Interfaces

Defining an Interface

An **Interface in Java** programming language is defined as an abstract type used to specify the behaviour of a class. An interface in Java is a blueprint of a behaviour. A Java interface contains static constants and abstract methods.

- The interface in Java is a mechanism to achieve abstraction.
- By default, variables in an interface are public, static, and final.
- It is used to achieve abstraction and multiple inheritances in Java.
- It is also used to achieve loose coupling.
- In other words, interfaces primarily define methods that other classes must implement.
- An interface in Java defines a set of behaviours that a class can implement, usually representing an IS-A relationship, but not always in every scenario.

Example:

```
import java.io.*;
interface testInterface {

    // public, static and final
    final int a = 10;
    // public and abstract
    void display();
}
// Class implementing interface
class TestClass implements testInterface {

    // Implementing the capabilities of
    // Interface
    public void display(){
      System.out.println("Demo");
    }
}

class A


{
    public static void main(String[] args)
    {
        TestClass t = new TestClass();
        t.display();
        System.out.println(t.a);
    }

}
```

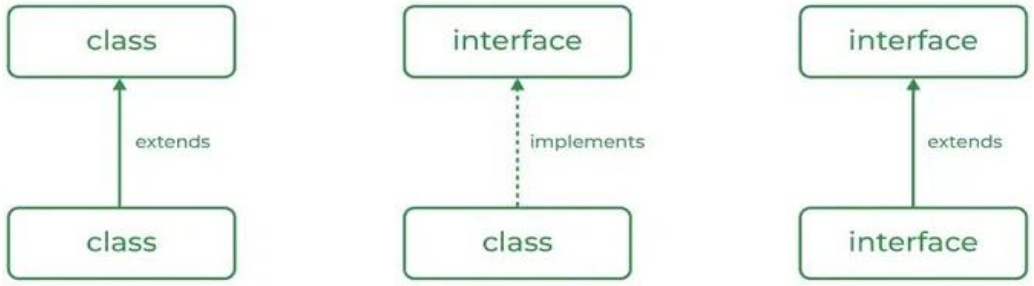
Note: In Java, the abstract keyword applies only to classes and methods, indicating that they cannot be instantiated directly and must be implemented. When we decide on a type of entity by its behaviour and not via attribute we should define it as an interface.

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 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 the interface, use the implements keyword.

Relationship Between Class and Interface

A class can extend another class, and similarly, an interface can extend another interface. However, only a class can implement an interface, and the reverse (an interface implementing a class) is not allowed.



Implementing Interfaces:

To implement an interface, we use the keyword **implements**

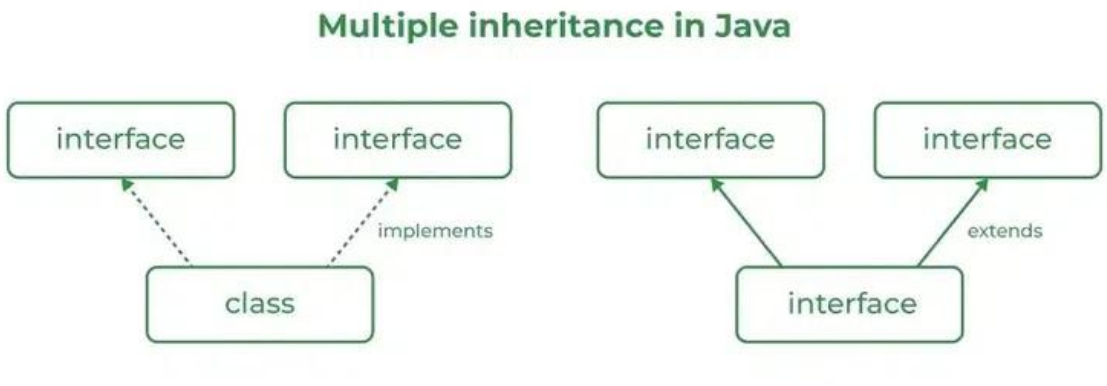
Example:

```
// Define an interface
interface Movable {
    void move();
}
// Implement the interface
class Car implements Movable {
    public void move() {
        System.out.println("Car is moving...");
    }
}
```

```
// Main class
public class InterfaceExample2 {
    public static void main(String[] args) {
        Movable car = new Car();  // Create a Car object
        car.move();  // Call the move method
    }
}
```

Multiple Inheritance in Java Using Interface

Multiple Inheritance is an OOPs concept that can't be implemented in Java using classes. But we can use multiple inheritances in Java using Interface. Let us check this with an example.



Example:

```
import java.io.*;
// Add interface
interface Add{
    int add(int a,int b);
}
// Sub interface
interface Sub{
      int sub(int a,int b);
}
// Calculator class implementing
// Add and Sub
class Cal implements Add , Sub
{
      // Method to add two numbers
      public int add(int a,int b){
          return a+b;
```

```
    }
        // Method to sub two numbers
      public int sub(int a,int b){
        return a-b;
    }
  }
class Calcex{
    // Main Method
    public static void main (String[] args)
    {
          // instance of Cal class
          Cal x = new Cal();
           System.out.println("Addition : " + x.add(2,1));
          System.out.println("Substraction : " + x.sub(2,1));
         }
}
```

Nested Interface in Java:

We can declare **interfaces as members of a class or another interface**. Such an interface is called a **member interface or nested interface**.

Interfaces declared outside any class can have only public and default access specifiers. In Java, **nested interfaces** (interfaces declared inside a class or another interface) can be declared with the **public, protected, package-private (default), or private access specifiers.**

- A nested interface can be declared public, protected, package-private (default), or private.
- A top-level interface (not nested) can only be declared as public or package-private (default).
- It cannot be declared as protected or private.

Syntax:

```
interface i_first{
interface i_second{
…
}
}
```

When implementing a nested interface, we refer to it as i_first.i_second, where i_first is the name of the interface in which the interface is nested and i_second is the interface's name.

An interface defined inside another interface or class is known as nested interface. Nested interface is also known as inner interfaces or member interfaces. Nested interfaces are generally used to group related interfaces.

The **syntax** of declaring nested interface is :

```
interface OuterInterfaceName
{
  interface InnerInterfaceName
    {
      // constant declarations
      // Method declarations
    }
 }
```

Example :

```
interface MyInterface {
interface MyInnerInterface {
 int id = 20;
void print();
 } }
```

Nested interface can be defined anywhere inside outer interface or class.

Some of the key points about nested interfaces are :

- Nested interfaces are static by default, irrespective of you declare it static or not.
- Nested interfaces are accessed using outer interface or class name.
- Nested interfaces declared inside a class can have any access modifier, while nested interfaces declared inside another interfaces are public by default.
- Classes implementing inner interface are required to implement only inner interface methods, not outer interface methods.
- Classes implementing outer interface are required to implement only outer interface methods, not inner interface methods.

Nested interface program in Java

```
interface MyInterface {
   void calculateArea();
   interface MyInnerInterface {
      int id = 20;
      void print();
   }
 }

class NestedInterface implements MyInterface.MyInnerInterface {
   public void print() {
      System.out.println("Print method of nested interface");
    }
   public static void main(String args []) {
      NestedInterface obj = new NestedInterface();
      obj.print();
      System.out.println(obj.id);
   }
}
```

Output:

```
Print method of nested interface
20
```

As mentioned above nested interface can be declared inside a class as well. Class implementing such interface needs to define all the methods of that nested interface. The program below demonstrates the nested interface declared inside a class.

```
class OuterClass {
   interface MyInnerInterface {
      int id = 20;
      void print();
   }
 }

class NestedInterfaceDemo implements OuterClass.MyInnerInterface {
   public void print() {
      System.out.println("Print method of nested interface");
    }
   public static void main(String args []) {
      NestedInterfaceDemo obj = new NestedInterfaceDemo();
      obj.print();
      System.out.println(obj.id);
      // Assigning the object into nested interface type
      OuterClass.MyInnerInterface obj2 = new NestedInterfaceDemo();
      obj2.print();
   }
}
```

The object of implementing class can also be assigned in to nested interface type, as in above program obj2 is an object of NestedInterfaceDemo class but assigned into MyInnerInterface type. As mentioned above class implementing an outer interface are not required to implement the inner or nested interface methods. The program below demonstrates the same.

```
interface MyInterface {
   void calculateArea();
   interface MyInnerInterface {
      int id = 20;
      void print();
    }
 }

class OuterInterfaceTest implements MyInterface {
   public void calculateArea() {
      System.out.println("Calculate area inside this method");
    }
   public static void main(String args []) {
      OuterInterfaceTest obj = new OuterInterfaceTest();
      obj.calculateArea();
    }
}
```

Why do we use nested interface in java

There are couple of reasons for using nested interfaces in java :

- Nesting of interfaces is a way of logically grouping interfaces which are related or used at one place only.
- Nesting of interfaces helps to write more readable and maintainable code.
- It also increases encapsulation.

NOTE:

- An interface or class can have any number of inner interfaces inside it.
- Inner interface can extend it's outer interface.
- Inner and outer interfaces can have method and variables with same name.
- Nested interfaces can also have default and static methods from java 8 onward.
- An inner class defined inside an interface can implement the interface.
- Nesting of interfaces can be done any number of times. As a good practice you should avoid doing nesting more than once.

Applying Interfaces

The interface is a powerful tool. The same interface can be used by different classes for some method. Then this method can be implemented by each class in its own way. Thus same interface can provide variety of implementation. The selection of different implementations is done at the run time. Following is a simple Java program which illustrates this idea.

Step 1: Write a simple interface as follows

Java Program [interface1.java]

```
interface interface1

{

void MyMsg(String s);

}
```

Step 2: Write a simple Java Program having two classes having their own implementation of MyMsg method.

Java Program[Test.java]

```
import java.io.*;
import java.util.*;
class Class1 implements interface1
{
private String s;
public void MyMsg(String s)
{
System.out.println("Hello "+s);
}
}
class Class2 implements interface1
{
private String s;
public void MyMsg(String s)
{
System.out.println("Hello "+s);
}
}
class Test
{
public static void main(String[] args)
{
interface1 inter;
Class1 obj1=new Class1();
Class2 obj2=new Class2();
inter=obj1;
inter.MyMsg("User1");
inter=obj2;
inter.MyMsg("User2");
}
}
```

Step 3: Execute the program.

```
Javac test.java
Java test
Hello User1
Hello User 2
```

Variables in Interfaces

- Every interface variable is always public, static, final whether we are declaring or not.
- If any variable in an interface is defined without public, static and final keywords the compiler automatically add the same.

Ex:

Interface int1

{

Int X =10;

}

Public: to make this variable available for every implementation class.

Static: Without existing object also implementation class can access this variable.

Final: Implementation class can access this variable but cannot modify.

Hence inside the interface the following declaration are valid and equal.

```
interface Interf {
   int x = 10;
}

class Test implements Interf {
   public static void main(String args[]) {
      x = 888; //C.E
      System.out.println(x);
   }
}
```

- Interface variables are public, static and final we Cannot declare with following modifiers.

Private

Protected

<default>

Transient

Volatile

- Inside Implementation classes we can access interface variables but we can't modify their values.

```
int x = 10;
public int x = 10;
public static int x = 10;
public static final int x = 10;
public static int x = 10;
final int x = 10;
public final int x = 10;
static final int x = 10;
```

In Java interfaces, both default and static methods allow for adding implementations to interfaces without breaking backward compatibility with existing implementations, but they differ in how they are invoked and their purpose.

Default Methods:

- **Purpose:** Provide a default implementation for methods in an interface, allowing classes implementing the interface to inherit that implementation or override it.
- **Invocation:** Default methods are invoked on instances of the implementing class, just like regular instance methods.

```
interface MyInterface {
    void abstractMethod(); // Abstract method
    default void defaultMethod() {
        System.out.println("Default implementation");
    }
}
```

Static Methods:

- **Purpose:** Define utility methods that belong to the interface itself, not to any instance of the interface.
- **Invocation:** Static methods are invoked using the interface name, not an instance of the implementing class.

```
interface MyInterface {
    void abstractMethod();
    static void staticMethod() {
        System.out.println("Static implementation");
    }
}
```

Default Methods:

- Default and static methods were introduced in Java 8 to enhance the functionality of interfaces.
- Default methods provide a way to add new methods to an interface without breaking existing implementations.
- They are declared using the default keyword and include a method body, offering a default implementation.
- Implementing classes can choose to override default methods or inherit the default implementation.

Example:

```
interface MyInterface {
    void abstractMethod();
    default void defaultMethod()
{
    System.out.println("Default method implementation");
  }
}

class MyClass implements MyInterface {
   @Override
  public void abstractMethod() {
   System.out.println("Abstract method implementation");
  }
 // Inherits defaultMethod() from MyInterface
}
public class Main {
 public static void main(String[] args)
 {
 MyClass obj = new MyClass();
 obj.abstractMethod(); // Output: Abstract method implementation
 obj.defaultMethod();  // Output: Default method implementation
  }
}
```

Static Methods:

- Static methods in interfaces are similar to static methods in classes.
- They are declared using the static keyword and belong to the interface itself, not to instances of implementing classes.
- Static methods are called using the interface name and are often used for utility or helper functions related to the interface.

Example:

```
interface MyInterface
{
static void staticMethod()
{
System.out.println("Static method implementation");
}
}
public class Main
{
 public static void main(String[] args)
{
MyInterface.staticMethod(); // Output: Static method implementation
}
}
```