float r;
   Circle(float r)
   {
      this.r=r;
   }
   public void area()
   {
      System.out.println("Area of rec is "+ 3.14*r*r);
   }
}
public class Main
{
        public static void main(String[] args)
        {
                Shape r = new Rectangle();
                r.msg();//calling default method
                r.area();
                r= new Circle(2.34f);
                r.area();
        }
}
```

Explanation

- In the last example the dynamic method call resolution is done
- When we call method through reference variable of interface type, the correct version will be called based on type of object(or instance) where the interface reference currently referring to.

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example


```
interface A{
void print();
}
interface B{
void print();
}
class C implements A, B
{
public void print()
{
System.out.println("Hello");
}
public static void main(String args[])
{
C  obj = new C();
obj.print();
 }
}
```

Output:

Hello

Multiple inheritance when default method with same signatures is available in two interfaces

```
interface A
{
   default void show()
   {
      System.out.println("A");
   }
}
interface B
{
   default void show()
   {
      System.out.println("B");
   }
}
```

```
class C implements A,B
{
    public void show()
   {
      System.out.println("New definition of show");
   }
}
public class Main
{
            public static void main(String[] args) {
            A obj1=new C();
            obj1.show();
            B obj2=new C();
            obj2.show();
            }
}
```

Output:

```
New definition of show
New definition of show
```

Key point from the last program

If class C is not overriding the show() method, then error would arise, so it becomes mandatory for the class to override the method which implements two interfaces which further contain default methods of same signatures.

Abstract class vs Interface

- **Type of methods:** Interface can have only abstract methods. Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static methods also.
- **Final Variables:** Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.
- **Type of variables:** Abstract class can have final, non-final, static and non-static variables. Interface has only static and final variables.
- **Implementation:** Abstract class can provide the implementation of interface. Interface can't provide the implementation of abstract class.
- **Inheritance vs Abstraction:** A Java interface can be implemented using keyword "implements" and abstract class can be extended using keyword "extends".
- **Multiple implementation:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}
}
```

Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Functional Interfaces In Java

- A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit.
- Functional interface can have any number of default methods
- @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.

Also known as *Single Abstract Method (SAM)* interfaces.

```
@FunctionalInterface
        interface MyInterface
        {
                int test(int n);
        }
```

We will study this topic in main detail when we cover the lambda expression

Important Points in abstract class and interfaces

- Abstract classes are like regular classes with data and methods, but you cannot create instances of abstract classes using the new operator.
- An abstract method cannot be contained in a non abstract class. If a subclass of an abstract superclass does not implement all the inherited abstract methods of the superclass, the subclass must be defined as abstract.
- A class that contains abstract methods must be abstract. However, it is possible to define an abstract class that doesn't contain any abstract methods.
- A subclass can be abstract even if its superclass is concrete.

- An interface is a class-like construct that may contain constants ,abstract methods, static and default methods. In many ways, an interface is similar to an abstract class, but an abstract class can contain constants and abstract methods as well as non-final variables and concrete methods.
- An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file(.class), just like a regular class.
- A class can extend only one superclass but can implement one or more interfaces.
- An interface can extend one or more interfaces.

Extends vs Implements

S.No.	Extends	Implements
1.	By using "extends" keyword a class can inherit another class, or an interface can inherit other interfaces	By using "implements" keyword a class can implement an interface
2.	It is not compulsory that subclass that extends a superclass override all the methods in a superclass.	It is compulsory that class implementing an interface has to implement all the methods of that interface.
3.	Only one superclass can be extended by a class.	A class can implement any number of an interface at a time
4.	Any number of interfaces can be extended by interface.	An interface can never implement any other interface

Lambda Expression

- A lambda expression is, essentially, an anonymous (unnamed) method which is used to implement a method defined by a functional interface.
- Lambda expressions are also commonly referred to as **closures.**
- Lambda expressions are implementation of only abstract method of functional interface that is being implemented or instantiated anonymously.

- Lambda Expression introduces a new operator (−>) which is referred to as the **lambda operator** or the **arrow operator**.
- It divides a lambda expression into two parts.
- The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.
- On the right side is the lambda body, which specifies the actions of the lambda expression.

- The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.
- A lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type.
- The type of the abstract method and the type of the lambda expression must be compatible.
- Java lambda expression is treated as a function, so compiler does not create .class file.

Body of a lambda expression

- Body can be a single expression or a statement block.
- When the body consist of a single expression, it is called as Expression Lambda or Expression Body.
- When the code on the right side of the lambda operator consists of a block of code that can contain more than one statement, It is called as block body or Block Lambda.
- Aside from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed. One key difference, however, is that you must explicitly use a **return** statement to return a value. This is necessary because a block lambda body does not represent a single expression.

Example

```
int sum(int a, int b)
    {
        return (a+b);
    }
```

Using Lambda:

```
(int a, int b)  ->  return(a+b);
```

OR

```
(a, b) -> return(a+b);
```

Structure of a lambda expression

Argument List:

- A lambda expression can contain zero or more arguments.

```
// No argument
        ( ) -> {System.out.println("No argument");}

// Single argument
        (int arg) ->  {System.out.println(“One argument : " + arg);}

// More than one arguments
        ( int arg1, String arg2 ) -> {System.out.println(“Multiple
    Arguments:”);}
```

Argument List:

- We can eliminate the argument type while passing it to lambda expressions, those are inferred types. i.e. (int a) and (a) both are same.

  But we can not use inferred and declared types together

  ```
  ( int arg1, arg2 ) -> { … }          // This is invalid
  ```

- We can also eliminate “()” if there is only one argument.

- More than one arguments are separated by comma (,) operator.

Characteristics of Lambda Expressions

- Optional type declaration
- Optional parenthesis around parameter
- Optional curly braces
- Optional return keyword

NOTE: We can pass a lambda expression wherever a functional interface is expected.

Example 1 of lambda Expression

```
interface A
{
   void cube(int x);
}

public class Main
{
        public static void main(String[] args)
        {
                A ob = (int x) -> System.out.println("Cube is "+ (x*x*x));
                ob.cube(5);
        }
}
```

Example 2 of lambda Expression

```
interface A
{
   void cube(int x);
}

public class Main
{
        public static void main(String[] args)
        {
                A ob = (x) -> System.out.println("Cube is "+ (x*x*x));
                ob.cube(5);
        }
}
```

Example 3 of lambda Expression

```
interface A
{
   void cube(int x,int y);
}

public class Main
{
        public static void main(String[] args)
        {
                A ob = (x,y) -> System.out.println("Sum is "+ (x+y));
                ob.cube(5,6);
        }
}
```

Example 4 of lambda Expression

Lambda Expression can be used to implement multiple operations

```
interface A
{
   void calculate(int x,int y); // by default public and abstract
}

public class Main
{
        public static void main(String[] args)
        {
                A add = (x,y) -> System.out.println("Sum is "+ (x+y));
                A sub = (x,y) -> System.out.println("Sum is "+ (x-y));
                add.calculate(12,34);
                sub.calculate(12,34);
        }
}
```

Example 5 of lambda expression


```
// Demonstrate a lambda expression that takes a parameter.
interface NumericTest {
boolean test(int n);
}
public class Main {
public static void main(String args[])
{
// A lambda expression that tests if a number is even.
int num=12;
//NumericTest isEven = (int n) -> (n % 2)==0;//We can specify the type in left side
NumericTest isEven = (n) -> (n % 2)==0;//We can skip the type in left side
//NumericTest isEven =  n -> (n % 2)==0;//When there is one argument, we can remove brackets also
if(isEven.test(num))
System.out.println("num is even");
else
System.out.println("num is odd");
NumericTest isnum=(n)->n>=0;
if(isnum.test(num))
System.out.println("Number is non-negative");
else
System.out.println("Number is negative");
}
}
```

Example 6(Block lambda)


```
// A block lambda that computes the factorial of an int value.
interface NumericFunc {
int func(int n);
}
public class Main {
public static void main(String args[])
{
// This block lambda computes the factorial of an int value.
NumericFunc factorial = (n) -> {
int result = 1;
for(int i=1; i <= n; i++)
result = i * result;
return result;
};
System.out.println("The factoral of 3 is " + factorial.func(3));
System.out.println("The factoral of 5 is " + factorial.func(5));
}
}
```

Output:

The factorial of 3 is 6

The factorial of 5 is 120

Example 7(Using lambda expression to create thread)


```
public class Main
{
        public static void main(String[] args) {
Thread ref = new Thread(() -> {
// this logic is implementation of run() method to print only even numbers
for (int i = 0; i < 20; i++)
{
if (i % 2 == 0)
 {
System.out.println("Even Number Thread : "+i);
try
{
Thread.sleep(1000);
}
catch (InterruptedException e)
 {
e.printStackTrace();
 }
 }
 }
});
// starting the thread
ref.start();
// Printing the odd numbers from main thread.
for (int i = 0; i < 20; i++)
{
if (i % 2 == 1)
 {
System.out.println("Odd Number Thread : "+i);
try
{
        Thread.sleep(1000);

}
catch (InterruptedException e)
 {
        e.printStackTrace();
 }
 }
 }
 }
 }
```

ArrayList forEach() method in Java

The **forEach()** method of **ArrayList** used to perform the certain operation for each element in ArrayList. This method traverses each element of the Iterable of ArrayList until all elements have been Processed by the method or an exception is raised. The operation is performed in the order of iteration if that order is specified by the method. Exceptions thrown by the Operation are passed to the caller.

The forEach() method can also be used with lambda Expression.

Example

```
import java.util.*;
public class Main
{
        public static void main(String[] args)
        {
         ArrayList<String> list=new ArrayList<String>();
    list.add(“ABC");
    list.add(“PQR");
    list.add(“STU");
    list.add(“XYZ");
    list.forEach(
       (n)->System.out.println(n)
    );
        }
}
```

Advantages of lambda expressions

- 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.
- Fewer lines of code will be needed
- Sequential and Parallel execution support by passing behavior as an argument in methods

Q1

Variables in interface are by default

A. Non-static
B. static
C. final
D. Both static and final

Q2

Methods in interface are by default

A. Protected
B. Public
C. Private
D. Non-static

Q3(Output??)

CODERINDEED

```
interface A {
void show();
}
public class Main implements A {
void show()
{
   System.out.println("Hello");
}
public static void main(String args[])
{
A ref=new Main();
ref.show();
}
}
```

A. Hello
B. Blank output
C. Compile time error
D. Runtime error

Q4(Output??)


```
interface A {
void show();
}
interface B {
void show();
}
public class Main implements A,B {
public void show()
{
   System.out.println("Hello");
}
public static void main(String args[])
{
Main ref=new Main();
ref.show();
}
}
```

A. Hello
B. Blank output
C. Compile time error
D. Runtime error

Q5(Output??)

CODERINDEED

```
interface A {
void show();
void display(){ System.out.println("Welcome");}
}
public class Main implements A {
public void show()
{
   System.out.println("Hello");
}
public static void main(String args[])
{
A ref=new Main();
ref.display();
}
}
```

A. Compile time error
B. Hello
C. Welcome
D. Runtime error

Q6(Output??)

CODERINDEED

```
interface A {
protected int x=12;
public void show();
}
public class Main implements A {
public void show()
{
   System.out.println(x);
}
public static void main(String args[])
{
A ref=new Main();
ref.show();
}
}
```

A. 12
B. 0
C. Compile time error
D. Runtime error

Q7(Output??)

CODERINDEED

```
interface A {
int x=12;
}
interface B {
int y=13;
}
interface C extends A,B
{
   static int sum(){ return x+y;}
}
public class Main implements C {
public static void main(String args[])
{
System.out.println(C.sum());
}
}
```

A. 25
B. Compile time error
C. Runtime error
D. Blank output

Q8(Output??)

CODERINDEED

```
interface A
{
   int x=10;
}
class B
{
   static int y=13;
}
public class Main extends B implements A {
public static void main(String args[])
{
System.out.println(x+B.y);
}
}
```

A. Compile time error
B. 10
C. 23
D. Runtime error

Q9(Output??)

Which of the following option is true regarding following code?

```
interface A
{
   void show();
}
public class Main{
public static void main(String args[])
{
A ref=()->System.out.println("Hello");
ref.show();
}
}
```

A. Given interface is functional interface
B. Given interface may contain more than one abstract methods
C. Given interface cannot have default and static methods
D. Given interface can have concrete methods

Q10(Output??)

```
interface A
{
   boolean task(int a,int b);
}
public class Main{
public static void main(String args[])
{
A ref=(n1,n2)->n1+2==n2;
System.out.println(ref.task(1,3));
}
}
```

A. false
B. true
C. Compile time error
D. Runtime error

Q11(Output??)


```
interface A
{
   int task(int a,int b);
}
public class Main{
public static void main(String args[])
{
A ref=(n1,n2)->{
int x=n1+n2;
int y=n1-n2;
int z=x+y;
};
System.out.println(ref.task(4,3));
}
}
```

A. 8
B. 0
C. Compile time error
D. Runtime error

Q12(Output??)

Which of the following option is true regarding following code?

```
interface A
{
   void show1();
   void show2();
}
public class Main implements A{
public void show1(){ System.out.println("Output 1"); }
public static void main(String args[])
{
A ref=new Main();
ref.show1();
}
}
```

A. It is not possible to have two abstract methods in A
B. Main class is abstract in nature as it is not overriding show2()
C. show1() is non-abstract method
D. Program will run and Output 1 will come on screen

Q13(Output??)

CODERINDEED

```
//What will be the output of following code?
interface A
{
   int i = 111;
}
class B implements A
{
   void methodB()
   {
      i = 222;
   }
}
public class Main
{
public static void main(String[] args)
{
   B ref=new B();
   ref.methodB();
}
}
```

A. 222
B. 111
C. Compile time error
D. Runtime error

Q14

Which of the following is a true statement about the functional interface?

A. It has exactly one method and it must be abstract.

B. It has exactly one method and it may or may not be abstract.

C. It must have exactly one abstract method and may have any number of default or static methods.

D. It must have exactly one default method and may have any number of abstract or static methods.

Q15

interface My { double sum(int x, double b); }
Which of the following statement is correct?

A. My x1 = (x, y) -> return x+y;
B. My x1 = (int x, double y) -> x+y;
C. My x1 = (double x, int y) -> return x+y;
D. None of These

Q16

Which of the following is FALSE about lambda expression?

A. Curly braces in expression body is optional if the body contains a single statement.

B. return keyword is optional if the body has a single expression to return the value.

C. type declaration of all parameters is always compulsory.

D. Both B & C

Q17

Which of the following is FALSE about interfaces in java.

A. Interfaces can have constructors and abstract methods.

B. An instance of interface can not be created.

C. An interface can contain default and static methods with bodies.

D. An interface can inherit multiple interfaces.

Q18

Given, abstract class A{}

```
class B extends A{
        B(int a){}
       }
```

Which of the following is INCORRECT?

A. A a = new A();

B. A a = new B(1);

C. B b = new A();

D. Both A & C

Q19

What should be the minimum changes in code given below to correct it?

```
public class Car    //Line 1
{
   public abstract void addFuel(); //Line2
}
```

A. Line 1 should be written as abstract public class Car

B. Line 2 should be written as public void addFuel(){}

C. Line 2 should be written as public void addFuel();

D. Either A or B

Q20(Output??)

```
@FunctionalInterface
interface A
{
   int value(int x);
}
public class Main
{
   public static void main(String[] args)
   {
   A ref=(n)->{int x=n;return x;};
   System.out.println(ref.value(2));
   }
}
```

A. 2
B. 0
C. Compile time error
D. Runtime error