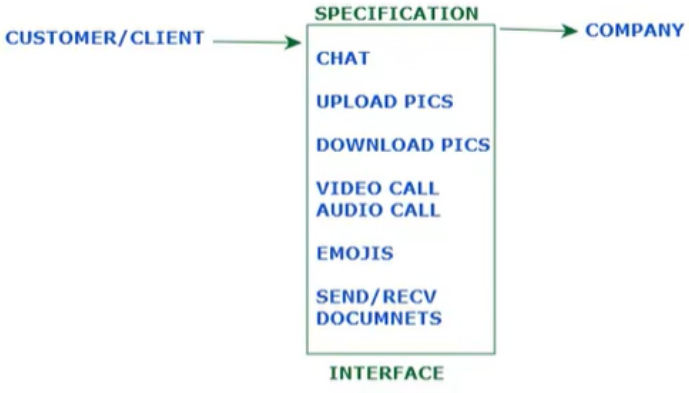
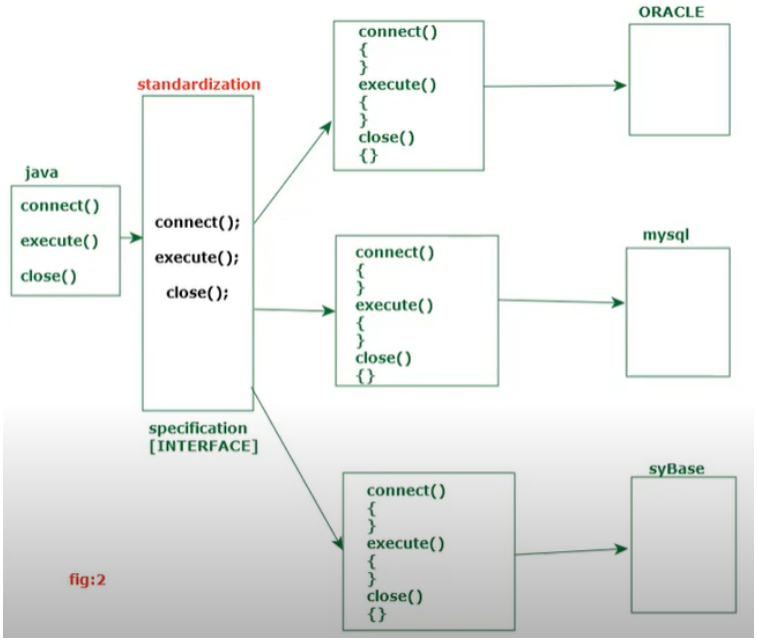
Interface

Table Of Content

Introduction



- Generally **INTERFACE** is a specification.



- In Java aspect Interface is **collection of abstract methods using interface 100% abstraction** can be achieved because inside the interface methods declaration are allowed
- **Interface is a collection of abstract methods**

Interface is a pillar of Object Oriented Programming Language which is used to bring **standardization to a java program**
Above figure is a example for interface

let's consider example by assume that while doing **JDBC Connection** we have many Database providers each providers have different approaches(connection, execution, termination), if each providers use their own approaches means the developers need to remember the approaches so this is not a good aspect **So in-order to bring standardization to different providers we will use interface in the same way for our project we can use interface & abstraction to bring standardization & 100% abstraction**

interface syntax

```
SYNTAX:
-------
interface interface_Name
{
  abstract method();      --> method
  Constants;        --> variables(final)
}
```

Advantages

1. Using the interface standardization is achieved. eg: pi value
2. It promotes polymorphism
3. 100% abstraction can be achieved
4. multiple inheritance can be achieved.

```
ADVANTAGE OF INTERFACE:
-----------------------
1. Using the interface standarization is achived.
eg: pi value

2. It promotes polymorphism

3. 100% abstraction can be achived

4. multiple inheritance can be achived.
```

NOTE:
By default all the data members(variable) present in the interface are 'public static final' in nature. Also by default all the methods present in the interface is 'public abstract' in nature.

Simple Words
1. In interface all methods are public abstract in nature
2. In interface all variables are public static final in nature

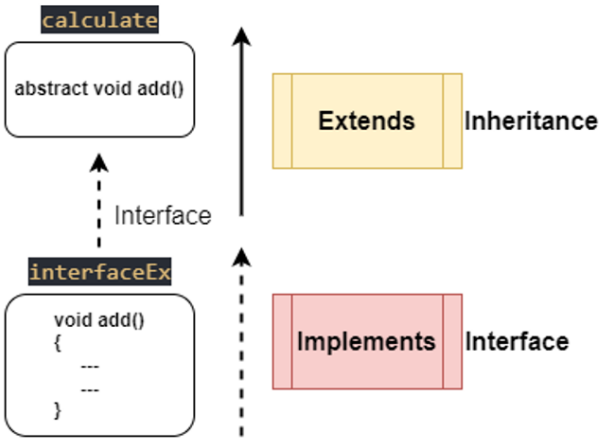
```
interface Test
{
  float pi = 3.14f;
  void fun();
}

compile:

interface Test
{
  public static final float pi = 3.14f; //by default public static final
  public abstract void fun();// by
}

Enter a: 10
Enter b: 20
The sum of 10 and 20 is 30

Process finished with exit code 0
```



▼ **Interface Example**

```
package Object_Oriented_Concepts.GettingStartedToJava.Interface;

import java.util.Scanner;

/** @interface_class */
```

```
interface calculate
{
    //Note - Interface methods are always abstract and its access modifier is always public
    public abstract  void add();
}

class interfaceEx implements calculate
{
    public void add()
    {
        int a;
        int b;
        int sum;

        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a: ");
        a= scanner.nextInt();
        System.out.println("Enter b: ");
        b= scanner.nextInt();

        sum = a+b;
        System.out.println("The sum of "+a+" and "+b+" is "+sum);
    }
}

public class _1_interfaceEx1
{
    public static void main(String[] args)
    {
        interfaceEx ex = new interfaceEx();
        ex.add();
    }
}
//output
```

```
public   weaker access modifiers can be given
protected
default
private
```

Rules In Interface

Rule 1

- When a class is implementing the interface **the methods access modifiers should be made as public** because by default all the abstract method in interface are public in nature.

▼ **RULE 1**

```
package Object_Oriented_Concepts.GettingStartedToJava.Interface;

import javax.naming.Name;

interface data
{
    //interface methods should be always public in nature, and also it should be abstract method
    // the access modifiers and abstract method and access modifiers of variable is automatically added by java compiler
    public abstract void myData();
}

class info implements data
{
    public void myData()
    {
        String Name, Email, Branch;
        Name = "Nandan G N";
        Email = "gnnandan7@gmail.com";
        Branch = "Computer Science";
        System.out.println("Name: "+Name);
        System.out.println("Branch: "+Branch);
        System.out.println("Email: "+Email);
    }
```

```
}


public class Rules_1
{
    public static void main(String[] args)
    {
        info info_Object = new info();
        info_Object.myData();
    }
}
//output
Name: Nandan G N
Branch: Computer Science
Email: gnnandan7@gmail.com

Process finished with exit code 0
```

Rule 2

- If a **class implements the interface then it should provide the body for all the methods present** in the interface class. If the class is **not in a position to provide the body for all the method then class should be made as abstract.**

▼ **RULE 2**

```
package Object_Oriented_Concepts.GettingStartedToJava.Interface;


//If a class implements the interface then it should provide the body for all the methods present in the interface class.
//If the class is not in a position to provide the body for all the method  then class should be made as abstract.

interface calculates
{
    abstract void add();
    abstract void product();
}

abstract class addition implements calculate
{
    public void add()
    {
        int a = 10, b = 20, c;
        c = a+b;

        System.out.println("The sum of "+a+" and "+b+" is "+c);
    }
    abstract public void product();
}

abstract class products implements calculates
{

    public void product()
    {
        int a = 10, b = 20, c;
        c = a*b;
        System.out.println("The product of "+a+" and "+b+" is "+c);
    }
      abstract public void add();
}

public class Rule_2
{
    public static void main(String[] args)
    {

    }
}
```

Rule 3

- An object of **abstract class can not be created** and An **object of interface cannot be created.**

Rule 4

- Though we cannot create the object for interface **its reference can be created**.

Rule 5

- A **class and interface cannot** be related using the **extends keyword**

Rule 6

- Using the **interface ref (parent ref)** the implemented methods of a **class can be accessed** and using **interface ref polymorphism** can be achieved

▼ **RULE 6**

```
package Object_Oriented_Concepts.GettingStartedToJava.Interface;

interface bank
{
    abstract void rateOfInterest();
}

class sbi implements bank
{
    public void rateOfInterest()
    {
        System.out.println("The SBI bank's interest is 7.5%");
    }
}

class hdbf implements  bank
{
    public void rateOfInterest()
    {
        System.out.println("The HDFC bank's interest is 9.5%");
    }
}
class canara implements  bank
{
    public void rateOfInterest()
    {
        System.out.println("The CANARA bank's interest is 8.5%");
    }
}


// Note here
class combined
{
    //interface class reference
    public void banks(bank b)
    {
        b.rateOfInterest();
    }
}

public class Rule_6
{
    public static void main(String[] args)
    {
        sbi b1 = new sbi();
        hdbf b2 = new hdbf();
        canara b3 = new canara();

        combined c =  new combined();
        c.banks(b1);
        c.banks(b2);
        c.banks(b3);
    }
}
//output
The SBI bank's interest is 7.5%
The HDFC bank's interest is 9.5%
The CANARA bank's interest is 8.5%
```

Rule 7

- Using the **interface ref** only the overridden method can be accessed in order to access the specialized methods **downcasting has to be performed**.

▼ **RULE 7**

```
package Object_Oriented_Concepts.GettingStartedToJava.Interface;

interface union
{
    abstract void goal();
    abstract void branch();
}

class nandan implements union
{
    public void goal()
    {
        System.out.println("My goal is to become 'Entrepreneur'");
    }
    public void branch()
    {
        System.out.println("My Branch is 'Computer Science'");
    }

    //special method
    public void job()
    {
        System.out.println("Currently i'm working as a 'Software Engineer'");
    }
}

class yashas implements union
{
    public void goal()
    {
        System.out.println("My goal is to become 'Enginner'");
    }
    public void branch()
    {
        System.out.println("My Branch is 'Mechanical Engineering'");
    }

    public void job()
    {
        System.out.println("Currently i'm working as a 'System Engineer'");
    }
}

class reference
{
    public void reunion(union u)
    {
        u.goal();
        u.branch();
    }
}

public class Rule_7
{
    public static void main(String[] args)
    {
        nandan p1 = new nandan();
        yashas p2 = new yashas();
        reference r = new reference();

        r.reunion(p1);
        System.out.println();
        r.reunion(p2);

        System.out.println();
        ((nandan)(p1)).job();
        System.out.println();
        ((yashas)(p2)).job();
    }
}
//output
My goal is to become 'Entrepreneur'
My Branch is 'Computer Science'
```

```
My goal is to become 'Enginner'
My Branch is 'Mechanical Engineering'

Currently i'm working as a 'Software Engineer'

Currently i'm working as a 'System Engineer'

Process finished with exit code 0
```

Rule 8

- A interface class can be implemented **any number of interface**

▼ **RULE 8**

```
package Object_Oriented_Concepts.GettingStartedToJava.Interface;

interface c1
{
    abstract void add();
}

interface c2
{
    abstract void sub();
}

class result implements c1,c2
{
    public void add()
    {
        int a=10,b=20,c;
        c = a + b;
        System.out.println("The sum of "+a+" and "+b+" is "+c);
    }
    public void sub()
    {
        int a=20,b=10,c;
        c= a-b;
        System.out.println("The difference between "+a+" and "+b+" is "+c);
    }
}

public class Rule_8
{
    public static void main(String[] args)
    {
        result result = new result();
        result.add();
        result.sub();
    }
}
//output
The sum of 10 and 20 is 30
The difference between 20 and 10 is 10
```

Rule 9, 10

- **RULE 9** - If two interface contains **same method signature and same return type** then the implementation class **can provide the body for only one method**.
- **RULE 10** - If two interface contains **same method name and return type and change in parameter** then the implementation class should provide the body **for all the methods.**

▼ **RULE 9, 10**

```
package Object_Oriented_Concepts.GettingStartedToJava.Interface;

interface data1
{
    abstract void person(String name, int age);
}
```

```
interface data2
{
    abstract void person(String branch, String email);
}

class colab implements data1,data2
{
    //method name is same but parameter is different so there is two body of function
    public void person(String name, int age)
    {
        System.out.println("Name: "+name);
        System.out.println("Age: "+age);
    }
    public void person(String branch, String email)
    {
        System.out.println("Branch: "+branch);
        System.out.println("Email: "+email);
    }
}

public class Rule_10
{
    public static void main(String[] args)
    {
        colab colab = new colab();
        colab.person("Nandan",22);
        colab.person("CSE","gnnandan7@gmail.com");
    }
}
//output
Name: Nandan
Age: 22
Branch: CSE
Email: gnnandan7@gmail.com
```

Rule 11

- In interface if the method signature is same and the return type is different then the implementing class can not provide the body.
  If implementing class provide the body results in ambiguity problem.

Rule 12

- If the implementing class is not able to provide the body for all the methods of interface then **the implementing class should be made as abstract**.
  The **child class of the implementing class can provide the body** for the remaining methods of interface.

Rule 13

- If the implementing class is not able to provide the body for all the methods of interface then **the implementing class should be made as abstract**.
  The **child class of the implementing class can provide the body** for the remaining methods of interface.

▼ **RULE 13**

```
package Object_Oriented_Concepts.GettingStartedToJava.Interface;

interface operation
{
    abstract void add();
    abstract void diff();
    abstract void product();
}

abstract class calculator implements operation
{
    public void add()
    {
        int a = 10, b = 20, c;
        c = a+b;
        System.out.println("The sum of "+a+" and "+b+" is "+c);
    }
```

```
    abstract public void diff();
    abstract public void product();
}

//child class gives body if abstract is not able to give the body
class juniorCalculator extends calculator
{
    public void add()
    {
        int a = 10, b = 20, c;
        c = a+b;
        System.out.println("The sum of "+a+" and "+b+" is "+c);
    }
    public void diff()
    {
        int a = 20, b = 10, c;
        c = a-b;
        System.out.println("The difference of "+a+" and "+b+" is "+c);
    }
    public void product()
    {
        int a = 10, b = 20, c;
        c = a*b;
        System.out.println("The product of "+a+" and "+b+" is "+c);
    }

}

public class Rule_13
{
    public static void main(String[] args)
    {
        juniorCalculator jc = new juniorCalculator();
        jc.add();
        jc.diff();
        jc.product();
    }
}
//output
The sum of 10 and 20 is 30
The difference of 20 and 10 is 10
The product of 10 and 20 is 200
```

Rule 14

- An interface can extend any number of interface.

▼ **RULE 14**

```
package Object_Oriented_Concepts.GettingStartedToJava.Interface;

interface calculate1
{
    void add();
}
interface calculate2
{
    void sub();
}
interface calculate3 extends calculate1,calculate2
{
    void mul();
}

class Test implements calculate3
{
    public void add()
    {
        int a,b,c;
        a=10;
        b=20;
        c=a+b;
        System.out.println("Sum: "+c);
    }
    public void sub()
    {
        int a,b,c;
        a=20;
```

```
        b=05;
        c=a-b;
        System.out.println("Difference: "+c);
    }
    public void mul()
    {
        int a,b,c;
        a=20;
        b=05;
        c=a*b;
        System.out.println("Product: "+c);
    }

}

public class Rule_14
{
    public static void main(String[] args)
    {
        Test test = new Test();
        test.add();
        test.sub();
        test.mul();
    }
}

//output
Sum: 30
Difference: 15
Product: 100
```

Rule 15

- A class implements a interface, the interface extends another interface then the **implementing class should provide the body for all the abstract methods** in the hierarchy

▼ **RULE 15**

```
package Object_Oriented_Concepts.Interface;

interface a1
{
    abstract void name();
}

interface a2 extends a1
{
    abstract void fullname();
}

class a implements a2
{
    public void name()
    {
        System.out.println("My name is 'Nandan' ");
    }
    public void fullname()
    {
        System.out.println("My fullname is 'Nandan G N' ");
    }
}

public class Rule_15
{
    public static void main(String[] args)
    {
        a implementsClass = new a();
        implementsClass.name();
        implementsClass.fullname();
    }
}
//output
My name is 'Nandan'
My fullname is 'Nandan G N'
```

Rule 16

- An interface **cannot implement another interface**.

▼ **RULE 16**

```
interface A
{
  void add();
}
interface B implements A
{
  void sub();
}
//OUTPUT:
compilation error
```

Rule 17

- A **class can extends another class simultaneously it can implement the interface also** in this case the implementing class should **extends first** and it
  should implements

▼ **RULE 17**

```
package Object_Oriented_Concepts.Interface;

class normalClass
{
    public void insideNormalClass()
    {
        System.out.println("Inside the normal class");
    }
}

interface interfaceClass
{
    abstract void insideInterfaceClass();
}


class extendsClass extends normalClass implements interfaceClass
{
    public void insideInterfaceClass()
    {
        System.out.println("Inside the interface class");
    }
}

public class Rule_17
{
    public static void main(String[] args)
    {
        extendsClass extendsClassObject = new extendsClass();
        extendsClassObject.insideNormalClass();
        extendsClassObject.insideInterfaceClass();
    }
}
//output
Inside the normal class
Inside the interface class
```

Adapter Class

- Adapter class is a normal class in java which **as empty body for all the abstract methods** of an interface.

▼ **Adapter Class Example**

```
package Object_Oriented_Concepts.Interface.Adapter_Class;

interface adbstractMethods
```

```
{
    abstract void method1();
    abstract void method2();
    abstract void method3();
    abstract void method4();
}

class Adapter implements adbstractMethods
{
    @Override
    public void method1()
    {

    }

    @Override
    public void method2()
    {
         //adapter class can have empty methods
    }

    @Override
    public void method3()
    {
         //adapter class can have empty methods
    }

    @Override
    public void method4()
    {

    }
}

class test1 extends Adapter
{
    public void method2()
    {
        System.out.println("inside method 2");
    }
}

class test2 extends  Adapter
{
    public void method3()
    {
        System.out.println("inside method 3");
    }
}

public class _1_adapterClassEx1
{
    public static void main(String[] args)
    {
        test1 t1= new test1();
        t1.method2();
        test2 t2 = new test2();
        t2.method3();
    }
}
//output
inside method 2
inside method 3
```

InstanceOf Operator

- It is a Boolean operator, it can be used to **check whether the specified reference variable is representing the specified class object or not that is compatible or not**.
- If **ref_Var** class is **same** as the specified **Class_Name** then instanceof operator will return **"true"**.
  If **ref_Var** class is **subclass** to the specified **Class_Name** then instanceof operator will return **"true"**.
  If **ref_Var** class is **superclass** to the specified **Class_Name** then instanceof operator will return **"false"**.
- **Syntax**

```
ref_Var instanceof Class_Name
```

▼ **InstanceOf Example**

```
package Object_Oriented_Concepts.Interface.InstanceOf_Operator;


abstract class animals
{
    abstract void animalName();
    abstract void animalColor();
}

class cat extends animals
{
    String name = "CAT";
    public void animalName()
    {
        System.out.println("Animal Name is: "+name);
    }
    public void animalColor()
    {
        System.out.println(name+" color is white and so on...!");
    }
}

class dog extends animals
{
    String name = "DOG";
    public void animalName()
    {
        System.out.println("Animal Name is: "+name);
    }
    public void animalColor()
    {
        System.out.println(name+" color is brown and so on...!");
    }
}


public class instanceOf_OperatorEx
{
    public static void main(String[] args)
    {
        cat a1 = new cat();
        dog a2 = new dog();
        a1.animalName();
        a1.animalColor();
        System.out.println();

        a2.animalName();
        a2.animalColor();

        //let's check the instance of
        /**@instanceOf
         * It is a boolean operator,it can be used to check whether the specified reference variable is
         * representing the specified class object or not that is compatible or not.
         */
        System.out.print("checking whether 'a1' is instance of class 'cat': ");
        System.out.println(a1 instanceof cat);
        System.out.print("checking whether 'a2' is instance of class 'dog': ");
        System.out.println(a2 instanceof dog);
    }
}
//output
Animal Name is: CAT
CAT color is white and so on...!

Animal Name is: DOG
DOG color is brown and so on...!
checking whether 'a1' is instance of class 'cat': true
checking whether 'a2' is instance of class 'dog': true
```

Note

Difference b/w extends and implements

- **extends**:
    1. A class can inherit other class using the extends keyword.An interface can inherit another interface using the extends keyword.
    2. A subclass which extends super class may or may not override all the methods of superclass
    3. An interface can extends any number of interface
    4. A class can extends only one super class.
- **implements**:
    1. A class can be implemented other interface using implements keyword.
    2. class implementing interface should provide body for all methods of interface
    3. An interface can never implements other interface
    4. A class can implement multiple interface simultaneously.

Difference b/w interface and abstract class

- **interface:**
    1. 100% abstraction is achieved
    2. interface are used when we don't know complete implementation
    3. only abstract methods are allowed
    4. Every methods inside interface are by default public and abstract
    5. Every variables inside interface are by default public static final
    6. following modifiers are not allowed for abstract methods static, private, strictfp, protected, final
    7. interface cannot have static blocks, instance blocks
    8. constructors are not allowed
- **abstract class:**
    1. 0-100% abstraction can be achieved
    2. abstract class is used when we know parent implementation
    3. both abstract and concrete methods are allowed
    4. every method present in abstract class need not to be public and abstract
    5. every variables present in abstract class need not to be public static final
    6. no such restrictions for methods
    7. abstract class can have static block and instance block
    8. constructors are allowed