static member: or

static keyword:

The **static keyword** in is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

1. static Variable (also known as a class variable)
2. static Method (also known as a class method)
3. static Block
4. static Nested class

1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

```
class Student{
    int rollno;
    String name;
    String college="Accurate";
}
```

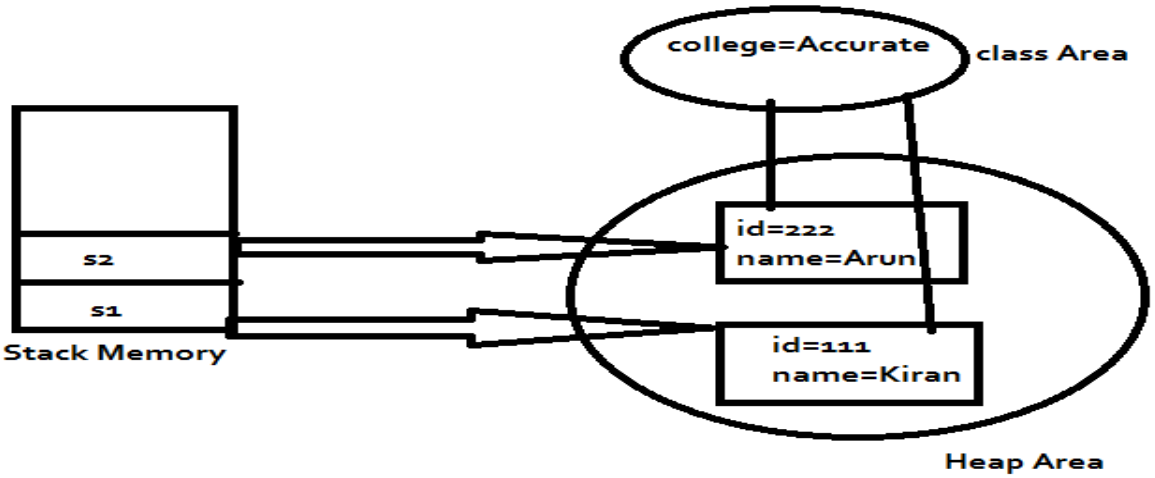
Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Example of static variable

```
class Student{
  int rollno;//instance variable
  String name;
  static String college ="Accurate";//static variable
  //constructor
  Student(int r, String n){
  rollno = r;
  name = n;
  }
  //method to display the values
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
```

```
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Kiran");
 Student s2 = new Student(222,"Arun");
 //we can change the college of all objects by the single line of code
 //Student.college="AIMT";
 s1.display();
 s2.display();
 }
}
```

Output:
111 Kiran Accurate
222 Arun Accurate



2) static method:

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
class Student{
    int rollno;
    String name;
    static String college = "Accurate";
    //static method to change the value of static variable
    static void change(){
    college = "AIMT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
   //creating objects
```

```
    Student s1 = new Student(111,"Kiran");
    Student s2 = new Student(222,"Arun");
    Student s3 = new Student(333,"Rohit");
    //calling display method
    s1.display();
    s2.display();
    s3.display();
    }
```

Output:
111 Kiran AIMT
222 Arun AIMT
333 Rohit AIMT

Restrictions for the static method

There are two main restrictions for the static method. They are:

1.The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A{
 int a=40;//non static

 public static void main(String args[]){
  System.out.println(a);
 }
}
```

Output:Compile Time Error

Q) Why is the Java main method static?

Ans) It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main() method that will lead the problem of extra memory allocation.

---

3) static block:

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

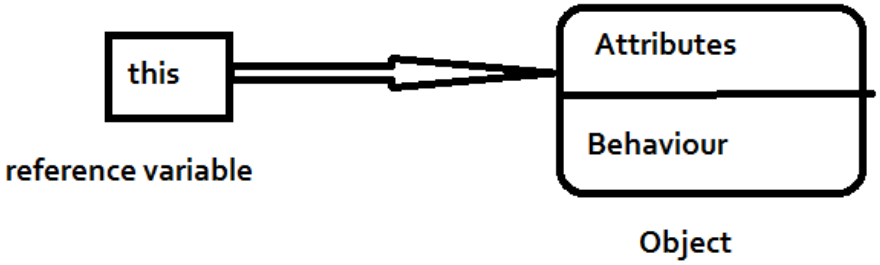
Example of static block

```
class A2{
  static{
System.out.println("static block is invoked");
}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Output:static block is invoked
Hello main

this keyword:

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.



uses of this keyword:
1. this can be used to refer current class instance variable.
2.this can be used to invoke current class method(implicitly).
3.this can be used to invoke current class constructor.

1.) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class ThisEx{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
0 null 0.0
```

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
```

```
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
 class  ThisEx1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000
112 sumit 6000
```

If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

```
class Student{
int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class ThisEx2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

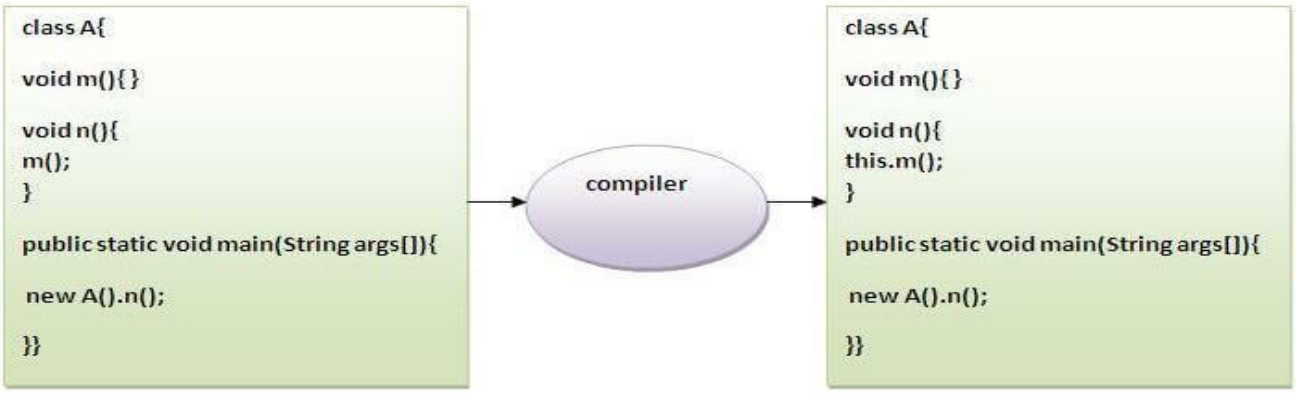
Output:

```
111 ankit 5000
112 sumit 6000
```

Note: *It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.*

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. For example:




```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

Output:

```
hello n
hello m
```

3) this() : to invoke current class constructor

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

Output:

```
hello a
10
```

Calling parameterized constructor from default constructor:

```
class A{
A(){
this(5);
```

```
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

Output:

```
5
hello a
```

Real usage of this() constructor call

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. For example.

```
class Student{
int rollno;
String name,course;
float fee;
Student(int rollno,String name,String course){
this.rollno=rollno;
this.name=name;
this.course=course;
}
Student(int rollno,String name,String course,float fee){
this(rollno,name,course);//reusing constructor
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
}
class    RealThis{
public static void main(String args[]){
Student s1=new Student(111,"ankit","java");
Student s2=new Student(112,"sumit","java",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit java null
112 sumit java 6000
```

Note: Call to this() current class constructor must be the first statement in constructor. Otherwise there will be compile time error.

Access Specifier in Java:

The access specifier or access modifier in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access Specifier:

1. **private**: The access level of a private access specifer is only within the class. It cannot be accessed from outside the class.

   Ex: class Student{

   private int rollno;

   private int dob;

   }

2. **default**: The access level of a default access specifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

   Ex: class Student{

   int rollno;//no need to write default in front of variable.

   int dob;

   }

3. **protected**: The access level of a protected access specifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

   Ex: class Student{

   protected int rollno;

   proctected   int dob;

   }

4. **public**: The access level of a public access specifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

   Ex: class Student{

   public   int rollno;

   public   int dob;

   }

Access Specifier	within class	within package	outside the package by subclass only	out side the package
private	yes	No	NO	No
default	yes	yes	No	No
protected	yes	yes	yes	No
public	yes	yes	yes	yes

Inheritance :

Inheritance a mechanism in which one object acquires all the properties and behaviors of a parent object.

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Advantages of inheritance :

- **For Code Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.
- For Method Overriding (so runtime polymorphism can be achieved).

Terms used in Inheritance

Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

Types of inheritance in java:

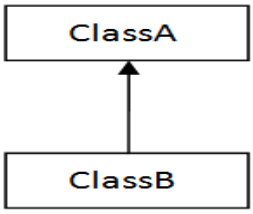
1.Single Inheritance

2.Multilevel Inheritance

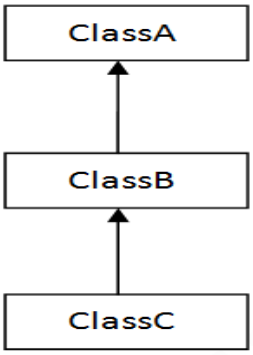
3.Hierarchical Inheritance

4.Multiple Inheritance

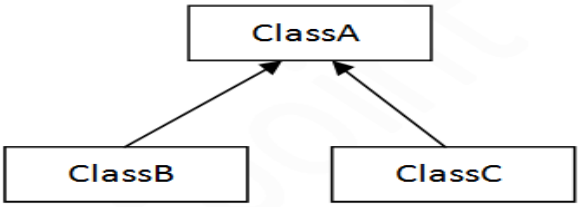
5.Hybrid Inheritance.



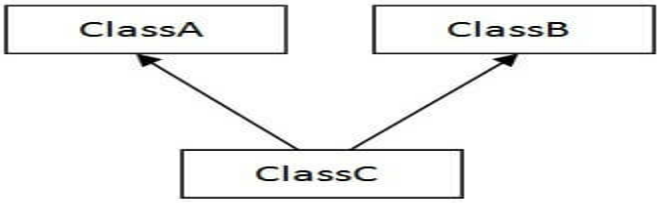
1) Single



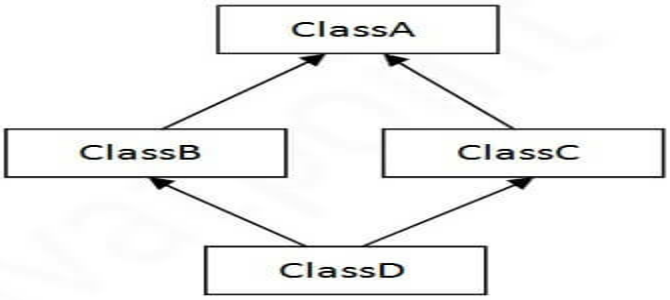
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

.

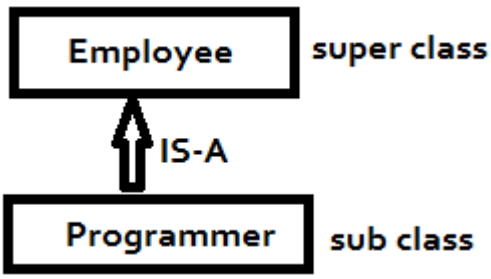
Note: Multiple inheritance is not supported in Java through class. And if there is Multiple inheritance involved in Hybrid inheritance then it is also not supported in java java through class. We use interface to solve Multiple inheritance problems.

The syntax of Java Inheritance:

```
class Subclass-name extends Superclass-name
{
  //methods and variables
}
```

Single Inheritance:

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Programmer class inherits the Employee class, so there is the single inheritance.

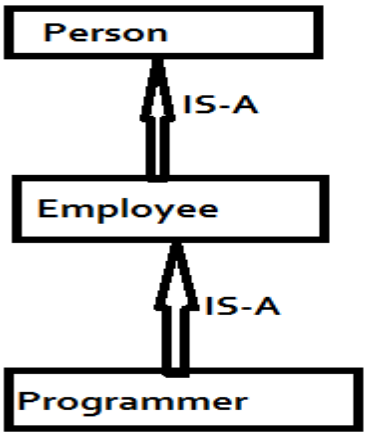



```
class  Employee {
        int empid;
```

```
    int salary;
    void show(){
    System.out.println(empid+" "+name+" "+salary);
    }

}
class Programmer extends Employee{
    int bonous;
void display(){
    System.out.println(empid+" "+name+" "+salary+" "+bonous+" "+age);

}
}
class  SingleEx{
public static void main(String args[]){
    Employee e=new Employee();
    e.empid=101;  e.salary=60000;      e.name="Mohit"; //e.bonous=1000;
    e.show();
    e.display();
    Programmer p=new Programmer();
    p.empid=100;  p.salary=40000;      p.name="Rohit";      p.bonous=2000; p.age=20;
    p.show();
    p.display();
    }
}
```

Multilevel Inheritance:

When there is a chain of inheritance, it is known as *multilevel inheritance*. for example given below, Programmer class inherits the Employee class which again inherits the Person class, so there is a multilevel inheritance.




```
class Person{

}
class Employee extends Person{

}
class Programmer extends Employee{

}
```

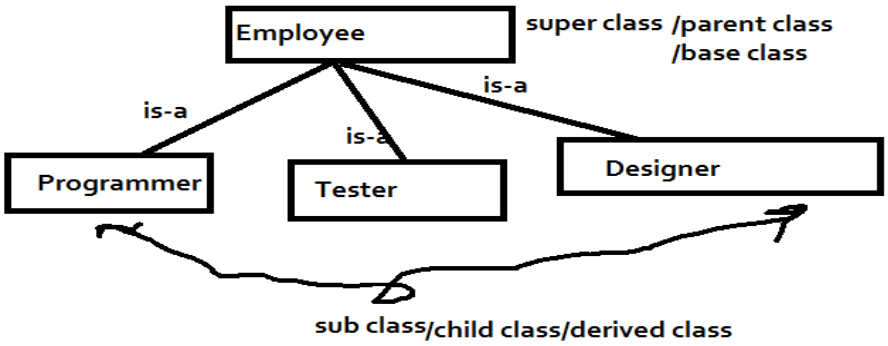
Ex: //Multilevel Inheritance

```
class Person{
    String name;
    int age;
}
class  Employee extends Person{
    int empid;
     int salary;
    void show(){
    System.out.println(empid+" "+name+" "+salary);
    }

}
class Programmer extends Employee{
```

```
        int bonous;
void display(){
        System.out.println(empid+" "+name+" "+salary+" "+bonous+" "+age);

}
}
class MultilevelEx{
public static void main(String args[]){
        //Employee e=new Employee();
        //e.empid=101;          e.salary=60000;         e.name="Mohit"; //e.bonous=1000;
        //e.show();
        //e.display();
        Programmer p=new Programmer();
        p.empid=100;  p.salary=40000;         p.name="Rohit";         p.bonous=2000; p.age=20;
        p.show();
        p.display();
        }
}
```

Hierarchical Inheritance:

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Programmer ,Tester and Designer classes inherits the Employee class, so there is hierarchical inheritance.




```
class Employee{

}
class Programmer extends Employee{

}
class Tester extends     Employee{

}
class Designer extends Designer{

}
```

```
class Employee{
 void getSalary(){
        System.out.println("We cant tell the employee salary");
 }
    }
class Programmer extends Employee{
void progSalary(){
        int salary=60000;
        System.out.println("Programmer salary:"+salary);

}
            }
class Tester extends Employee{
void testSalary(){
        int salary=50000;
        System.out.println("Tester salary:"+salary);

}
}
```

```
class Designer extends Employee{
void DesigSalary(){
        int salary=90000;
        System.out.println("Designer salary:"+salary);
}
}
class HierarchicalEx{
public static void main(String args[]){
Programmer p=new Programmer();
p.progSalary();
p.getSalary();
Tester t=new Tester();
t.testSalary();
t.getSalary();
Designer d=new Designer();
d.DesigSalary();
d.getSalary();
}}
```

Q) Why multiple inheritance is not supported in java?

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

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

Example:

```
class A{
void display(){System.out.println("Hello");}
}
class B{
void display(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.display();//Now which display() method would be invoked?
}
}
```

Output:
Compile Time Error

Super Keyword in java:

super keyword in java

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

There is Threee uses of super keyword in java:--

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

1) super is used to refer immediate super class instance variable:

Problem without super keyword

```
class Test{
  int s=50;
            }
class  SuperVariable  extends Test{
  int s=100;
  void display(){
   System.out.println(s);//will print variable  subclass   which is s=100 since it hide the super class variable s=50
  }
  public static void main(String args[]){
   SuperVariable   ob=new     SuperVarialbe();
   ob.display();
}
}
```

Output:100

In the above example Test and Test1 both class have a common variables. Instance variable of super class is hide by the instance variable of subclass, to solve the above problem we use super keyword to distinguish between parent class instance variable and current class instance variable.

Solution by super keyword

```
//example of super keyword

class Test{
  int s=50;
}
class Test1 extends Test{
  int s=100;
  void display(){
   System.out.println(super.s);  //super refer to parent class variable s will print s of Test
   System.out.println(s);    //will print s of Test Test1
  }
  public static void main(String args[]){
   Test1 t=new Test1();
   t.display();
    }
}
```

Output: 50
100

Note:- here super refer to parent class instance variable.

2) super can be used to invoke super class method:

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

Ex:

```
class Employee{
  int salary=60000;
  void show(){                    //super class method
          System.out.println("The salary of  Employee:"+salary);

  }
  }
  class Developer extends Employee{
   int salary=50000;
   void show(){                //sub class method which hide super class method
            System.out.println("The salary of Developer:"+salary);
```

```
        super.show();  //use super keyword to call super class method.

    }
        }
class SuperMethod{
public static void main(String args[]){
Developer ob=new Developer();
ob.show();
}
}
```

Output:
The salary of Developer:50000
The salary of Employee:60000

3) super is used to invoke super class constructor.
The super keyword can also be used to invoke the super class constructor as given below:

Ex:

```
class Person{
int id;
String name;
Person(int id,String name){          //super class constructor
this.id=id;
this.name=name;
}
}
class Employee extends Person{
double salary;
Employee(int id,String name, double salary){
super(id,name);                  //super class constructor is called
this.salary=salary;
}
void show(){
System.out.println(id+" "+name+" "+salary);
}
}
class SuperCons{
        public static void main(String args[]){
                Employee ob=new Employee(100,"Shubham", 45000.500);
                ob.show();
        }

}
```

Output:
100 Shubham 45000.500

Polymorphism in java:

Polymorphism is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java:

1.compile-time polymorphism :

2. runtime polymorphism.

1.Static polymorphism or compile-time polymorphism:

The process of binding the overloaded method within object at compile time is known as **Static polymorphism** due to static polymorphism utilization of resources (main memory space) is poor because for each and every overloaded method a memory space is created at compile time when it binds with an object.

method overloading is example of Compile time polymorphism or static polymorphism.

Method Overloading:

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

Advantage of method overloading:

Method overloading *increases the readability of the program*.

Different ways to overload the method:

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1.By changing number of arguments:

```
//1.By differnt number of arguments.
class MethodOverloading
{
void add(int a,int b){
int c=a+b;
System.out.println("The sum 0f two numbers:"+c);
}
void add(int a,int b,int c){
int d=a+b+c;
System.out.println("The sum 0f three numbers:"+d);
}
public static void main(String args[]){
MethodOverloading ob=new MethodOverloading();
ob.add(10,20);
ob.add(10,20,30);
}}
```

Output:
The sum of two numbers:30
The sum of three numbers:60

2.By changing the data type:

Ex:

```
class MethodOverDataType
{
void add(int a,int b){
int c=a+b;
System.out.println("The sum 0f two numbers:"+c);
}
float add(float a,int b){
float d=a+b;
return d;
}
public static void main(String args[]){
```

```
MethodOverDataType ob=new MethodOverDataType();
ob.add(10,20);
float f=ob.add(10.5f,5);
System.out.println("The sum 0f two numbers:"+f);
}}
```

Output:

```
The sum of two numbers:30
The sum of two numbers:15.5
```

2. runtime polymorphism:

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

Ex: Method overriding is the example of runtime or dynamic polymorphism.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

```
class A{}
class B extends A{}
A a=new B();//upcasting
```

For upcasting, we can use the reference variable of class type or an interface type.

Method Overriding:

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

Advantages of Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism.

Ex:

```
class Employee{
void getSalary(){  //overridden method
System.out.println("we can't tell the emp salary");
}
}
class Developer extends Employee{
int salary=70000;
void getSalary(){    //Method Overriding
System.out.println("Developer salary is:"+salary);
}
}
class Tester extends Employee{
int salary=50000;
void getSalary(){           //Method Overriding
System.out.println("Tester salary is:"+salary);
}
```

```
}

class OverridingEx{
public static void main(String args[]){
                                        Employee e;
                                        Developer d=new Developer();
                                e=d;//upcasting
                                e.getSalary();
                                Tester    t=new Tester();
                                    e=t;
                                e.getSalary();
                                        }
                                        }
```

Output:
Developer salary:70000
Tester salary:50000

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

1. variable
2. method
3. class

1) final variable

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

Example:

```
class FinalVariable{
          final int A=10; //Constant varialbe
  void show(){
          A=50; //eroor can not change final varialbe value
System.out.println(a);
}
public static void main(String args[]){
FinalVariable ob=new FinalMethod();
ob.show();
}
}
```

Output:
Compile Time Error

2) final method

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

```
Ex:     class FinalMethod{
          int a=10;
         final void show(){  //final method

System.out.println(a);
}
}
class FinalMethodEx extends FinalMethod{
          int b=40;
          void show(){        //Error Final Method can not override
                    System.out.println(b);
```

```
        }

public static void main(String args[]){
FinalMethodEx ob=new FinalMethodEx();
ob.show();
}
}
```
Output:
Compile Time Error

3) final class

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

```
final class FinalClass{  //final class
        int a=10;
  void show(){

System.out.println(a);
}
}
class FinalClassEx extends FinalClass{    //Error Final class cannot be inherited
        int b=40;
        void show(){
                System.out.println(b);

        }
public static void main(String args[]){
FinalClassEx ob=new FinalClassEx();
ob.show();
}
}
```
Output:
Compile Time Error

Abstract class :

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

- It can have **constructors** and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.
- object of abstract class cannot be created.

Example: abstract class A{

//abstract method and non abstract method

}

Abstract Method: A method which is declared as abstract and does not have implementation is known as an abstract method.

Ex: abstract void display();//no method body and abstract

```
Ex:  abstract class Employee{
        String salary="we can't salary say about Employee salary.";
 abstract void getSalary();//abstract method
```

```
 void display(){//non-abstract method
 System.out.println(salary);
}
}
class Developer extends Employee{
int salary=70000;
void getSalary(){
System.out.println("Developer salary is:"+salary);
}
}
class Tester extends Employee{
int salary=50000;
void getSalary(){
System.out.println("Tester salary is:"+salary);
}
}
class AbstractEx{
public static void main(String args[]){
                        Employee e;
                        Developer d=new Developer();
                        e=d;//upcasting
                        e.getSalary();
                        Tester    t=new Tester();
                        e=t;
                        e.getSalary();
                        e.display();
                                }
                                }
```

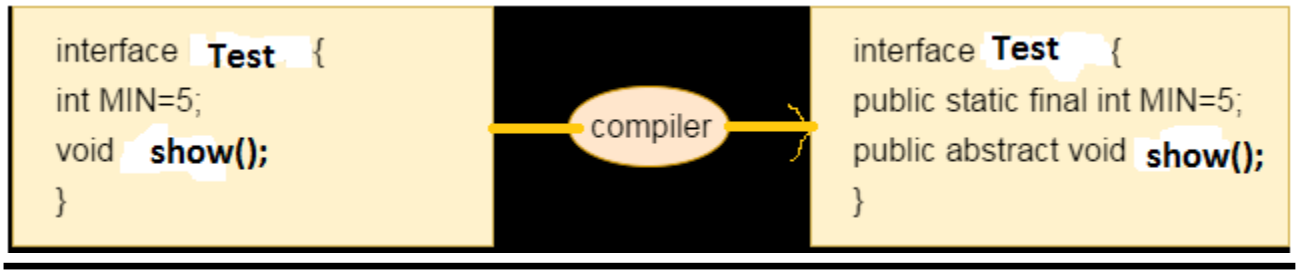
Output:
Developer salary:70000
Tester salary:50000
We can't say about Employee salary

Note1: If there is an abstract method in a class, that class must be abstract.

Note2: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Interface:

=>An interface is like a class which is used for obtaining fully abstraction and multiple inheritance in java.
=> interface contains only public static final variable and public abstract method.
=>we cannot create an object of interface.
=>The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before variables if we do not provide.

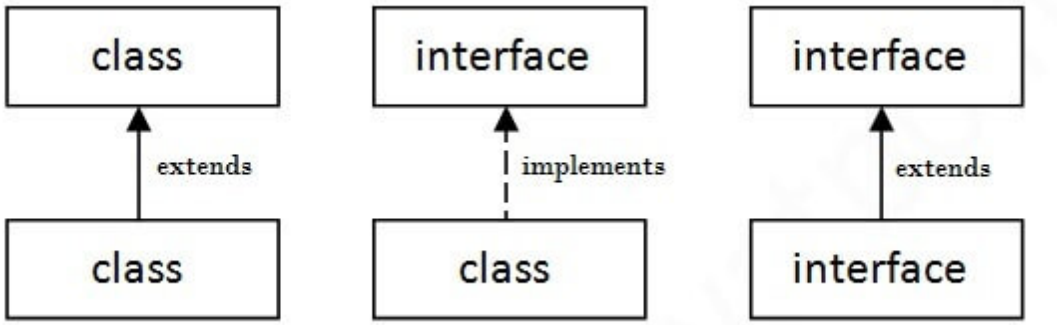


Syntax:

```
interface <interface_name>{

   // declare constant variable
   // declare  abstract   method
   //
}
```

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**.



Interface Example

```
interface Employee{
void salary();
static void show(){
        System.out.println("Default method in interface");
}
}
class Developer implements Employee{
public void salary(){
int salary=80000;
System.out.println("Salary of Employee:"+salary +" ");
}
}
class InterfaceEx{
public static void  main(String args[]){
Employee ob;
Developer d=new Developer();
ob=d;
ob.salary();
}
}
```

extending interfaces:

A class implements an interface, but one interface extends another interface.

Example:

```
interface Company{
void display();
}
interface Employee extends Company{
void show();
}
class Developer implements Employe{
public void display(){
System.out.println("Company name is ABC");
```

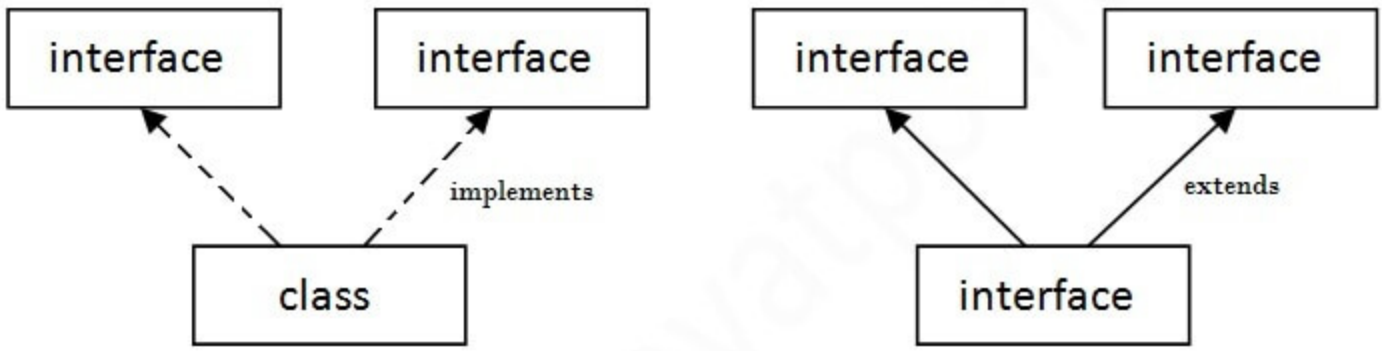
```
}
public void show(){
System.out.println("Welcome Employee);
}

public static void main(String args[]){
Developer obj = new Developer();
obj.display();
obj.show();
 }
}
```

Output:

```
Company name is ABC
Welcome Employee
```

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 of Multiple inheritance in java

```
interface Employee{
void salary();
}
interface ABC{
        void getName();
}
class Developer implements Employee,ABC{
public void salary(){
int salary=80000;
System.out.println("Salary of Employee:"+salary +" ");
}
public void getName(){
        String name="Alok";
        System.out.println("Name of Employee:"+name);
}
}
class MultiInterface{
public static void  main(String args[]){
Employee ob;
ABC a;
Developer d=new Developer();
```

```
ob=d;
ob.salary();
Employee.show();
a=d;
a.getName();
}
}
```

Difference between abstract class and interface:

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	1).Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	2).Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	3).Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	4).Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	5).The **interface keyword** is used to declare interface.

7) Example:

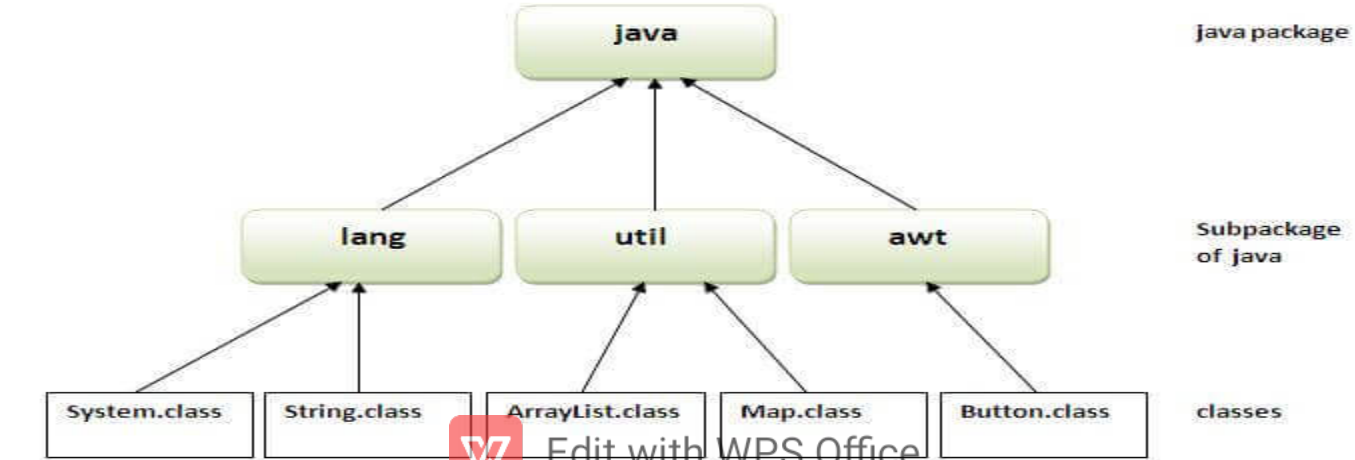
```
abstract class Employee{
 abstract void salary();
}
class Developer extends Employee{
public void salary(){
int salary;
System.out.println("Salary of Employee:"+salary)
}
 public static void main(String args[]){
Employee e;
Developer d=new Developer();
e =d;
e.salary();
}
}
```

7).Example:

```
      interface  Employee{
        void salary();
      }
class Developer implements Employee{
public void salary(){
int salary;
System.out.println("Salary of Employee:"+salary)
}
public static void main(String args[]){
Employee e;
Developer d=new Developer();
e =d;
e.salary();
}
}
```

Package

=>A **package** is a group of similar types of classes, interfaces and sub-packages.
=>Package can be categorized in two form.
    1. built-in package   2. user-defined package.
=>There are many built-in packages such as java, lang, awt, io, applet, swing, bean, net, util, sql etc. Top most package of every java package is java.



Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision. i.e two different package can have a same name class.

Note:-If we do not make any package in a program then that program contains a default package.

User-defined package:--

For making user defined package we use package keyword.

Simple example of java user-defined package:

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }
}
```

How to compile java package:

javac -d directory javafilename

For example

javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name d:\abc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output: Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access user defined package from another package:

There are three ways to access the package from outside the package.

1. import packagename.*;
2. import packagename.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible. Since this way of using package loads all classes and interfaces in memory so it is not memory efficient approach.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
  package pack;
 public class A{
  public  void msg(){
System.out.println("Hello");
}
}
```

```
//save by B.java
package mypack;
import pack.*;
  class B{
  public static void main(String args[]){
  A obj = new A();
  obj.msg();
  }
}
```
Output:Hello

2) Using packagename.classname:

If you import package.classname then only declared class of this package will be accessible. In this way only that particular class or interface is loaded in memory which we want so this method is more memory efficient.

Example of package by import package.classname

```
//save by A.java
package pack;
public class A{
  public void msg(){
System.out.println("Hello");
}
}
```

```
//save by B.java
package mypack;
import pack.A;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```
Output:Hello

3) Using fully qualified name:

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
  package pack;
public class A{
public void msg(){
System.out.println("Hello");
}}
```

```
//save by B.java
  package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```
Output:Hello

Subpackage in java:

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

For example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

Note: The standard of defining package is domain.company.package e.g. com.abc.p1 or org.xyz.p2

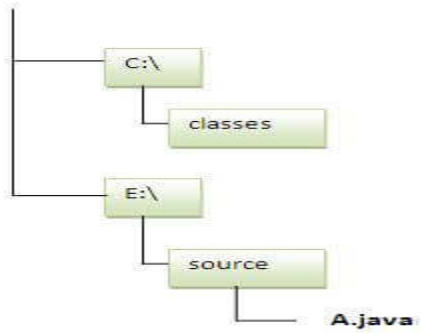
Example of Subpackage:

```
package com.abc.p1;
   class SubPack{
     public static void main(String args[]){
      System.out.println("Hello from subpackage");
     }
   }
```

To Compile: javac -d . SubPack.java
To Run: java com.abc.p1.SubPack
Output:Hello from subpackage

CLASSPATH Setting for Packages:

We use classpath setting for packages when we have to send the class file to another directory or drive?
There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




```
//save as Simple.java
 Package packnext;
 public class NextDrive{
  public static void main(String args[]){
     System.out.println("Welcome to package");
    }
```

To compile:

C:\javaprograms> javac -d G:\javafile NextDrive.java
To Run:
To run this program from C:\javaprograms directory, you need to set classpath of the directory where the class file resides.
C:\javaprograms> set classpath=G:\javafile;.;
C:\javaprograms> java packnext.NextDrive

Static import in Java

In Java, static import concept is introduced in 1.5 version. With the help of static import, we can access the static members of a class directly without class name or any object. For Example: we always use sqrt() method of Math

class by using Math class i.e. **Math.sqrt()**, but by using static import we can access sqrt() method directly. According to SUN microSystem, it will improve the code readability and enhance coding. But according to the programming experts, it will lead to confusion and not good for programming. If there is no specific requirement then we should **not** go for static import.

```
class StaticImp {
   public static void main(String[] args)
   {
      System.out.println(Math.sqrt(4));
      System.out.println(Math.pow(2, 2));
   }
}
```

Output:

```
2.0
4.0
```

```
// Java Program to  calling of predefined methods
// with static import
import static java.lang.Math.*;
class StaticImportEX {
   public static void main(String[] args)
   {
      System.out.println(sqrt(4));
      System.out.println(pow(2, 2));
   }
}
```

Output:

```
2.0
4.0
```

```
// Java program to  calling of static member of
// System class without Class name
import static java.lang.Math.*;
import static java.lang.System.*;
class Geeks {
   public static void main(String[] args)
   {
      // We are calling static member of System class
      // directly without System class name
      out.println(sqrt(4));
      out.println(pow(2, 2));
   }
}
```

Output:

```
2.0
4.0
```

NOTE : System is a class present in java.lang package and out is a static variable present in System class. By the help of static import we are calling it without class name.

Making JAR Files for Library Packages:

A JAR (Java Archive) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform.
In simple words, a JAR file is a file that contains compressed version of .class files, audio files, image files or directories. We can imagine a .jar files as a zipped file(.zip) that is created by using WinZip software. Even , WinZip software can be used to used to extract the contents of a .jar . So you can use them for tasks such as lossless data compression, archiving, decompression, and archive unpacking.

The jar tool provides many switches, some of them are as follows:

1. **-c** creates new archive file
2. **-v** generates verbose output. It displays the included or extracted resource on the standard output.
3. **-m** includes manifest information from the given mf file.
4. **-f** specifies the archive file name
5. **-x** extracts files from the archive file
6. **–u update jar file**

Create a JAR file: To create a .jar file , we can use jar cf command in the following way:

```
jar cf jarfilename inputfiles
```

Here, cf represents create the file. For example , assuming our package pack is available in C:\directory , to convert it into a jar file into the pack.jar , we can give the command as:

```
C:\> jar cf pack.jar pack
```

Now , pack.jar file is created

Viewing a JAR file: To view the contents of .jar files, we can use the command as:

```
jar tf jarfilename
```

Here , tf represents table view of file contents. For example, to view the contents of our pack.jar file , we can give the command:

```
C:/> jar tf pack.jar
```

Now , the contents of pack.jar are displayed as:

```
META-INF/
META-INF/MANIFEST.MF
pack/
pack/class1.class
pack/class2.class
..
..
```

where class1 , class2 etc are the classes in the package pack. The first two entries represent that there is a manifest file created and added to pack.jar. The third entry represents the sub-directory with the name pack and the last two represent the files name in the directory pack.

When we create .jar files , it automatically receives the default manifest file. There can be only one manifest file in an archive , and it always has the pathname.

```
META-INF/MANIFEST.MF
```

This manifest file is useful to specify the information about other files which are packaged.

Manifest files in JAR

The manifest files typically contain metadata about the files inside the JAR. This contains package related information.

However, if a single application is bundled together in a JAR file, then the manifest file should have the definition of the main class.

You can find the manifest in the location **META-INF/MANIFEST.MF.**

Extracting a JAR file: To extract the files from a .jar file , we can use:
jar xf jarfilename

Here, xf represents extract files from the jar files. For example , to extract the contents of our pack.jar file, we can write:

C:\> jar xf pack.jar

This will create the following directories in C:\

META-INF
pack // in this directory , we can see class1.class and class2.class.

Updating a JAR File The Jar tool provides a 'u' option which you can use to update the contents of an existing JAR file by modifying its manifest or by adding files. The basic command for adding files has this format:
jar uf jar-file input-file(s)

here uf represent update jar file. For example , to update the contents of our pack.jar file, we can write:

C:\>jar uf pack.jar

How to make an executable jar file in Java

Creating manifest file:

To create manifest file, you need to write Main-Class, then colon, then space, then classname then enter. For example:

myfile.mf

Main-Class: First

The manifest file starts with Main-Class colon space class name. Here, class name is First. Then enter.

You need to write **jar** then **swiches** then **mf_file** then name of **jar_file** then **.classfile** as given below:

jar -cvmf myfile.mf myjar.jar First.class

Running a JAR file: In order to run an application packaged as a JAR file (requires the Main-class manifest header) , following command can be used:
C:\>java -jar pack.jar

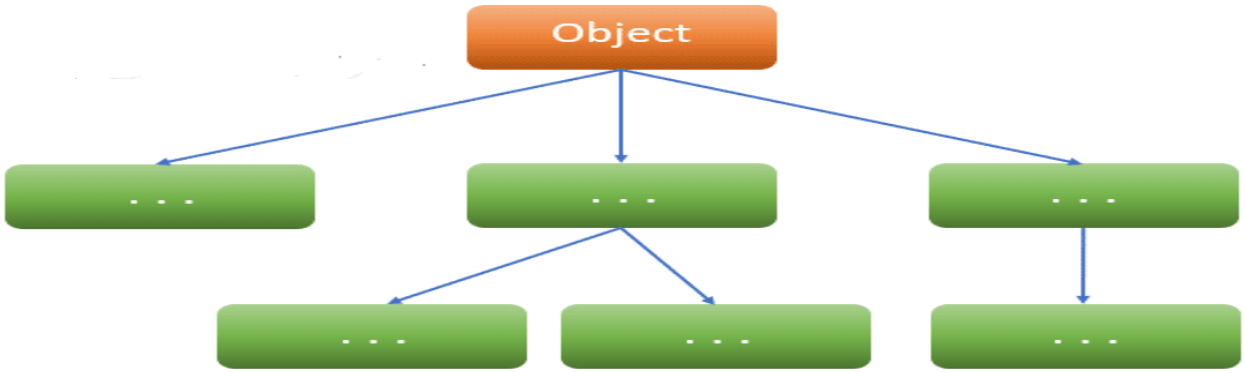
Object class:

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

Object class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

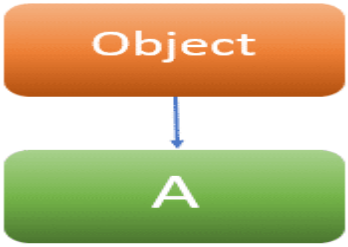
The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, known as upcasting.

The **Object class** provides some methods that extended by each class of Java. So, The Object class provides common behaviors to all the objects.



Example:

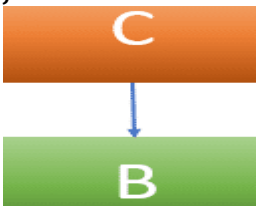
```
class A
{
//Body of class A
}
```



class A is not inheriting any class, it means it inheriting the **Object class**. Here you can see the **Object class** is directly inherited by **class A**.

Example:

```
Class C
{
//Body of class C
}
Class B extends C
{
//Body of class B
}
```



In the above example, **class B** is inheriting a **class C**, it means it is not directly inheriting the **Object class**. Here you can see the **class C** is directly inherited by **class B**. But **Object class** is indirectly inherited by **class B.** because somewhere **class C** inheriting the **Object Class**.

NOTE: All the method of Object class is available to all Java classes.

NOTE: If you want to refer any object whose type is unknown then the **Object class** is beneficial. Because **Object** is parent class so that you can refer to the child class object.

Object obj = new A();
Object obj = new B();

Methods of Object class

The Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

toString() method in Java

The **toString() method** is defined in **Object class** which is the super most class in Java. This method returns the **string** representation of the object. Let's have a look at code.

```
public String toString()
{
return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

in the above code its return type is String. It returns the class name of the object and appends the *hashCode* of that object.

As you already know every class in java is the child of **Object class**. It means each class inheriting the **toString() method** from **Object class.**

Whenever a user tries to print the object of any class then the java compiler internally invokes the **toString() method**.

Ex:

```
class ToString{
public static void main(String args[]){
Object ob=new Object();
System.out.println(ob);
}}
```

```
C:\javaprograms>java ToString
java.lang.Object@70dea4e
```

When OR Why we should Override the toString() method?

The default implementation of the **toString() method** is to return only the class name and ***hashcode*** of the object. But sometimes default implementation is not useful. In these types of situations, it is better to override the **toString() method** and return some informative data.

An example of the default implementation of **toString() method is shown below**

```
public class Student
{
   int rollNo;
   String name;
   public static void main(String[] args)
   {
      Student s1 = new Student();
      System.out.println("Object s1 = "+ s1);

      Student s2 = new Student();
      System.out.println("Object s2 = "+ s2);
   }
}
```

Output: Object s1 = Student@1e81f4dc
Object s2 = Student@5ccd43c2

In the above example, we are creating a class **Student.** In the **Student** class, we are creating and printing two objects **s1** and **s2.** When the compiler tries to print the object **s1** and **s2,** Then it invokes the **toString() method**. As **toString() method** is not defined in **Student** class then it will invoke the **toString() method** of **Object class**. The default implementation prints the **class name** and ***hashcode*** as output.

The example of override the toString() method.

```
class StudentTo{
int rollNo;
String name;
public static void main(String[] args)
{
StudentTo s1 = new StudentTo();
s1.name = "Ravi";
s1.rollNo = 1;
System.out.println("Object s1 = "+ s1);
StudentTo s2 = new StudentTo();
s2.name = "Ram";
s2.rollNo = 2;
System.out.println("Object s2 = "+ s2);
}
@Override
public String toString()
{
return this.rollNo +"  "+ this.name;
}}
```

Output: Object s1 = 1 Ravi
Object s2 = 2 Ram

Object Cloning in Java:

The **object cloning** is a way to create exact copy of an object. The 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:

```
protected Object clone() throws CloneNotSupportedException{

                                                    }
```

The **clone() method** saves the extra processing task for creating the exa

ct copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

Example of Object Cloning

```
class StudentClone implements Cloneable{
int rollno;
String name;
StudentClone(int rollno,String name){
this.rollno=rollno;
this.name=name;
}
 public Object clone()throws CloneNotSupportedException{
return super.clone();
}
public static void main(String args[]) throws CloneNotSupportedException{
 StudentClone s1=new StudentClone(101,"amit");
StudentClone s2;
s2=s1;
s2.clone();
 System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);
}
}
```

Output:101 amit
101 amit

Inner Classes:

=>Java inner class or nested class is a class which is declared inside the class or interface.
=>We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.
=>it can access all the members of outer class including private data members and methods.
=>Without existing one type of object if there is no chance of existing another type of object then we should go for Inner class.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Example:

```
class Car{
        class Engine{

                }

    }
```

=>The relation between Outer class and Inner is not IS-A relation and it is HAS-A relationship(Composition or Aggregation)

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Inner classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Inner classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Difference between nested class and inner class in Java

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

a. Non-static nested class (inner class) : Based on position of declaration and behaviour there are three types of inner classes:

1. **Member inner class or Normal or Regular Inner class**
2. **Anonymous inner class**
3. **Method or Local inner class**

b. Static nested class

a.inner class(Non-static nested class) :

1. Member inner or Normal or Regular class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer{
 //code
```

```
 class Inner{
  //code
 }
}
```

Member inner class example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
class TestOuter{
 private int a=30;
 class Inner{
  void show()
{
System.out.println("This is from inner class:"+a);
}
 }
 public static void main(String args[]){
  TestOuter    obj=new TestOuter();
  TestOuter.Inner  in=obj.new  Inner();
  in.show();
 }
}
```

Output:
This is from inner class:30

Internal working of Java member inner class

The java compiler creates two class files in case of inner class. The class file name of inner class is "Outer$Inner". If you want to instantiate inner class, you must have to create the instance of outer class. In such case, instance of inner class is created inside the instance of outer class.

2.Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

anonymous inner class example using class

```
abstract class Person{
  abstract void display();
}
class TestAnonymousClass{
 public static void main(String args[]){
  Person p=new Person(){
  void display()
{
System.out.println("Hello Anonymous inner class");}
  };
  p.display();
 }
}
```

Output:

```
Hello Anonymous inner class
```

1. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the display() method.
2. An object of Anonymous class is created that is referred by p reference variable of Person type.

Java anonymous inner class example using interface

```
interface Car{
 void display();
}
class TestAnnonymousInterface{
 public static void main(String args[]){
 Car c=new Car(){
  public void display(){
System.out.println("Hello Anonymous inner class using interface");
}
 };
 c.display();
 }
}
```

Output:

```
Hello Anonymous inner class using interface
```

1. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Eatable type.

3.Method or Local inner class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Method or local inner class example

```
class LocalInner{
        private int a=10;
        void display(){
                class Local{
                        void msg(){
                                System.out.println("This is from local inner class:"+a);
                        }
                }
                        Local l=new Local();
                        l.msg();
                }
        public static void main(String args[]){
                LocalInner ob=new LocalInner();
                ob.display();
                }
        }
```

Output:

```
        This is from local inner class:10
```

1) ***Local inner class cannot be invoked from outside the method.***

b. static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

static nested class example with instance method

```
class Outernested{
static int a=10;
static class InnerNested{
void display(){
System.out.println("This is static nested class:"+a);
}
}
public static void main(String args[]){
Outernested.InnerNested ob=new Outernested.InnerNested();
ob.display();
}
}
```

Output:

This is static nested class:10

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.