LECTURE NOTES

UNIT 2

UNIT-II INHERITANCE AND INTERFACES
Extended Class, Constructors in Extended classes, Inheriting and Redefining Members
Type Compatibility and Conversion, protected, final Methods and Classes, Abstract methods and classes
Object Class, Cloning Objects, Designing extended classes, Single Inheritance versus Multiple Inheritance
Interface, Interface Declarations, Extending Interfaces
Working with Interfaces, Marker Interfaces, When to Use Interfaces

1.1. INHERITANCE IN JAVA

Inheritance is a mechanism in which one object acquires all the properties and behaviours of parent object.The idea behind inheritance is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you reuse (or inherit) methods and fields, and you add new methods and fields to adapt your new class to new situations.

Inheritance represents the **IS-A relationship**.

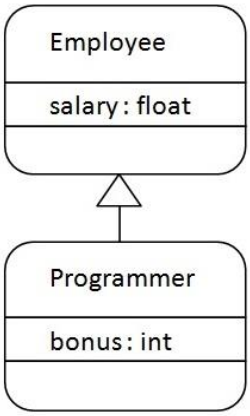
USE OF INHERITANCE

- For Method Overriding (So Runtime Polymorphism).
- For Code Reusability.

SYNTAX OF INHERITANCE

```
1. class Subclass-name extends Superclass-name
2. {
3.   //methods and fields
4. }
```

The keyword extends indicates that you are making a new class that derives from an existing class. In the terminology of Java, a class that is inherited is called a superclass. The new class is called a subclass.



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**.It means that Programmer is a type of Employee.

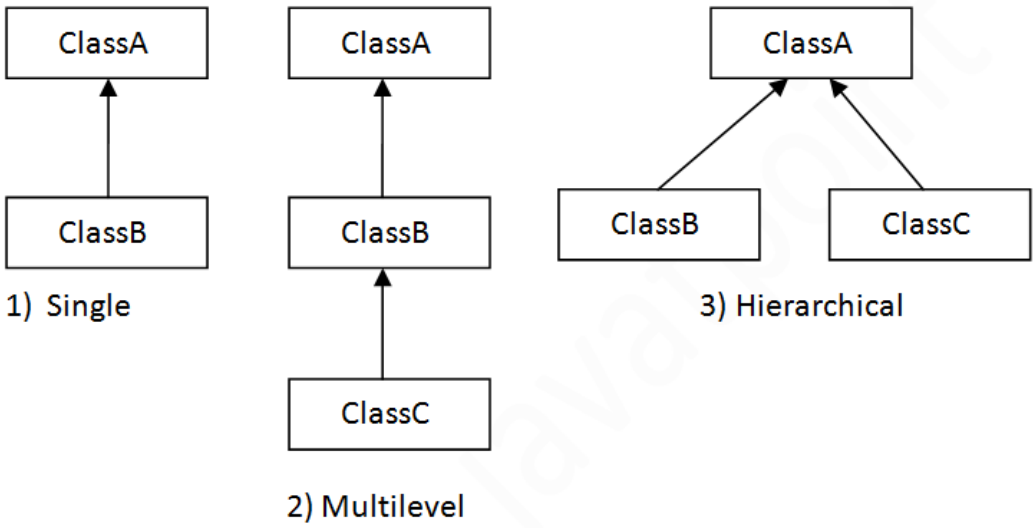
```
1. class Employee{
2.  float salary=40000;
3. }
4.
5. class Programmer extends Employee{
6.  int bonus=10000;
7.
8.  public static void main(String args[]){
9.    Programmer p=new Programmer();
10.   System.out.println("Programmer salary is:"+p.salary);
11.   System.out.println("Bonus of Programmer is:"+p.bonus);
12. }
13. }
```

```
Output:Programmer salary is:40000.0
       Bonus of programmer is:10000
```

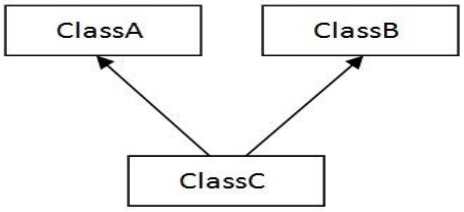
In the above example,Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

1.1.1. TYPES OF INHERITANCE

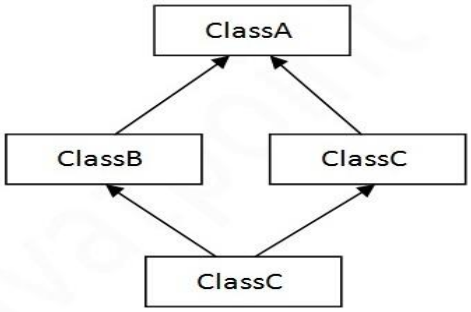
On the basis of class, there can be three types of inheritance: single, multilevel and hierarchical. Multiple and Hybrid is supported through interface only. We will learn about interfaces later.



When a class extends multiple classes i.e. known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

.

WHY MULTIPLE INHERITANCE IS NOT SUPPORTED IN JAVA?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java. For example:

```
1. class A{
2. void msg(){System.out.println("Hello");}
3. }
4.
5. class B{
6. void msg(){System.out.println("Welcome");}
7. }
8.
9. class C extends A,B{//suppose if it were
10.
11. Public Static void main(String args[]){
12.   C obj=new C();
13.   obj.msg();//Now which msg() method would be invoked?
14. }
15. }
```

1.1.2. AGGREGATION IN JAVA

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

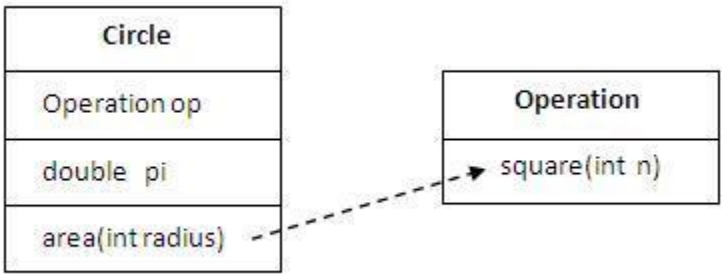
```
1. class Employee{
2. int id;
3. String name;
4. Address address;//Address is a class
5. ...
6. }
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

WHY USE AGGREGATION

- For Code Reusability.

SIMPLE EXAMPLE OF AGGREGATION



In this example, we have created the reference of Operation class in the Circle class.

```
1. class Operation{
2.  int square(int n){
3.   return n*n;
4.  }
```

```
5.  }
6.
7.  class Circle{
8.   Operation op;//aggregation
9.   double pi=3.14;
10.
11. double area(int radius){
12.   op=new Operation();
13.   int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
14.   return pi*rsquare;
15. }
16.
17.
18.
19. public static void main(String args[]){
20.   Circle c=new Circle();
21.   double result=c.area(5);
22.   System.out.println(result);
23. }
24. }
```

```
Output:78.5
```

WHEN USE AGGREGATION?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

Address.java

```
1.  public class Address {
2.  String city,state,country;
3.
4.  public Address(String city, String state, String country) {
5.      this.city = city;
6.      this.state = state;
7.      this.country = country;
8.  }
9.
10. }
```

Emp.java

```
1. public class Emp {
2. int id;
3. String name;
4. Address address;
5.
6. public Emp(int id, String name,Address address) {
7.     this.id = id;
8.     this.name = name;
9.     this.address=address;
10. }
11.
12. void display(){
13. System.out.println(id+" "+name);
14. System.out.println(address.city+" "+address.state+" "+address.country);
15. }
16.
17. public static void main(String[] args) {
18. Address address1=new Address("gzb","UP","india");
19. Address address2=new Address("gno","UP","india");
20.
21. Emp e=new Emp(111,"varun",address1);
22. Emp e2=new Emp(112,"arun",address2);
23.
24. e.display();
25. e2.display();
26.
27. }
28. }
```

```
Output:111 varun
       gzb UP india
       112 arun
       gno UP india
```

1.1.3. METHOD OVERRIDING IN JAVA

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding**.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as Method Overriding.

ADVANTAGE OF JAVA METHOD OVERRIDING

- Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method Overriding is used for Runtime Polymorphism

RULES FOR METHOD OVERRIDING

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

UNDERSTANDING THE PROBLEM WITHOUT METHOD OVERRIDING

Let's understand the problem that we may face in the program if we don't use method overriding.

```
1.  class Vehicle{
2.  void run(){System.out.println("Vehicle is running");}
3.  }
4.  class Bike extends Vehicle{
5.
6.  public static void main(String args[]){
7.  Bike obj = new Bike();
8.  obj.run();
9.  }
10. }
```

```
Output:Vehicle is running
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
1.  class Vehicle{
2.  void run(){System.out.println("Vehicle is running");}
3.  }
4.  class Bike extends Vehicle{
5.  void run(){System.out.println("Bike is running safely");}
6.
7.  public static void main(String args[]){
8.  Bike obj = new Bike();
9.  obj.run();
10. }
```

```
Output:Bike is running safely
```

REAL EXAMPLE OF JAVA METHOD OVERRIDING

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.

```
1. class Bank{
2. int getRateOfInterest(){return 0;}
3. }
4.
5. class SBI extends Bank{
6. int getRateOfInterest(){return 8;}
7. }
8.
9. class ICICI extends Bank{
10. int getRateOfInterest(){return 7;}
11. }
12. class AXIS extends Bank{
13. int getRateOfInterest(){return 9;}
14. }
15.
16. class Test{
17. public static void main(String args[]){
18. SBI s=new SBI();
19. ICICI i=new ICICI();
20. AXIS a=new AXIS();
21. System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
22. System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
23. System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
24. }
25. }
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

CAN WE OVERRIDE STATIC METHOD?

No, static method cannot be overridden. It can be proved by runtime polymorphism so

we will learn it later.

WHY WE CANNOT OVERRIDE STATIC METHOD?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

CAN WE OVERRIDE JAVA MAIN METHOD?

No, because main is a static method.

WHAT IS THE DIFFERENCE BETWEEN METHOD OVERLOADING AND METHOD OVERRIDING?

There are three basic differences between the method overloading and method overriding. They are as follows:

Method Overloading	Method Overriding
1) Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2) method overloading is performed within a class.	Method overriding occurs in two classes that have IS-A relationship.
3) In case of method overloading parameter must be different.	In case of method overriding parameter must be same.

1.2. SUPER KEYWORD

The **super** is a reference variable that is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

USAGE OF SUPER KEYWORD

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

1) SUPER IS USED TO REFER IMMEDIATE PARENT CLASS INSTANCE VARIABLE.

Problem without super keyword

```
1. class Vehicle{
2.   int speed=50;
3. }
4.
5. class Bike extends Vehicle{
6.   int speed=100;
7.
8.   void display(){
9.    System.out.println(speed);//will print speed of Bike
10.  }
11.  public static void main(String args[]){
12.   Bike b=new Bike();
13.   b.display();
14.
15.}
16.}
```

```
Output:100
```

In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is refered by instance bydefault, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable.

Solution by super keyword

```
1. //example of super keyword
2.
3. class Vehicle{
4.   int speed=50;
5. }
```

```
6.
7.  class Bike extends Vehicle{
8.    int speed=100;
9.
10.   void display(){
11.    System.out.println(super.speed);//will print speed of Vehicle now
12.   }
13.   public static void main(String args[]){
14.    Bike b=new Bike();
15.    b.display();
16.
17. }
18. }
```

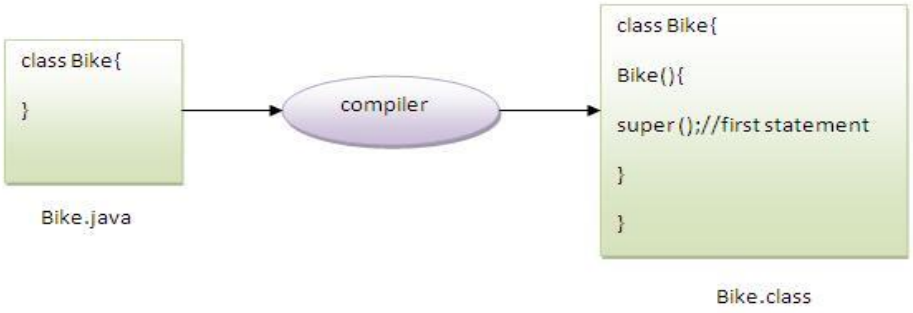
```
Output:50
```

2) SUPER IS USED TO INVOKE PARENT CLASS CONSTRUCTOR.

The super keyword can also be used to invoke the parent class constructor as given below:

```
1.  class Vehicle{
2.    Vehicle(){System.out.println("Vehicle is created");}
3.  }
4.
5.  class Bike extends Vehicle{
6.    Bike(){
7.     super();//will invoke parent class constructor
8.     System.out.println("Bike is created");
9.    }
10.   public static void main(String args[]){
11.    Bike b=new Bike();
12.
13. }
14. }
```

```
Output:Vehicle is created
       Bike is created
```



As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

EXAMPLE OF SUPER KEYWORD WHERE SUPER() IS PROVIDED BY THE COMPILER IMPLICITLY.

```
1. class Vehicle{
2.   Vehicle(){System.out.println("Vehicle is created");}
3. }
4.
5. class Bike extends Vehicle{
6.   int speed;
7.   Bike(int speed){
8.     this.speed=speed;
9.     System.out.println(speed);
10.  }
11.  public static void main(String args[]){
12.   Bike b=new Bike(10);
13. }
14. }
```

```
Output:Vehicle is created
       10
```

3) SUPER CAN BE USED TO INVOKE PARENT CLASS METHOD.

The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below:

```
1. class Person{
2. void message(){System.out.println("welcome");}
3. }
4.
5. class Student extends Person{
6. void message(){System.out.println("welcome to java");}
7.
8. void display(){
9. message();//will invoke current class message() method
10. super.message();//will invoke parent class message() method
11. }
12.
13. public static void main(String args[]){
14. Student s=new Student();
15. s.display();
16. }
17. }
```

```
Output:welcome to java
       welcome
```

In the above example Student and Person both classes have message() method if we call message() method from Student class, it will call the message() method of Student class not of Person class because priority is given to local.

In case there is no method in subclass as parent, there is no need to use super. In the example given below message() method is invoked from Student class but Student class does not have message() method, so you can directly call message() method.

Program in case super is not required

```
1. class Person{
2. void message(){System.out.println("welcome");}
3. }
4.
5. class Student extends Person{
6.
7. void display(){
8. message();//will invoke parent class message() method
9. }
10.
11. public static void main(String args[]){
12. Student s=new Student();
13. s.display();
14. }
15. }
```

```
Output:welcome
```

1.3. FINAL KEYWORD IN JAVA

The **final keyword** in java is used to restrict the user. The final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) FINAL VARIABLE

If you make any variable as final, you cannot change the value of final variable(It will be constant).

EXAMPLE OF FINAL VARIABLE

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1. class Bike{
2.  final int speedlimit=90;//final variable
3.  void run(){
4.   speedlimit=400;
5.  }
6.  public static void main(String args[]){
```

```
 Bike obj=new  Bike();
 obj.run();
 }
}//end of class
```

```
Output:Compile Time Error
```

2) FINAL METHOD

If you make any method as final, you cannot override it.

EXAMPLE OF FINAL METHOD

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

```
Output:Compile Time Error
```

3) FINAL CLASS

If you make any class as final, you cannot extend it.

EXAMPLE OF FINAL CLASS

```
final class Bike{}

class Honda extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda honda= new Honda();
  honda.run();
  }
}
```

```
Output:Compile Time Error
```

Q) IS FINAL METHOD INHERITED?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1. class Bike{
2.   final void run(){System.out.println("running...");}
3. }
3. class Honda extends Bike{
5.    public static void main(String args[]){
6.     new Honda().run();
7.    }
8. }
```

```
Output:running...
```

Q) WHAT IS BLANK OR UNINITIALIZED FINAL VARIABLE?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

EXAMPLE OF BLANK FINAL VARIABLE

```
1. class Student{
2. int id;
3. String name;
4. final String PAN_CARD_NUMBER;
5. ...
6. }
```

QUE) CAN WE INITIALIZE BLANK FINAL VARIABLE?

Yes, but only in constructor. For example:

```
1. class Bike{
2.   final int speedlimit;//blank final variable
3.
4.   Bike(){
5.   speedlimit=70;
6.   System.out.println(speedlimit);
7.   }
```

```
8.
9.   public static void main(String args[]){
10.    new Bike();
11. }
12.}
```

```
Output:70
```

---

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

EXAMPLE OF STATIC BLANK FINAL VARIABLE

```
1.  class A{
2.    static final int data;//static blank final variable
3.    static{ data=50;}
4.    public static void main(String args[]){
5.      System.out.println(A.data);
6.   }
7.  }
```

Q) WHAT IS FINAL PARAMETER?

If you declare any parameter as final, you cannot change the value of it.

```
1.  class Bike{
2.    int cube(final int n){
3.     n=n+2;//can't be changed as n is final
4.     n*n*n;
5.    }
6.    public static void main(String args[]){
7.      Bike b=new Bike();
8.      b.cube(5);
9.   }
10.}
```

```
Output:Compile Time Error
```

1.4. ABSTRACT CLASS IN JAVA

A class that is declared with abstract keyword, is known as abstract class in java. Before learning java abstract class, let's understand the abstraction first.

ABSTRACTION

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

ABSTRACT CLASS IN JAVA

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

```
1. abstract class A{}
```

ABSTRACT METHOD

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

```
1. abstract void printStatus();//no body and abstract
```

EXAMPLE OF ABSTRACT CLASS THAT HAVE ABSTRACT METHOD

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
1. abstract class Bike{
2.   abstract void run();
3. }
4.
5. class Honda extends Bike{
6. void run(){System.out.println("running safely..");}
7.
8. public static void main(String args[]){
9.  Bike obj = new Honda();
10. obj.run();
11.}
12.}
```

```
running safely..
```

UNDERSTANDING THE REAL SCENARIO OF ABSTRACT CLASS

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.

A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: Test.java

```
1. abstract class Shape{
2. abstract void draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. class Rectangle extends Shape{
6. void draw(){System.out.println("drawing rectangle");}
7. }
8.
9. class Circle extends Shape{
10.void draw(){System.out.println("drawing circle");}
11.}
12.
13.//In real scenario, method is called by programmer or user
14.class Test{
15.public static void main(String args[]){
```

```
16. Shape s=new Circle();//In real scenario, object is provided through method e.g. getShape( ) method
17. s.draw();
18. }
19. }
```

```
drawing circle
```

ANOTHER EXAMPLE OF ABSTRACT CLASS IN JAVA

File: TestBank.java

```
1.  abstract class Bank{
2. abstract int getRateOfInterest();
3. }
4.
5. class SBI extends Bank{
6. int getRateOfInterest(){return 7;}
7. }
8. class PNB extends Bank{
9. int getRateOfInterest(){return 7;}
10. }
11.
12. class TestBank{
13. public static void main(String args[]){
14. Bank b=new SBI();//if object is PNB, method of PNB will be invoked
15. int interest=b.getRateOfInterest();
16. System.out.println("Rate of Interest is: "+interest+" %");
17. }}
```

```
Rate of Interest is: 7 %
```

ABSTRACT CLASS HAVING CONSTRUCTOR, DATA MEMBER, METHODS ETC.

```
1. //example of abstract class that have method body
2.  abstract class Bike{
3.    abstract void run();
4.    void changeGear(){System.out.println("gear changed");}
5.  }
6.
7.  class Honda extends Bike{
8.  void run(){System.out.println("running safely..");}
9.
10. public static void main(String args[]){
11.  Bike obj = new Honda();
12.  obj.run();
13.  obj.changeGear();
14. }
15. }
```

```
        running safely..
        gear changed
```

```
1. //example of abstract class having constructor, field and method
2. abstract class Bike
3. {
4.  int limit=30;
5.  Bike(){System.out.println("constructor is invoked");}
6.  void getDetails(){System.out.println("it has two wheels");}
7.  abstract void run();
8. }
9.
10. class Honda extends Bike{
11.  void run(){System.out.println("running safely..");}
12.
13.  public static void main(String args[]){
14.   Bike obj = new Honda();
15.   obj.run();
16.   obj.getDetails();
17.   System.out.println(obj.limit);
18.  }
19. }
```

```
constructor is invoked
running safely..
it has two wheels
30
```

```
1. class Bike{
2. abstract void run();
3. }
```

```
compile time error
```

ANOTHER REAL SCENARIO OF ABSTRACT CLASS

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
1. interface A{
2. void a();
3. void b();
```

```
4.  void c();
5.  void d();
6.  }
7.
8.  abstract class B implements A{
9.  public void c(){System.out.println("I am C");}
10. }
11.
12. class M extends B{
13. public void a(){System.out.println("I am a");}
14. public void b(){System.out.println("I am b");}
15. public void d(){System.out.println("I am d");}
16. }
17.
18. class Test{
19. public static void main(String args[]){
20. A a=new M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}
```

```
Output:I am a
       I am b
       I am c
       I am d
```

1.5. INTERFACE IN JAVA

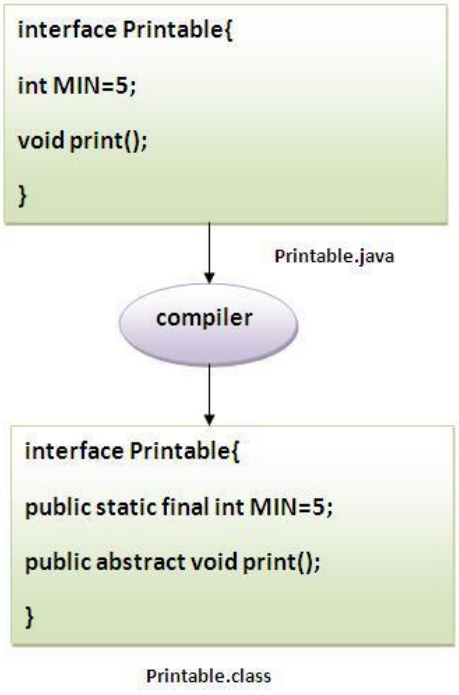
An **interface** is a blueprint of a class. It has static constants and abstract methods. The interface is **a mechanism to achieve fully abstraction** in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java. Interface also **represents IS-A relationship**. It cannot be instantiated just like abstract class.

WHY USE INTERFACE?

There are mainly three reasons to use interface. They are given below.

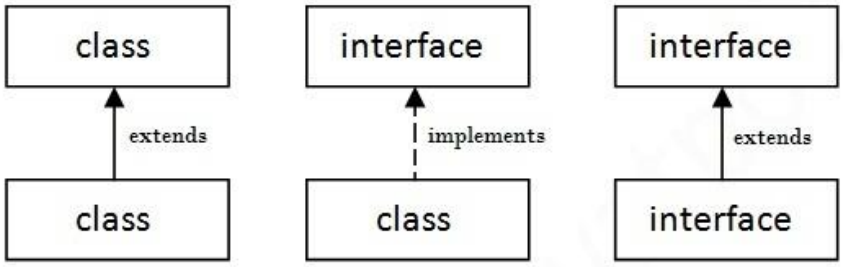
- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.

In other words, Interface fields are public, static and final bydefault, and methods are public and abstract.



Understanding 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**.



SIMPLE EXAMPLE OF INTERFACE

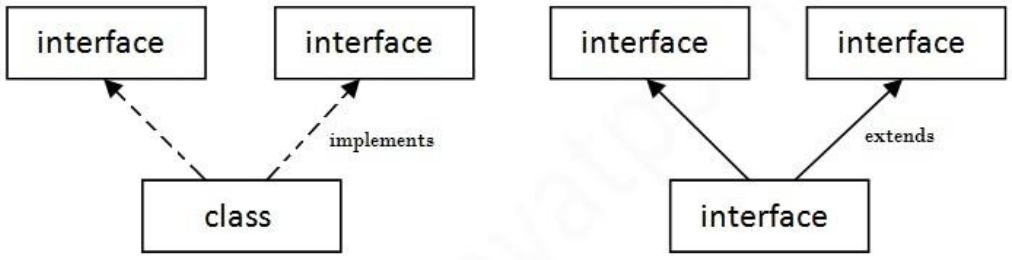
In this example, Printable interface have only one method, its implementation is provided in the A class.

```
1. interface printable{
2. void print();
3. }
4.
5. class A implements printable{
6. public void print(){System.out.println("Hello");}
7.
8. public static void main(String args[]){
9. A obj = new A();
10. obj.print();
11. }
12. }
```

```
Output:Hello
```

MULTIPLE INHERITANCE IN JAVA BY INTERFACE

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```
1. interface Printable{
2. void print();
3. }
4.
5. interface Showable{
6. void show();
7. }
8.
9. class A implements Printable,Showable{
10.
11. public void print(){System.out.println("Hello");}
12. public void show(){System.out.println("Welcome");}
13.
14. public static void main(String args[]){
15. A obj = new A();
16. obj.print();
17. obj.show();
18.  }
19. }
```

```
Output:Hello
       Welcome
```

Q) MULTIPLE INHERITANCE IS NOT SUPPORTED IN CASE OF CLASS BUT IT IS SUPPORTED IN CASE OF INTERFACE, WHY?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface because there is no ambiguity as

implementation is provided by the implementation class. For example:

```
1. interface Printable{
2. void print();
3. }
4.
5. interface Showable{
6. void print();
7. }
8.
9. class A implements Printable,Showable{
10.
11. public void print(){System.out.println("Hello");}
12.
13. public static void main(String args[]){
14. A obj = new A();
15. obj.print();
16.  }
17. }
```

```
Output:Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class A, so there is no ambiguity.

```
1. interface Printable{
2. void print();
3. }
4.
5. interface Showable extends Printable{
6. void show();
7. }
8.
9. class A implements Showable{
10.
11. public void print(){System.out.println("Hello");}
12. public void show(){System.out.println("Welcome");}
13.
14. public static void main(String args[]){
15. A obj = new A();
16. obj.print();
17. obj.show();
18.  }
19. }
```

```
Output:Hello
       Welcome
```

1.6. MARKER OR TAGGED INTERFACE

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
1. //How Serializable interface is written?
2.
3. public interface Serializable{
4. }
```

Nested Interface

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

```
1. interface printable{
2.  void print();
3.  interface MessagePrintable{
4.    void msg();
5.  }
6. }
```

NESTED INTERFACE

An interface which is declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

POINTS TO REMEMBER FOR NESTED INTERFACES

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitely.

SYNTAX OF NESTED INTERFACE WHICH IS DECLARED WITHIN THE INTERFACE

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

SYNTAX OF NESTED INTERFACE WHICH IS DECLARED WITHIN THE CLASS

```
class class_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

EXAMPLE OF NESTED INTERFACE WHICH IS DECLARED WITHIN THE INTERFACE

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}

class Test implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

public static void main(String args[]){
 Showable.Message message=new Test();//upcasting here
 message.msg();
}
}
```

```
Output:hello nested interface
```

As you can see in the above example, we are acessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection frameword, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

INTERNAL CODE GENERATED BY THE JAVA COMPILER FOR NESTED INTERFACE MESSAGE

The java compiler internally creates public and static interface as displayed below:.

```
1. public static interface Showable$Message
2. {
3.   public abstract void msg();
4. }
```

EXAMPLE OF NESTED INTERFACE WHICH IS DECLARED WITHIN THE CLASS

Let's see how can we define an interface inside the class and how can we access it.

```
1. class A{
2.   interface Message{
3.    void msg();
4.   }
5. }
6.
7. class Test implements A.Message{
8.  public void msg(){System.out.println("Hello nested interface");}
9.
10. public static void main(String args[]){
11.  A.Message message=new Test();//upcasting here
12.  message.msg();
13. }
14. }
```

```
Output:hello nested interface
```

1.7. OBJECT CLONING IN JAVA

The **object cloning** is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object. The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates**CloneNotSupportedException**. The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

```
1. protected Object clone() throws CloneNotSupportedException
```

WHY USE CLONE() METHOD ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing to be performed that is why we use object cloning.

ADVANTAGE OF OBJECT CLONING

Less processing task.

EXAMPLE OF CLONE() METHOD (OBJECT CLONING)

Let's see the simple example of object cloning

```
1. class Student implements Cloneable{
2. int rollno;
3. String name;
4.
5. Student(int rollno,String name){
6. this.rollno=rollno;
7. this.name=name;
8. }
9.
10. public Object clone()throws CloneNotSupportedException{
11. return super.clone();
12. }
13.
14. public static void main(String args[]){
15. try{
16. Student s1=new Student(101,"amit");
17.
18. Student s2=(Student)s1.clone();
19.
```

```
System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}

}
}
```

```
Output:101 amit
       101 amit
```

As you can see in the above example, both reference variables have the same value. Thus, the clone() copies the values of an object to another. So we don't need to write explicit code to copy the value of an object to another.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.