Java means DURGA SOFT..

CORE JAVA

Material

Java™

India's No.1 Software Training Institute

DURGASOFT

www.durgasoft.com Ph: 9246212143 ,8096969696

Oops concepts

1. ***Inheritance***
2. ***Polymorphism***
3. ***Abstraction***
4. ***Encapsulation***

Inheritance:-

1. *The process of acquiring fields(variables) and methods(behaviors) from one class to another class is called inheritance.*
2. *The main objective of inheritance is code extensibility whenever we are extending class automatically the code is reused.*
3. *In inheritance one class giving the properties and behavior & another class is taking the properties and behavior.*
4. *Inheritance is also known as **is-a** relationship. By using extends keyword we are achieving inheritance concept.*
5. *extends keyword used to achieve inheritance & it is providing relationship between two classes when you make relationship then able to reuse the code.*
6. *In java parent class is giving properties to child class and Child is acquiring properties from Parent.*
7. *To reduce length of the code and redundancy of the code sun people introduced inheritance concept.*

Application code before inheritance

```
class A
{       void m1(){ }
        void m2(){ }
};
class B
{       void m1(){ }
        void m2(){ }
        void m3(){ }
        void m4(){ }
};
class C
{       void m1(){ }
        void m2(){ }
        void m3(){ }
        void m4(){ }
        void m5(){ }
        void m6(){ }
};
```

Application codeafter inheritance

```
class A  //parent class  or super class or base
{
        void m1(){ }
        void m2(){ }
};
class B extends A  //child or sub or derived
{
        void m3(){ }
        void m4(){ }
};
class C extends B
{
        void m5(){ }
        void m6(){ }
};
```

Note 1:-In java it is possible to create objects forboth parent and child classes.

1. If we are creating object for parent class it is possible to call only parent specific methods.

```
A a=new A();
a.m1();a.m2();
```

2. if we are creating object for child class it is possible to call parent specific and child specific.

```
B b=new B();
b.m1();b.m2();  b.m3();b.m4();
C c=new C();
c.m1();  c.m2();  c.m3();c.m4();c.m5();c.m6();
```

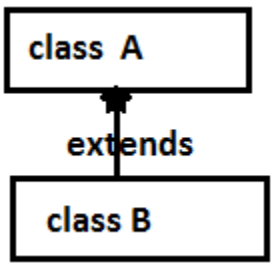
Types of inheritance :-

There are five types of inheritance in java

1. ***Single inheritance***
2. ***Multilevel inheritance***
3. ***Hierarchical inheritance***
4. ***Multiple inheritance***
5. ***Hybrid Inheritance***

Single inheritance:-

- *One class has one and only one direct super class is called single inheritance.*
- *In the absence of any other explicit super class, every class is implicitly a subclass of* ***Object class.***



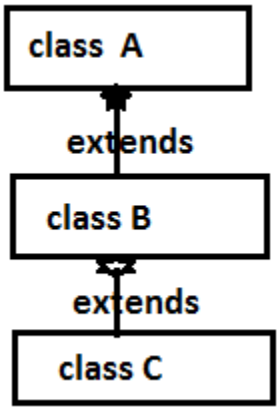
Class B extends A ===>class B acquiring properties of A class.

Example:-

```
class Parent
{       void property(){System.out.println("money");}
};
class Child extends Parent
{       void m1()       {       System.out.println("m1 method");       }
        public static void main(String[] args)
        {       Child c = new Child();
                c.property();   //parent class method executed
                c.m1();         //child class method executed
        }
};
```

Multilevel inheritance:-

One Sub class is extending Parent class then that sub class will become Parent class of next extended class this flow is called multilevel inheritance.



Class B extends A ===> class B acquiring properties of A class
Class C extends B ===> class C acquiring properties of B class
[indirectly class C using properties of A & B classes]

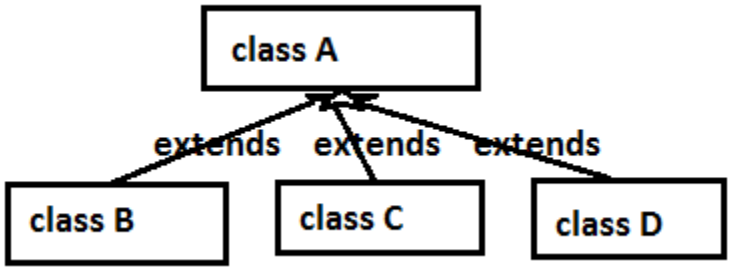
Example:-

```
class A
{       void m1(){System.out.println("m1 method");}
};
class B extends A
{       void m2(){System.out.println("m2 method");}
};
class C extends B
{       void m3(){System.out.println("m3 method");}
        public static void main(String[] args)
        {       A  a = new A();         a.m1();
                B b = new B();          b.m1(); b.m2();;
```

```
            C c = new C();          c.m1(); c.m2(); c.m3();
        }
}
```

Hierarchical inheritance :-

More than one sub class is extending single Parent is called hierarchical inheritance.



Class B extends A ===> class B acquiring properties of A class
Class C extends A ===>class C acquiring properties of A class
Class D extends A ===>class D acquiring properties of A class

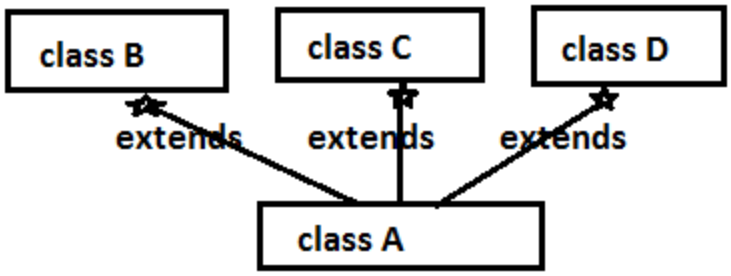
Example:-

```
class A
{       void m1(){System.out.println("A class");}
};
class B extends A
{       void m2(){System.out.println("B class");}
};
class C extends A
{       void m2(){System.out.println("C class");}
};
class Test
{       public static void main(String[] args)
        {       B b= new B();
                b.m1(); b.m2();
                C c = new C();
                c.m1();  c.m2();
        }
};
```

Multiple inheritance:-

- One sub class is extending more than one super class is called Multiple inheritance and java not supporting multiple inheritance because it is creating ambiguity problems (confusion state) .
- Java not supporting multiple inheritance hence in java one class able to extends only one class at a time but it is not possible to extends more than one class.

  Class A extends B ===>valid
  Class A extends B ,C ===>invalid

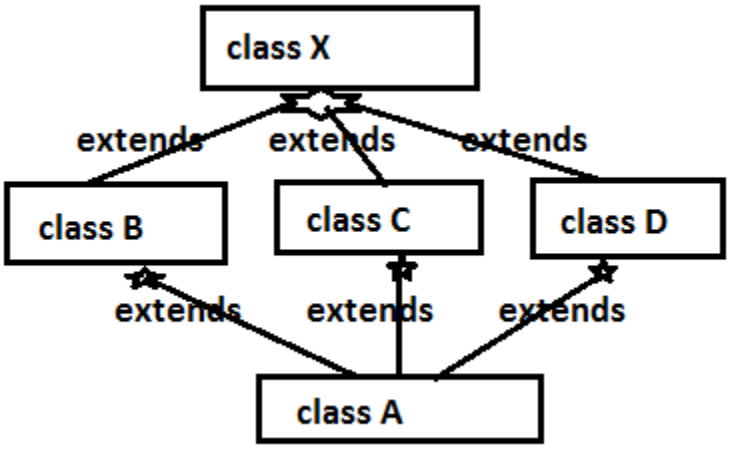


Example:-

```
class A
{       void money(){System.out.println("A class money");}
};
class B
{       void money(){System.out.println("B class money");}
};
class C extends A,B
{       public static void main(String[] args)
        {       C c = new C();
                c.money();      //which method executed A--->money() or B--->money
        }
};
```

Hybrid inheritance:-

- *Hybrid is combination of hierarchical & multiple inheritance .*
- *Java is not supporting hybrid inheritance because multiple inheritance(not supported by java) is included in hybrid inheritance.*



Preventing inheritance:-

You can prevent sub class creation by using final keyword in the parent class declaration.

```
final class Parent //for this class child class creation not possible because it is final.
{
};
class Child extends Parent
{
};
```

compilation error:- cannot inherit from final Parent

Note:-

1. ***Except for the Object class , a class has one direct super class.***
2. ***a class inherit fields and methods from all its super classes whether directly or indirectly.***
3. ***an abstract class can only be sub classed but cannot be instantiated.***
4. ***In parent and child it is recommended to create object of Child class.***

Java.lang.Object class methods :-

public class java.lang.Object {

1) *public final native java/lang/Class<?> getClass();*
2) *public native int hashCode();*
3) *public boolean equals(java.lang.Object);*
4) *protected native java.lang.Object clone() throws java.lang.CloneNotSupportedException;*
5) *public java.lang.String toString();*
6) *public final native void notify();*
7) *public final native void notifyAll();*
8) *public final native void wait(long) throws java.lang.InterruptedException;*
9) *public final void wait(long, int) throws java.lang.InterruptedException;*
10) *public final void wait() throws java.lang.InterruptedException;*
11. *protected void finalize() throws java.lang.Throwable;}*

Example :-

In java if we are extending java class that extended class will become Parent class , if we are not extending Object class will become Parent class.

In below example

A class Parent is ----> Object
B class Parent is ---->A
C class Parent is ---->B

class A

```
{       void m1(){}     };
class B extends A
{       void m2(){}     };
class C extends B
{       void m3(){}     };
```

In above example A class Parent is Object class

*Object class contains ---->**11 methods***
*A class contains ---->**12 methods***
*B class contains ---->**13 methods***
*C class contains ---->**14 methods***

Instanceof operator:-

- *It is used check the type of the object and return Boolean value as a return value.*
  Syntax:- reference-variable instanceof class-name;
  Example:- Test t=new Test();
  t instanceof Test
- Whenever we are using instanceof operator the reference variable and class-name must have some relationship either [parent to child] or [child-parent] otherwise compiler generates error message **"*inconvertible types*"**
- If the relationship is
  - Child – parent returns **true**
  - Parent - child returns **false**

Example :-

```
class Animal{   };
class Dog extends Animal{       };
class  Test
{       public static void main(String[] args)
        {       //syntax: (ref-ver instanceof  class-name);
                String str="ratan";
                Animal a = new Animal();
                Dog d = new Dog();
                Object o = new Object();
                System.out.println(a instanceof Object);        //true  [child-parent]
                System.out.println(d instanceof Animal);        //true [child-parent]
                System.out.println(str instanceof Object);      //true [child-parent]
                System.out.println(o instanceof Animal);        //false [parent-child]
                System.out.println(a instanceof Dog);           //false [parent-child]
                //no relationship compilation error :inconvertible types
                //System.out.println(str instanceof Animal);
        }
}
```

Association:-

- *Class A uses class B*
- *When one object wants another object to perform services for it.*
- *Relationship between teacher and student, number of students associated with one teacher or one student can associate with number of teachers. But there is no ownership and both objects have their own life cycles.*

Example-1:-

```
class Student
{       int sid;
        String sname;
        Student(int sid,String sname) //local variables
        {       //conversion
                this.sid =sid;
                this.sname=sname;
        }
        void disp()
        {       System.out.println("***student details***");
                System.out.println("student name--->"+sname);
                System.out.println("student name--->"+sid);
        }
};
Class   RatanTeacher            //teacher uses Student class "association"
{       public static void main(String[] args)
        {       Student s1 = new Student(111,"ratan");
                Student s2 = new Student(222,"anu");
                s1.disp();              s2.disp();
        }
};
```

Example-2:-

```
class Ratan
{       void disp(){System.out.println("ratan : corejava");}
};
class Anu
{       void disp(){System.out.println("anu : advjava");}
};
class Sravya
{       void disp(){System.out.println("Sravya : ocjp");}
};
class Student   //student uses different teachers "association"
{       public static void main(String[] args)
        {       Ratan r = new Ratan();  r.disp();
                Anu a = new Anu();      a.disp();
                Sravya d = new Sravya();        d.disp();
        }
};
```

Aggregation:-

- *Class A has instance of class B.*
- *Class A can exists without presence of class B . a university can exists without chancellor.*
- *Take the relationship between teacher and department. A teacher may belongs to multiple departments hence teacher is a part of multiple departments but if we delete department object teacher object will not destroy.*

Example -1:-

//Teacher.java

```
class Teacher
{       //instance variables
        String tname,sub;
        Teacher(String tname,String sub)//local variables
        {       //conversion
                this.tname=tname;               this.sub=sub;
        }
};
```

//Department.java:-

```
class Department //if we delete department teacher can exists is called aggregation
{       //instance variables
        int did;
        Teacher t;
        Department(int did ,Teacher t)   //local variables
        {       //conversion
                this.did = did;             this.t = t;
        }
        void disp()
        {       System.out.println("Department id :--->"+did);
                System.out.println("Teacher details :--->"+t.tname+"---"+t.sub);
        }

public static void main(String[] args)
        {       Teacher x1 = new Teacher("ratan","corejava");
                Department d = new Department(100,x1);
                d.disp();
```

```
        }
}
```

Example -2:

Address.java

```
class Address
{       //instance variables
        String country, state;
        int hno;
        Address(String country,String state,int hno)//local variables
        {//passing local variable values to instance variables (conversion)
        this.country = country;         this.state= state;      this.hno = hno;
        }
};
```


Heroin.java:

```
class Heroin
{       //instance varaibles
        String hname; int hage;
        Address addr;//reference of address class [address class can exists without Heroin class]
        Heroin(String hname,int hage,Address addr)//localvariables
        {       //conversion of local variables to instance variables
                this.hname = hname;             this.hage = hage;               this.addr = addr;
        }
        void display()
        {       System.out.println("*********heroin details******");
                System.out.println("heroin name-->"+hname);
                System.out.println("heroin age-->"+hage);
                //printing address values
        System.out.println("heroin address-->"+addr.country+" "+addr.state+" "+addr.hno)
        }
        public static void main(String[] args)
        {       //object creation of Address class
```

```
        Address a1  = new Address("india","banglore",111);
        Address a2  = new Address("india","mumbai",222);
        Address a3  = new Address("US","california",333);
        //Object creation of Heroin class by passing address object name
        Heroin h1 = new Heroin("anushka",30,a1);
        Heroin h2 = new Heroin("KF",30,a2);
        Heroin h3 = new Heroin("AJ",40,a3);
        h1.display();        h2.display();        h3.display();
    }
}
```

Example-3:-

Test1.java:-

```
class Test1
{  //instance variables
    int a;
    int b;
    Test1(int a,int b)
    {    this.a=a;
         this.b=b;
    }
};
```

Test2.java:-

```
class Test2
{//instance variables
    boolean b1;
    boolean b2;
Test2(boolean b1,boolean b2)
    {    this.b1=b1;
         this.b2=b2;
    }
};
```

Test3.java:-

```
class Test3
{  //instance variables
    char ch1;
    char ch2;
Test3(char ch1,char ch2)
    {    this.ch1=ch1;
         this.ch2=ch2;
    }
};
```

MainTest.java:-

```
class Test
{    //instance variables
    Test1 t1;
    Test2 t2;
    Test3 t3;
    Test(Test1 t1 ,Test2 t2,Test3 t3)//constructor [local variables]
    {    //conversion of local-instance
         this.t1 = t1;
         this.t2 = t2;
         this.t3 = t3;
    }
    void display()
    {    System.out.println("Test1 object values:-→"+t1.a+"----- "+t1.b);
         System.out.println("Test2 object values:-→"+t2.b1+"----- "+t2.b2);
         System.out.println("Test3 object values:-→"+t3.ch1+"----- "+t3.ch2);
    }
    public static void main(String[] args)
    {    Test1 t = new Test1(10,20);
```

```
                Test2 tt = new Test2(true,true);
                Test3 ttt = new Test3('a','b');
                Test main = new Test(t,tt,ttt);
                main.display();
        }
};
```

Composition :-

- *Class A owns class B , it is a strong type of aggregation. There is no meaning of child without parent.*
- *Order consists of list of items without order no meaning of items. **or** bank account consists of transaction history without bank account no meaning of transaction history **or w**ithout student class no meaning of marks class.*
- *Let's take Example house contains multiple rooms, if we delete house object no meaning of room object hence the room object cannot exists without house object.*
- *Relationship between question and answer, if there is no meaning of answer without question object hence the answer object cannot exist without question objects.*
- *Relationship between student and marks, there is no meaning of marks object without student object.*

Example :-

//Marks.java

```
class Marks
{       int m1,m2,m3;
        Marks(int m1,int m2,int m3)      //local variables
        {       //conversions
                this.m1=m1;
                this.m2=m2;
                this.m3=m3;
        }
};
```

//student.java

```
class Student
{       //instance variables
        Marks mk;       //without student class no meaning of marks is called "composition"
        String sname;
        int sid;
        Student(Marks mk,String sname,int sid) //local variables
        {       this.mk = mk;
                this.sname = sname;
                this.sid = sid;
        }
        void display()
        {       System.out.println("student name:-->"+sname);
                System.out.println("student id:-->"+sid);
                System.out.println("student marks:-->"+mk.m1+"---"+mk.m2+"--"+mk.m3);
        }
        public static void main(String[] args)
        {       Marks m1 = new Marks(10,20,30);
```

```
        Marks m2 = new Marks(100,200,300);
        Student s1 = new Student(m1,"ratan",111);
        Student s2 = new Student(m2,"anu",222);
        s1.display();
        s2.display();
    }
}
```


Object delegation:-

The process of sending request from one object to another object is called object delegation.

Example-1:-

```
class Test1
{ //instance variables
    int a=10;
    int b=20;
    static void add()//static method
    {   Test1 t = new Test1();
        System.out.println(t.a+t.b);
    }
    static void mul()//static method
    {   Test1 t = new Test1();
        System.out.println(t.a*t.b);
    }
    public static void main(String[] args)
    {   Test1.add();    //calling static method add()
        Test1.mul();    //calling static method mul()
    }
};
```

Example-2 :-

```
class Test1
{ //instance variables
    int a=10;       int b=20;
    static Test1 t = new Test1();   // t is a variable of Test1 type (instance variable)
```

```
        static void add()       //static method
        {       System.out.println(t.a+t.b);    }
        static void mul()       //static method
        {       System.out.println(t.a*t.b);    }
        public static void main(String[] args)
        {       Test1.add();    //calling static method add()
                Test1.mul();    //calling static method mul()
        }
};
```

Example-3:-

```
class Developer
{       void task1(){System.out.println("task-1");}
        void task2(){System.out.println("task-2");}
};
class TeamLead
{       Developer d = new Developer();//instance variable
        void display1()
        {       d.task1(); d.task2();    }
        void display2()
        {       d.task1(); d.task2();    }
        public static void main(String[] args)
        {       TeamLead t = new TeamLead();
                t.display1(); t.display2();
        }
};
```

Example -4:-

```
class RealPerson        //delegate   class
{       void book(){System.out.println("real java book");}
};
class DummyPerson   //delegator class
{       RealPerson r = new RealPerson();
        void book(){r.book();} //delegation
};
class Student
{       public static void main(String[] args)
        {       //outside world thinking dummy Person doing work but not.
                DummyPerson d = new DummyPerson();
                d.book();
        }
};
```

Super keyword:-

Super keyword is holding super class object. And it is representing

1. *Super class variables*
2. *Super class methods*

3. *Super class constructors*

super class variables calling:-

Super keyword not required:-

```
class Parent
{       int x=10, y=20;//instance variables
};
class Child extends Parent
{       int a=100,b=200;//instance variables
        void m1(int i,int j)//local variables
        {       System.out.println(i+j);          //local variables addition
                System.out.println(a+b);          //current class variables addition
                System.out.println(x+y);          //super class variables addition
        }
        public static void main(String[] args)
        {       Child c = new Child();
                c.m1(1000,2000);
        }//end main
};//end class
```

In above example current class and super class variables having different names so this keyword and super keyword not required.

Super keyword required:-

```
class Parent
{       int a=10,b=20;//instance variables
};
class Child extends Parent
{       //instance variables
        int a=100;
        int b=200;
        void m1(int a,int b)      //local variables
        {       System.out.println(a+b);                  //local variables addition
                System.out.println(this.a+this.b);        //current class variables addition
                System.out.println(super.a+super.b);      //super class variables addition
        }
        public static void main(String[] args)
        {       Child c = new Child();
                c.m1(1000,2000);
        }
};
```

In bove example sub class and super class having same variable names hence to represent.

a. **sub class variables use this keyword.**
b. **Super class variables use super keyword.**

super class methods calling:-

superkeyword not requeired:-

```
class Parent
{       void m1(int a)   {          System.out.println("parent m1()-->"+a);   }
};
class Child extends Parent
{       void m2(int a)   {        System.out.println("child m1()-->"+a);    }
        void m3()
        {       m1(10);         // parent class m1(int) method calling
                System.out.println("child m2()");
                m2(100);        // child class m2(int) method clling
        }
        public static void main(String[] args)
        {       Child c = new Child();
                c.m3();
        }
};
```

In above example sub class and super class contains different methods so super keyword not required.

Superkeyword requeired:-

```
class Parent
{       void m1(int a){  System.out.println("parent m1()-->"+a); }
};
class Child extends Parent
{       void m1(int a){  System.out.println("child m1()-->"+a); }
        void m2()
        {       this.m1(10);                //child class m1(int) method calling
                System.out.println("child m2()");
```

```
                super.m1(100);          // parent class m1(int) method calling
        }
        public static void main(String[] args)
        {       Child c = new Child();
                c.m2();
        }
};
class Parent
{       void m1(){System.out.println("parent m1() method");}
};
class Child extends Parent
{       void m1()       {System.out.println("child class m1() method");}
        void m3()
        {       this.m1();      //current class method is executed
                super.m1();     //super class method will be executed
        }
        public static void main(String[] args)
        {       new Child().m3();
        }
};
```


In above exmple super class and sub class contains methods with same names(*m1()* ***) at that situation to represent.***

a. ***Super class methods use super keyword.***
b. ***Sub clss methods use this keyword.***

<u>super class constructors calling:-</u>

super() ----> **super class 0-arg constructor calling**
super(10) ----> **super class 1-arg constructor calling**
super(10,20) ----> **super class 2-arg constructor calling**
super(10,'a',true)----> **super class 3-arg constructor calling**

<u>Example-1:-</u>To call super class constructor use super keyword.

```
class Parent
{       Parent() {System.out.println("parent 0-arg constructor");}
};
class Child extends Parent
```

```
{       Child()
        {       this(10);       //current class 1-arg constructor calling
                System.out.println("Child 0-arg constructor");
        }
        Child(int a)
        {       super();        //super class 0-arg constructor calling
                System.out.println("child 1-arg constructor--->"+a);
        }
        public static void main(String[] args)
        {       Child c = new Child();
        }//end class
};//end main
```

Example-2:-

nIside the constructor super keyword must be first statement otherwise compiler generate error message ***"call to super must be first line in constructor".***

No compilation error:-

```
Child()
{       this(10);//current class 1-arg constructor calling(must be first line)
        System.out.println("Child 0-arg constructor");
}
Child(int a)
{       super();//super class 0-arg constructor calling(must be first line)
        System.out.println("child 1-arg constructor--->"+a);
}
```

Compilation Error:-

```
Child()
{       System.out.println("Child 0-arg constructor");
        this(10);//current class 1-arg constructor calling
}
Child(int a)
{       System.out.println("child 1-arg constructor--->"+a);
        super();//super class 0-arg constructor calling  (compiltion Error)
}
```

Example-3:-

Inside the constructor this keyword must be first statement and super keyword must be first statement hence inside the constructor it is possible to use either this keyword or super keyword but both at a time not possible.

No compilation Error:-

```
Child()
{       this(10);//current class 1-arg constructor calling(must be first line)
        System.out.println("Child 0-arg constructor");
}
Child(int a)
{       super();//super class 0-arg constructor calling(must be first line)
        System.out.println("child 1-arg constructor--->"+a);
}
```

Compilation Error:-

```
Child()
{       this(10);//current class 1-arg constructor calling
        super();//super class 0-arg constructor calling
        System.out.println("Child 0-arg constructor");
}
```

Example-4:-

1. Inside the constructor (weather it is default or parameterized) if we are not declaring **super or this** keyword at that situation compiler generate super() keyword at first line of the constructor.
2. If we are declaring t lest one constructor compiler is not responsible to generate super() keyword.
3. The compiler generated super keyword is always 0-argument constructor calling.

```
class Parent
{       Parent() {    System.out.println("parent 0-arg constructor");    }
};
class Child extends Parent
{
        Child()
        {       //super();  generated by compiler at compilation time
                System.out.println("Child 0-arg constructor");
        }
        public static void main(String[] args)
        {       Child c = new Child();
        }
};
```

D:\>java Child
parent 0-arg constructor
Child 0-arg constructor

Example-5:-

In below example parent class default constructor is executed that is provided by compiler.

```
class Parent
{       //default constructor Parent() { }  generated by compiler at compilation time
};
```

```
class Child extends Parent
{       Child()
        {       //super(); generated by compiler at compilation time
                System.out.println("Child 0-arg constructor");
        }
        public static void main(String[] args)
        {       Child c = new Child();
        }
};
```

Example-6:-

By using below example we are assigning values to instance variable at the time of object creation eith the help of parameterized constructor.

```
class Parent
{       int a;    //instance variable
        Parent(int a)//local variable
        {       //conversion of local variable to instance variable
                this.a=a;
        }
};
class Child extends Parent
{       boolean x;              //instance variable
        Child(boolean x)         //local variable
        { super(10);            //super class constructor calling
          this.x =x ;           //conversion of local variable to instance variable
                                (passing local variable value to instance variable)
        }
        void display()
        {       System.out.println(a);
                System.out.println(x);
```

```
        }
        public static void main(String[] args)
        {       Child c = new Child(true);
                c.display();
        }
};
```

Example-7:-

In below example child class is calling parent class 0-argument constructor since not there so compiler generate error message

.

```
class Parent
{       Parent(int a)     {   System.out.println("parent 1-arg cons-->"+a);     }
};
class Child extends Parent
{       Child()
        {       //super(); generated by compiler t compilation time
                System.out.println("Child 0-arg constructor");
        }
        public static void main(String[] args)
        {       Child c = new Child();
        }
};
```

Example-8:-

In below example in child class 1-argument constructor compiler generate super keyword hence parent class 0-argument constructor is executed.

```
class Parent
{       Parent(){System.out.println("parent 0-arg cons");          }
};
class Child extends Parent
{       Child()
        {       this(10); //current class 1-argument constructor calling
                System.out.println("Child 0-arg constructor");
        }
        Child(int a)
        {        //super(); generated by compiler
                System.out.println("child 1-arg cons");
        }
        public static void main(String[] args)
        {       Child c = new Child();
        }
};
```

D:\>java Child
parent 0-arg cons
child 1-arg cons
Child 0-arg constructor

Example-9:-

Inside the constructor either it is zero argument or parameterized if we are not providing super or this keyword at that situation compiler generate super keyword at first line of constructor.

```
class Parent
{       Parent()  {   System.out.println("parent 0-arg cons");    }
};
class Child extends Parent
{       Child()
        {       //super(); generated by compiler
                System.out.println("Child 0-arg constructor");
        }
        Child(int a)
        {       //super(); generated by compiler
                System.out.println("child 1-arg cons");
        }
        public static void main(String[] args)
        {
                Child c = new Child();
                Child c1 = new Child(10);
        }
};
```

D:\>java Child
parent 0-arg cons
Child 0-arg constructor
parent 0-arg cons
child 1-arg cons

Example-10:-

In below compiler generate default constructor and inside that default constructor super keyword is generated by compiler.

Application code before compilation:- (.java)

```
class Parent
{       Parent(){
System.out.println("parent 0-arg cons");}
};
class Child extends Parent
{       public static void main(String[] args)
        {       Child c = new Child();
        }
};
```

```
class Parent
{       Parent(){System.out.println("parent 0-
arg cons");      }
};
class Child extends Parent
{       /* below code is generated by compiler
        Child()
        {       super();
        }*/
        public static void main(String[] args)
        {       Child c = new Child();
        }
};
```

Application code after compilation:- (.class)

Example-11:-

In below example inside the 1-argument constructor compiler generate super( ) keyword hence it is executing super class ***(Object)*** *0-argument constructor is executed.*

Application code before compilation:- (.java)

```
class Test
{
        Test(int a){
        System.out.println("Test 1-arg cons");
        }
        public static void main(String[] args)
        {
                Test t = new Test(10);
        }
};
```

Application code after compilation:- (.class)

(Object class 0-arg constructor executed)

```
class Test  extends Object
{       Test(int a)
        {
        super(); //generated by compiler
        System.out.println("Test 1-arg cons");
        }
        public static void main(String[] args)
        {       Test t = new Test(10);
        }
};
```

Note 1:- in java if we are extending class that extended class will become super class

Ex :- ***class B{ }***
class A extends B //B class is Parent of A class
{ }

*Note 2 :- in java if we are not extending any class then **Object** class will become parent of that class.*

Ex :- class A { } //in this Object class is Parent of A class

Note:-

1. *Every class in the java programming either directly or indirectly child class of **Object.***
2. *Root class for all java classes is Object class.*
3. *The object class present in **java.lang** package*

Example : assignment

```
class GrandParent
{       int c;
        GrandParent(int c)
        {       this.c=c;
        }
};
class Parent extends GrandParent
{       int b;
        Parent(int b,int c)
        {       super(c);
                this.b=b;
        }
};
class Child extends Parent
{       int a;
        Child(int a,int b,int c)
        {       super(b,c);
                this.a=a;
        }
        void disp()
        {       System.out.println("child class ="+a);
                System.out.println("parent class ="+b);
                System.out.println("grandparent class ="+c);
        }
        public static void main(String[] args)
        {       new Child(10,20,30).disp();
        }
};
```

Super class instance blocks:-

Example-1:-

In parent and child relationship first parent class instance blocks are executed then child class instance blocks are executed because first parent class object constructors executed.

```
class Parent
{       {System.out.println("parent instance block");}//instance block
};
class Child extends Parent
{
        {   System.out.println("Child instance block");    }  //instance block
        Child() {       System.out.println("chld 0-arg cons");        }//constructor
        public static void main(String[] args){
        Child c = new Child();
        }
};
```


Example-2:-

In below example just before child class instance blocks first parent class instance blocks are executed.

```
class Parent
{       {System.out.println("parent instance block");}//instance block
        Parent(){System.out.println("parent cons");}//constructor
};
class Child extends Parent
{       {System.out.println("Child instance block");}//instance block
        Child()
        {       //super(); generated by compiler
                System.out.println("chld 0-arg cons");}
        }
        Child(int a)
        {       //super(); generated by compiler
                System.out.println("chld 1-arg cons");
        }
        public static void main(String[] args)
        {       Child c = new Child();
```

```
                Child c1 = new Child(10);
        }
};
```

```
D:\>java Child
parent instance block
parent cons
Child instance block
chld 0-arg cons
parent instance block
parent cons
Child instance block
chld 1-arg cons
```

```
class Parent
{
        {System.out.println("parent class ins block");}
        Parent()
        {       System.out.println("parent class 0-arg cons");
        }
};
class Child extends Parent
{
        {System.out.println("Child class ins block");}
        Child()
        {       //super();  generated by compiler
                System.out.println("child class 0-arg cons");
        }
        public static void main(String[] args)
        {       new Child();
        }
};
```

```
E:\>java Child
parent class static block
child class static block
parent class ins block
parent class 0-arg cons
Child class ins block
child class 0-arg cons
parent class ins block
parent class 0-arg cons
Child class ins block
child class 0-arg cons
```

Parent class static block:-

Example-1:-

In parent and child relationship first parent class static blocks are executed only one time then child class static blocks are executed only one time because static blocks are executed with respect to .class loading.

```
class Parent
{       static{System.out.println("parent static block");}//static block
};
class Child extends Parent
{
        static{System.out.println("child static block");} //static block
        public static void main(String[] args)
        {
        }
};
class Parent
{       Parent(){System.out.println("parent 0-arg cons");}
        {System.out.println("parent class instance block");}
        static{System.out.println("parent class static  block");}
};
class Child extends Parent
{       {System.out.println("child class instance block");}
        Child()
        {       //super(); generated by compiler
                System.out.println("child class 0-arg cons");
        }
        static {System.out.println("child class static block");}
        public static void main(String[] args)
        {       new Child();
        }
};
```


Example-2:-

Note 1:-*instanceblocks execution depends on number of object creations but not number of constructor executions. If we are creating 10 objects 10 times constructors are executed just before constructor execution 10 times instance blocks are executed.*

Note 2:-*Static blocks execution depends on .class file loading hence the static blocks are executed only one time for single class.*

```
class Parent
{       static {System.out.println("parent static  block");}//static block
        {System.out.println("parent instance  block");}//instance block
        Parent(){System.out.println("parent 0-arg cons");}//constructor
};
class Child extends Parent
{       static {System.out.println("Child static block");}//static block
        {System.out.println("child instance  block");} //instance block
        Child()
        {       //super(); generated by compiler
                System.out.println("Child 0-arg cons");}
        Child(int a){
                this(10,20);//current class 2-argument constructor calling
                System.out.println("Child 1-arg cons");}
        Child(int a,int b)
        {       //super();  generated by compiler
                System.out.println("Child 2-arg cons");
        }
        public static void main(String[] args)
        {       Parent p = new Parent();  //creates object of Parent class
                Child c = new Child();  //creates object of Child class
                Child c1 = new Child(100);//creates object of  child class
        }
};
D:\>java Child
parent static  block
Child static block
parent instance  block
parent 0-arg cons
parent instance  block
parent 0-arg cons
child instance  block
Child 0-arg cons
parent instance  block
parent 0-arg cons
child instance  block
Child 2-arg cons
Child 1-arg cons
```

Polymorphism:-

- One thing can exhibits more than one form is called polymorphism.
- Polymorphism shows some functionality(method name same) with different logics execution.
- The ability to appear in more forms is called polymorphism.
- Polymorphism is a Greek word poly means many and morphism means forms.

There are two types of polymorphism in java

1) Compile time polymorphism / static binding / early binding
   [method execution decided at compilation time]
   Example :- method overloading.
2) Runtime polymorphism /dynamic binding /late binding.
   [Method execution decided at runtime].
   Example :- method overriding.

Compile time polymorphism [Method Overloading]:-

1) If java class allows two methods with same name but different number of arguments such type of methods are called overloaded methods.
2) We can overload the methods in two ways in java language
   a. By passing different number of arguments to the same methods.
      void m1(int a){}
      void m1(int a,int b){}
   b. Provide the same number of arguments with different data types.
      void m1(int a){}
      void m1(char ch){}
3) If we want achieve overloading concept one class is enough.
4) It is possible to overload any number of methods in single java class.

Types of overloading:-

a. *Method overloading* } *explicitly by the programmer*
b. *Constructor overloading* }
c. *Operator overloading* } *implicitly by the JVM('+' addition& concatenation)*

Method overloading:-

Example:-

```
class Test
{       //below three methods are overloaded methods.
        void m1(char ch)        {System.out.println(" char-arg constructor ");    }
        void m1(int i)          {System.out.println("int-arg constructor ");     }
        void m1(int i,int j)    {System.out.println(i+j);                         }
        public static void main(String[] args)
        {Test t=new Test();
        //three methods execution decided at compilation time
        t.m1('a');t.m1(10);t.m1(10,20);
        }
}
```

Example :- overloaded methods vs. all data types

```
class Test
{       void m1(byte a)   {    System.out.println("Byte value-->"+a);   }
        void m1(short a)  {    System.out.println("short value-->"+a);  }
        void m1(int a)    {    System.out.println("int value-->"+a);    }
        void m1(long a)   {    System.out.println("long value is-->"+a); }
        void m1(float f)  {    System.out.println("float value is-->"+f); }
        void m1(double d) {    System.out.println("double value is-->"+d);  }
        void m1(char ch)  {    System.out.println("character value is-->"+ch);  }
        void m1(boolean b) {   System.out.println("boolean value is-->"+b);  }
        void sum(int a,int b)
        {       System.out.println("int arguments method");
                System.out.println(a+b);
        }
        void sum(long a,long b)
        {       System.out.println("long arguments method");
                System.out.println(a+b);
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                t.m1((byte)10);         t.m1((short)20);        t.m1(30);       t.m1(40);
                t.m1(10.6f);            t.m1(20.5);             t.m1('a');      t.m1(true);
                t.sum(10,20);
                t.sum(100L,200L);
        }
}
```

Constructor Overloading:-

The class contains more than one constructors with same name but different arguments is called constructor overloading.

```
class Test
{       //overloaded constructors
        Test()          {       System.out.println("0-arg constructor");          }
        Test(int i)     {       System.out.println("int argument constructor"); }
        Test(char ch,int i){    System.out.println(ch+"-----"+i);                 }
        public static void main(String[] args)
        {       Test t1=new Test();     //zero argument constructor executed.
                Test t2=new Test(10);   // one argument constructor executed.
                Test t3=new Test('a',100);//two argument constructor executed.
        }
}
```

Operator overloading:-

- One operator with different behavior is called Operator overloading .
- Java is not supporting operator overloading but only one overloaded in java language is '+'.
    - If both operands are integer + perform addition.
    - If at least one operand is String then + perform concatenation.

Example:-

```
class Test
{       public static void main(String[] args)
        {       int a=10;
                int b=20;
                System.out.println(a+b);          //30    [addition]
                System.out.println(a+"ratan");    //10Ratan [concatenation]


        }
}
```

Runtime polymorphism [Method Overriding]:-

1) If we want to achieve method overriding we need two class with parent and child relationship.
2) The parent class method contains some implementation (logics).
    a. If child is satisfied use parent class method.
    b. If the child class not satisfied (required own implementation) then override the method in Child class.
3) A subclass has the same method as declared in the super class it is known as method overriding.

The parent class method is called ===> **overridden method**
The child class method is called ===> **overriding method**

While overriding methods must fallow these rules:-

1) ***While overriding child class method signature & parent class method signatures must be same otherwise we are getting compilation error.***
2) ***The return types of overridden method & overriding method must be same.***
3) ***While overriding the methods it is possible to maintains same level permission or increasing order but not decreasing order, if you are trying to reduce the permission compiler generates error message "attempting to assign weaker access privileges ".***
4) ***You are unable to override final methods. (Final methods are preventing overriding).***
5) ***While overriding check the covariant-return types.***
6) ***Static methods are bounded with class hence we are unable to override static methods.***

7) It is not possible to override private methods because these methods are specific to class.

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

```
class Animal
{       void instanceMethod(){System.out.println("instance method in Animal");}
        static void staticMethod(){System.out.println("static method in Animal");}
};
class Dog extends Animal
{       void instanceMethod(){System.out.println("instance method in Dog");}//overriding
        static void staticMethod(){System.out.println("static method in Dog");}//hiding
        public static void main(String[ ] args)
        {       Animal a = new Dog();
                a.instanceMethod();
                a.staticMethod(); // [or] Animal.instanceMethod();
        }
};
```

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

The `Cat` class overrides the instance method in `Animal` and hides the static method in `Animal`.
The `main` method in this class creates an instance of `Cat` and invokes`testClassMethod()` on the class and `testInstanceMethod()` on the instance.

Example-1 :-method Overriding

```
class Parent      //parent class
{       void property()         {System.out.println("money+land+hhouse");}
        void marry()            {System.out.println("black girl");      }//overridden method
};
class Child extends Parent//child class
{       void marry()    {System.out.println("white girl/red girl");}      //overriding method
        public static void main(String[] args)
        {       Child c=new Child();
                c.property();           c.marry();
                Parent p=new Parent();
                p.property();           p.marry();
```

```
        }
};
```

Covariant return types :-

Example 1:-

in below example overriding is not possible because overridden method return type & overriding method return types are not matched.

```
class Parent
{       void m1(){}
};
class Child extends Parent
{int m1(){}
};
```

Compilation error:- m1() in Child cannot override m1() in Parent
return type int is not compatible with void

Example-2:-

1) *Before java 5 version it is not possible to override the methods by changing it's return types .*
2) *From java5 versions onwardsjava supports support covariant return types it means while overriding it is possible to change the return types of parent class method(overridden method) &child class method(Overriding).*
3) *The return type of overriding method is must be sub-type of overridden method return type this is called covariant return types.*

```
class Animal
{       void m2(){System.out.println("Animal class m2() method");}
        Animal m1()
        {       return new Animal();
        }
}
class Dog extends Animal
{       Dog m1()
        {       return new Dog();
        }
        public static void main(String[] args)
        {       Dog d = new Dog();      d.m2();
                Dog d1 = d.m1();        //[d.m1() returns Dog object]
                d1.m2();
                Animal a = new Animal();
                a.m2();
                Animal a1 = a.m1();     //  [a.m1() returns Animal object]
                a1.m2();
        }
};
```

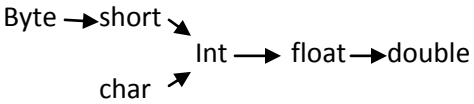
Type-casting:-

The process of converting data one type to another type is called type casting.
There are two types of type casting

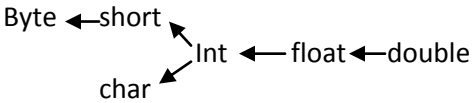
1. Implicit typecasting /widening/up casting
2. Explicit type-casting (narrowing)/do

Type casting chart:-

Up-casting :-



down-casting:-



When we assign higher value to lower data type range then compiler will rise compiler error "possible loss of precision" but whenever we are type casting ***higher data type-lower data type*** *compiler won't generate error message but we will loss the data.*

Implicit-typecasting:- (widening) or (up casting)

1. *When we assign lower data type value to higher data type that typecasting is called up- casting.*
2. *When we perform up casting data no data loss.*
3. *It is also known as up-casting or widening.*
4. *Compiler is responsible to perform implicit typecasting.*

Explicit type-casting:- (Narrowing) or (down casting)

1. *When we assign a higher data type value to lower data type that type-casting is called down casting.*
2. *When we perform down casting data will be loss.*
3. *It is also known as narrowing or down casting.*
4. *User is responsible to perform explicit typecasting.*

Note :- Parent class reference variable is able to hold child class object but Child class reference variable is unable to hold parent class object.

```
class Parent
{       };
class Child extends Parent
{       };
```

```
Parent p = new Child();   //valid
Child c = new Parent();   //invalid
```

Example :-type casting

```
 class Parent
{       };
class Child extends Parent
{       };
class Test
{       public static void main(String[] args)
        {       //implicit typecasting (up casting)
                byte b=120;
                int i=b;  //[automatic conversion of byte-int]
                System.out.println(b);
                char ch='a';
                int a=ch;          //[automatic conversion of char to int]
                System.out.println(a);
                long l1=20;
                float f = l1;      //[automatic conversion of long-float]
                System.out.println(f);
                /*below examples it seems up-casting but compilation error:possible loss of precision
                : conversion not possible
                byte i=100; (1 byte size)
                char ch=i;        (assigned to 2 bytes char)
                System.out.println(ch);
                char ch='a';
                short a=ch;
                System.out.println(a);   compilation error:possible loss of precision
                float f = 10.5f;
                long l = f;
                System.out.println(l);   compilation error:possible loss of precision
                float f=10.5f;
                long l = f;
                System.out.println(l);   compilation error:possible loss of precision (memory
                representation different)                */
                //explicit-typecasting (down-casting)
                // converted-type-name  var-name = (converted-type-name)conversion-var-type;
                int a1=120;
                byte b1 =(byte)a1;
                System.out.println(b1);
                int a2=130;
                byte b2 =(byte)a2;
                System.out.println(b2);
                float ff=10.5f;
                int x = (int)ff;
                System.out.println(x);
                Parent p = new Child();
```

```
                //target-type variable-name=(target-type)source-type;
                Child c1 =(Child)p;
                Parent p = new Child();
                Child c1 = (Child)p;
        }
}
```

Example-2:-

- In java parent class reference variable is able to hold Child class object but Child class reference variable unable to hold Parent class object.
  - **Parent p = new Child(); ---->valid**
  - **Child c = new Parent(); ---->invalid**

```
class Parent
{       void m1(){System.out.println("parent m1 method");}      //overridden method
}
class Child extends Parent
{       void m1(){System.out.println("child m1 method");}       //override method
        void m2(){System.out.println("child m2 method");}       //direct method of child class
        public static void main(String[] args)
        {
        //parent class is able to hold child class object
        Parent p1 = new Child();   //creates object of Child class
        p1.m1();          //child m1() will be executed
        //p1.m2(); Compilation  error we are unable to call m2() method
        Child c1 =(Child)p1;  //type casting parent reference variable to child object.
        c1.m1();
        c1.m2();
        }
};
```

- ➔ *In above example parent class is able to hold child class object but when you call* ***p.m1();*** *method compiler is checking m1() method in parent class at compilation time. But at runtime child object is created hence Child method will be executed.*
- ➔ *Based on above point decide in above method execution decided at runtime hence it is a runtime polymorphism.*
- ➔ *When you call* ***p.m2 (); compiler****is checking m2 () method in parent class since not there so compiler generate error message. Finally it is not possible to call child class m2 () by using parent reference variable even thought child object is created.*
- ➔ *Based on above point we can say by using parent reference it is possible to call only overriding methods (****m1 ()*** *) of child class but it is not possible to call direct method(****m2()*** *) of child class.*
- ➔ *To overcome above limitation to call child class method perform typecasting.*

Example :- importance of converting parent class reference variable into child class object

```
//let assume predefined class
class ActionForm
{       void xxx(){} //predefined method
        void yyy(){} //predefined method
};
class LoginForm  extends ActionForm  //assume to create LoginForm our class must extends ActonForm
```

```
{       void m1(){System.out.println("LoginForm m1 method");}//method of LoginFrom class
        void m2(){System.out.println("LoginForm m2 method");}//method of LoginFrom class

        public static void main(String[] args)
        {       //assume server(Tomcat,glassfish…) is creating object of LoginForm
                ActionForm af = new LoginForm();//creates object of LoginForm class
                //af.m1();      af.m2();  //by using af it is not possible to call m1() & m2()

                LoginForm lf = (LoginForm)af;//type casting
                lf.m1();
                lf.m2();
        }
};
```

Example :-[overloading vs. overriding]

```
class Parent
{       //overloaded methods
        void m1(int a){System.out.println("parent int m1()-->"+a);}//overridden method
        void m1(char ch){System.out.println("parent char m1()-->"+ch);}//overridden method
};
class Child extends Parent
{       //overloaded methods
        void m1(int a){System.out.println("Child int m1()-->"+a);}//overriding method
        void m1(char ch){System.out.println("child char m1()-->"+ch);}//overriding method
        public static void main(String[] args)
        {
                Parent p = new Parent();//[it creates object of Parent class]
                p.m1(10); p.m1('s'); //10 s [parent class methods executed]
                Child c = new Child();//[it creates object of Child class]
                c.m1(100); c.m1('a'); //[100 a Child class methods executed]
                Parent p1 = new Child();//[it creates object of Child class]
                p1.m1(1000);    p1.m1('z'); //[1000 z child class methods executed]
        }
};
```

Example:- method overriding vs. Hierarchical inheritance

```
class Heroin
{       int rating(){return 0;}
};
class Anushka extends Heroin
{       int rating(){return 1;}
};
class Nazriya extends Heroin
{       int rating(){return 5;}
};
class Kf extends Heroin
```

```
{       int rating(){return 2;}
}
class Test
{       public static void main(String[] args)
        {       /*Heroin h,h1,h2,h3;
                h = new Heroin();
                h1 = new Anushka();
                h2 = new Nazriya();
                h3 = new Kf();*/
                Heroin h = new Heroin();
                Heroin h1 = new Anushka();
                Heroin h2 = new Nazriya();
                Heroin h3 = new Kf();
                System.out.println("Heroin rating      :--->"+h.rating());
                System.out.println("Anushka rating   :--->"+h1.rating());
                System.out.println("Nazsriya rating :--->"+h2.rating());
                System.out.println("Kf  rating           :--->"+h3.rating());
        }
};
```

In above example when you call rating() method compilation time compiler is checking method in parent class(Heroin) but runtime Child class object are crated hence child class methods are executed.

Example :-

```
class Animal
{void eat(){System.out.println("animal eat");}
};
class Dog extends Animal
{void eat(){System.out.println("Dog eat");}
};
class Cat extends Animal
{       void eat(){System.out.println("cat eat");}
};
class Test
{       public static void main(String[] args)
        {       Animal a1,a2;
```

```
                a1=new Dog();          //creates object of Dog class
                a1.eat();              //compiletime:Animal runtime : Dog
                a2=new Cat();          //creates object of Cat class
                a2.eat();              //compiletime:Animal runtime : Cat
        }
};
```

Example-:-method overriding vs. multilevel inheritance.

```
class Person
{       void eat(){System.out.println("Person eat");}
};
class Ratan extends Person
{       void eat(){System.out.println("Ratan eat");}
};
class RatanKid extends Ratan
{       void eat(){System.out.println("RatanKid eat");}
};
class Test
{       public static void main(String[] args)
        {       Person  pp = new Person();     //[creates object of Person class]
                pp.eat();
                Person p = new Ratan();//[creates object of Ratan class]
                p.eat();                //compile time: Person  runtime:Ratan
                Person p1 = new RatanKid();//[creates object of RatanKid  class]
                p1.eat();               //compile time: Person  runtime:RatanKid
                Ratan r = new RatanKid();//[creates object of RatanKid  class]
                r.eat();                //compile time: Ratan  runtime:RatanKid
        }
};
```

Example:-in java it is possible to override methods in child classes but it is not possible to override variables in child classes.

```
class Parent
{       int a=100;
};
class Child extends Parent
{       int a=1000;
        public static void main(String[] args)
        {       Parent p = new Child();
                System.out.println("a vlues is :--->"+p.a);//100
                Child c = (Child)p;
                System.out.println("a vlues is :--->"+c.a);//1000
        }
};
```

Method overloading:-

1) Method name same & parameters must be different.
   a. Void m1 (int a) { }
   b. Void m1(int a,int b) { }

2) To achieve overloading one java class sufficient.
3) It is also known as Compile time polymorphism/static binding/early binding.

Method overriding :-

1) Method name same & parameters must be same.
   a. Void m1(int a){ } //parent class method
   b. Void m1(int a){ } //child class method
2) To achieve overriding we required two java classes with parent and child relationship.
3) It is also known as runtime polymorphism/dynamic binding/late binding.

Example :- overriding vs method hiding

- static method cannot be overridden because static method bounded with class where as instance methods are bounded with object.
- In java it is possible to override only instance methods but not static methods.
- The below example seems to be overriding but it is method **hiding concept.**

```
class Parent
{       static void m1(){System.out.println("parent m1()");}
};
class Child extends Parent
{       static void m1(){System.out.println("child m1()");}
        public static void main(String[] args)
        {       Parent p = new Child();
                p.m1();//output : parent m1()
        }
};
```

toString():-

- toString() method present in Object and it is printing String representation of Object.
- toString() method return type is String object it means toString() method is returning String object.
- The toString() method is overridden some classes check the below implementation.
    - In String class toString() is overridden to return content of String object.
    - In StingBuffer class toString() is overridden to returns content of StringBuffer class.
    - In Wrapper classes(Integer,Byte,Character…etc) toString is overridden to returns content of Wrapper classes.

internal implementation:-

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

Example:-

Note :- whenever you are printing reference variable internally toString() method is called.

```
        Test t = new Test();    //creates object of Test class reference variable is "t"
        //the below two lines are same.
        System.out.println(t);
        System.out.println(t.toString());
class Test
{       public static void main(String[] args)
        {       Test t = new Test();
                System.out.println(t);
                System.out.println(t.toString()); // [Object class toString() executed]
        }
};
```

Example -2:-

toString() method present in Object class but in our Test class we are overriding toString() method hence our class toString() method is executed.

```
class Test
{       //overriding toString() method
        public String toString()
        {       return "ratansoft";
        }
        public static void main(String[] args)
        {       Test t = new Test();
                //below two lines are same
                System.out.println(t);          //Test class toString() executed
```

```
                System.out.println(t.toString());  //Test class toString() executed
        }
};
```

Example-3:- very important

```
class Student
{       //instance variables
        String sname;
        int sid;
        Student(String sname,int sid) //local variables
        {       //conversion
                this.sname = sname;
                this.sid = sid;
        }
        public String toString()  //overriding toString() method
        {       return "student name:-->"+sname+"  student id:-->"+sid;
        }
};
class TestDemo
{       public static void main(String[] args)
        {       Student s1 = new Student("ratan",111);
                //below two lines are same
                System.out.println(s1);             //student class toString() executed
                System.out.println(s1.toString()); //student class toString() executed

                Student s2 = new Student("anu",222);
                //below two lines are same
                System.out.println(s2);             //student class toString() executed
                System.out.println(s2.toString()); //student class toString() executed
        }
};
```

Example :-overriding of toString() method

```
class Test
{       //overriding method
        public String toString()
        {       return "ratnsoft";      }
        public static void main(String[] args)
        {       Test t = new Test();
                System.out.println(t);
                System.out.println(t.toString()); // [here overriding toString() executed it means
                our class toString()  method will be executed]
        }
};
```

In above example overriding toString() method will be executed.

Example :- employee class is not overriding toString()

```
class Employee
{       //instance variables
        String ename;
        int eid ;
          double esal;
        Employee(String ename,int eid,double esal)  //local variables
        {       //conversion of local variables to instance variables
                this.ename = ename;
                this.eid = eid;
                this.esal = esal;
        }
        public static void main(String[] args)
        {       Employee e1 = new Employee("ratan",111,60000);
        //whenever we are printing reference variables internally it calls toString() method
        System.out.println(e1); //e1.toString() [our class toString() executed output printed]
        }
};
```

D:\morn11>java Employee
Employee@530daa

*In above example Employee class is not overriding toString() method so parent class**(Object)** toString() method will be executed it returns hash code of the object.*

Example :- Employee class overriding toString() method

```
class Employee
{       //instance variables
        String ename;
        int eid ;
double esal;
        Employee(String ename,int eid,double esal)//local variables
```

```
{       //conversion of local variables to instance variables
        this.ename = ename;
        this.eid = eid;
        this.esal = esal;
}
public String toString()
{       return ename+" "+eid+" "+esal;
}
public static void main(String[] args)
{       Employee e1 = new Employee("ratan",111,60000);
        Employee e2 = new Employee("aruna",222,70000);
        Employee e3 = new Employee("nandu",222,80000);
//whenever we are printing reference variables internally it calls toString() method
System.out.println(e1);//e1.toString() [our class toString() executed output printed]
System.out.println(e2);//e2.toString() [our class toString() executed output printed]
System.out.println(e3);//e3.toString() [our class toString() executed output printed]
}
};
```

In above example when you print reference variables it is executing toString() hence Employee values will be printed.

Final modifier:-

1) Final is the modifier applicable for classes, methods and variables (for all instance, Static and local variables).

Case 1:-

1) if a class is declared as final, then we cannot inherit that class it means we cannot create child class for that final class.
2) Every method present inside a final class is always final but every variable present inside the final class not be final variable.

Example :-

```
final class Parent   //parent is final class child class creation not possible
{       };
class Child extends Parent  //compilation error
{       };
```

Example :-

Note :- Every method present inside a final class is always final but every variable present inside the final class not be final variable.

```
final class Test
{       int a=10;    //not a final variable
        void  m1()   //final method
        {       System.out.println("m1 method is final");
                a=a+100;
                System.out.println(a);  //110
        }
        public static void main(String[] args)
        {       Test t=new Test();
                t.m1();
        }
}
```

Case 2:-

If a method declared as a final we can not override that method in child class.

Example :-

```
class Parent
{       final void marry(){}        //overridden method is final
};
class Child extends Parent
{       void marry(){}              //overriding method
};
```

Compilation Error:- marry() in Child cannot override marry() in Parent overridden method is final

Case 3:-

1) *If a variable declared as a final we can not reassign that variable if we are trying to reassign compiler generate error message.*
2) *For the local variables only one modifier is applicable that is final.*

Example:-

```
class Test
{       public static void main(String[] args)
        {       final int a=100;//local variables
                a = a+100; // [compilation error because trying to reassignment]
                System.out.println(a);
        }
};
```

Compilation Error :- cannot assign a value to final variable a

Example :-

```
class Parent
{       void m1(){}
};
class Child extends Parent
{       int m1(){}
```

```
};
D:\morn11>javac Test.java
m1() in Child cannot override m1() in Parent
return type int is not compatible with void
```

Advantage of final modifier :-

The main advantage of final modifier is we can achieve security as no one can be allowed to change our implementation.

Disadvantage of final modifier:-

But the main disadvantage of final keyword is we are missing key benefits of Oops like inheritance and polymorphism. Hence is there is no specific requirement never recommended to use final modifier.

Garbage Collector

- Garbage collector is destroying the useless object and it is a part of the JVM.
- To make eligible objects to the garbage collector

Example-1 :-

Whenever we are assigning null constants to our objects then objects are eligible for GC(garbage collector)

```
class Test
{	public static void main(String[] args)
	{	Test t1=new Test();
		Test t2=new Test();
		System.out.println(t1);
		System.out.println(t2);
		;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
		t1=null;		//t1 object is eligible for Garbage collector
		t2=null;		//t2 object is eligible for Garbage Collector
		System.out.println(t1);
		System.out.println(t2);
	}
};
```

Example-2 :-

Whenever we reassign the reference variable the objects are automatically eligible for garbage collector.

```
class Test
{	public static void main(String[] args)
	{	Test t1=new Test();
		Test t2=new Test();
		System.out.println(t1);
		System.out.println(t2);
		t1=t2;			//reassign reference variable then one object is destroyed.
		System.out.println(t1);
		System.out.println(t2);
	}
};
```

Example -3:-

Whenever we are creating objects inside the methods one method is completed the objects are eligible for garbage collector.

```
class Test
{	void m1()
	{	Test t1=new Test();
		Test t2=new Test();
	}
	public static void main(String[] args)
	{	Test t=new Test();
		t.m1();
		System.gc();
```

```
        }
};
class Test
{       //overriding finalize()
        public void finalize()
        {       System.out.println("ratan sir object is destroyed");
                System.out.println(10/0);
        }
        public static void main(String[] args)
        {       Test t1 = new Test();
                Test t2 = new Test();
                ;;;;;;;//usage of objects
                t1=null;        //this object is eligible to Gc
                t2=null;        //this object is eligible to Gc
                System.gc();    //calling GarbageCollector
        }
}

//import java.lang.System;
import static java.lang.System.*;
class Test extends Object
{       public void finalize()
        {System.out.println("object destroyed");
        }
        public static void main(String[] args)
        {       Test t1 = new Test();
                Test t2 = new Test();
                t1=null;
                t2=null;
                gc();   //static import
        }
};
```

Ex:- if the garbage collector is calling finalize method at that situation exception is raised such type of exception are ignored.

```
class Test
{       public void finalize()
        {
System.out.println("ratan sir destroyed");
                int a=10/0;
        }
        public static void main(String[] args)
        {       Test t1=new Test();
                Test t2=new Test();
                t1=t2;
                System.gc();
        }
};
```

Ex:- If user is calling finalize() method explicitly at that situation exception is raised.

```
class Test
{       public void finalize()
        {
System.out.println("ratan sir destroyed");
                int a=10/0;
        }
        public static void main(String[] args)
        {       Test t1=new Test();
                Test t2=new Test();
                t1=t2;
                t2.finalize();
        }
```

};

Abstraction:-

There are two types of methods in java

a. **Normal methods**
b. **Abstract methods**

Normal methods:- (component method/concrete method)

Normal method is a method which contains method declaration as well as method implementation.

Example:-

void m1() ***--->method declaration***
{ body; ***--->method implementation***
}

Abstract methods:-

1) The method which is having only method declaration but not method implementations such type of methods are called abstract Methods.
2) In java every abstract method must end with ";".

Example : - ***abstract void m1 (); ----→ method declaration***

Based on above representation of methods the classes are divided into two types

1) **Normal classes.**
2) **Abstract classes.**

Normal classes:-

Normal class is a ordinary java class it contains only normal methods if we are trying to declare at least one abstract method that class will become abstract class.

Example:-

class Test ***//normal class***
{ void m1(){body;} ***//normal method***
void m2(){body;} ***//normal method***
*void m3(){body;}****//normal method***
};

Abstract class:-

Abstract class is a javaclass which contains at least one abstract method(wrong definition).

If any abstract method inside the class that class must be abstract.

Example 1:-

```
class Test //abstract class
{
void m1(){}//normal method
void m2(){}//normal method
void m3();//abstract method
};
```

Example-2:-

```
class Test  //abstract class
{
abstract  void m1();//abstract method
abstract  void m2();//abstract method
abstractvoid m3();//abstract method
};
```

Abstract modifier:-

- Abstract modifier is applicable for methods and classes but not for variables.
- To represent particular class is abstract class and particular method is abstract method to the compiler use abstract modifier.
- The abstract class contains declaration of methods it says abstract class partially implement class hence for partially implemented classes object creation is not possible. If we are trying to create object of abstract class compiler generate error message "class is abstract con not be instantiated"

Example -1:-

- ***Abstract classes are partially implemented classes hence object creation is not possible.***
- ***For the abstract classes object creation not possible, if you are trying to create object compiler will generate error message.***

```
abstract  class  Test              //abstract class
{       abstract  void m1();      //abstract method
        abstract void m2();       //abstract method
        abstract void m3();       //abstract method
        void m4(){System.out.println("m4 method");}  //normal method
        public static void main(String[] args)
        {       Test t = new Test();
                t.m4();
```

```
        }
};
```
Compilation error:- Test is abstract; cannot be instantiated
```
                Test t = new Test();
```

Example-2 :-

- **Abstract class contains abstract methods for that abstract methods provide the implementation in child classes.**
- **Provide the implementations is nothing but override the methods in child classes.**
- **The abstract class contains declarations but for that declarations implementation is present in child classes.**

```
abstract class Test
{       abstract void m1();
        abstract void m2();
        abstract void m3();
        void m4(){System.out.println("m4 method");}
};
class Test1 extends Test
{       void m1(){System.out.println("m1 method");}
        void m2(){System.out.println("m2 method");}
        void m3(){System.out.println("m3 method");}

        public static void main(String[] args)
        {       Test1 t = new Test1();
                t.m1();         t.m2();         t.m3();         t.m4();

                Test t1 = new Test1(); //abstract class reference variable Child class object
                t1.m1();        //compile : Test   runtime  : Test1
                t1.m2();        //compile : Test   runtime  : Test1
                t1.m3() ;       //compile : Test   runtime  : Test1
                t1.m4() ;       //compile : Test   runtime  : Test1
        }
};
```

Example -3 :-

- **Abstract class contains abstract methods for that abstract methods provide the implementation in child classes.**
- **if the child class is unable to provide the implementation of all parent class abstract methods at that situation declare that class with abstractmodifier then take one more child class to complete the implementation of remaining abstract methods.**
- **It is possible to declare multiple child classes but at final complete the implementation of all methods.**

```
abstract class Test
{       abstract void m1();
        abstract void m2();
```

```
        abstract void m3();
        void m4(){System.out.println("m4 method");}
};
abstract class Test1 extends Test
{       void m1(){System.out.println("m1 method");}
};
abstract class Test2 extends Test1
{       void m2(){System.out.println("m2 method");}
};
class Test3 extends Test2
{       void m3(){System.out.println("m3 method");}
        public static void main(String[] args)
        {       Test3 t = new Test3();
                t.m1();
                t.m2();
                t.m3();
                t.m4();
        }
};
```

Example :- *inside the abstract class it is possible to declare*

```
abstract class Test
{       public int a=10;
        public final int b=20;
        public static final int c=30;
        void disp1()
        {       System.out.println("a value is="+a);
        }
};
class Test1 extends Test
{       void disp2()
        {       System.out.println("b value is="+b);
                System.out.println("c value is="+c);
        }
        public static void main(String[] args)
        {       Test1 t = new Test1();
                t.disp1();
                t.disp2();
        }
};
```

Example-5:-

for the abstract methods it is possible to provide any return type(void, int, char,Boolean…..etc)

```
class Emp{};
abstract class Test1
```

```
{       abstract int m1(char ch);
        abstract boolean m2(int a);
        abstract Emp m3();
}
abstract class Test2 extends Test1
{       int m1(char ch)
        {       System.out.println("char value is:-"+ch);
                return 100;
        }
};
class Test3 extends Test2
{       boolean m2(int a)
        {       System.out.println("int value is:-"+a);
                return true;
        }
        Emp m3()
        {       System.out.println("m3 method");
                return new Emp();
        }
        public static void main(String[] args)
        {       Test3 t=new Test3();
                int a=t.m1('a');
                System.out.println("m1() return value is:-"+a);
                boolean b=t.m2(111);
                System.out.println("m2() return value is:-"+b);
                Emp e = t.m3();
                System.out.println("m3() return value is:-"+e);
        }
};
```


Example-6:-

It is possible to override non-abstract as a abstract method in child class.

```
abstract class Test
{   abstract void m1();                          //m1()  abstract method
    void m2(){System.out.println("m2 method");}  //m2() normal method
};
abstract class Test1 extends Test
{   void m1(){System.out.println("m1 method");}  //m1() normal method
```

```
    abstract void m2();                    //m2() abstract method
};
class FinalClass  extends Test1
{   void m2(){System.out.println("FinalClass m2() method");}
    public static void main(String[] args)
    {       FinalClass f = new FinalClass();
            f.m1();
            f.m2();
    }
};
```

Example:-

```
abstract class Test
{       public static void main(String[] args)
        {       System.out.println("this is abstract class main");
        }
};
```

Example-8:-

- *Constructor is used to create object (wrong definition).*
- Constructor is executed during object creation to initialize values to instance variables.
- Constructors are used to write the write the functionality that functionality executed during object creation.
- There are multiple ways to crate object in java but if we are crating object by using “new” then only constructor executed.

Note :- in below example abstract class constructor is executed but object is not created.

```
abstract class Test
{       Test()
        {       System.out.println("abstrac calss con");
        }
};
class Test1 extends Test
{       Test1()
        {       super();
```

```
                System.out.println("normal class con");
        }
        public static void main(String[] args)
        {       new Test1();
        }
};
D:\>java Test1
abstrac calss con
normal class con
case 1:- [abstract method to normal method]
abstract class Test
{       abstract void m1();
};
class Test1 extends Test
{       void m1(){System.out.println("m1 method");}
};


case 2:-[normal method to abstract method]
class Test
{       void m1(){System.out.println("m1 method");}
};
abstract class Test1 extends Test
{       abstract void m1();
};
```

Example-9:-the abstract class allows zero number of abstract method.

Definition of abstract class:-

Abstract class may contains abstract methods or may not contains abstract methods but object creation is not possible. The below example contains zero number of abstract methods.

Ex:- HttpServlet (doesn't contains abstract methods still it is abstract object creation not possible)

```
abstract class Test
{       void cm()       {       System.out.println("ratan");      }
        void pm()       {       System.out.println("anushka");  }
        public static void main(String[] args)
        {       Test t = new Test();
                t.cm();   t.pm();
        }
};
Test.java:6: Test is abstract; cannot be instantiated
```

Abstraction definition :-

- The process highlighting the set of services and hiding the internal implementation is called abstraction.
- Bank ATM Screens Hiding the internal implementation and highlighting set of services like , money transfer, mobile registration,…etc).

- Syllabus copy of institute just highlighting the contents of java but implementation there in classed rooms .
- We are achieving abstraction concept by using Abstract classes &Interfaces.

Encapsulation:-

The process of binding the data and code as a single unit is called encapsulation.
We are able to provide more encapsulation by taking the private data(variables) members.
To get and set the values from private members use getters and setters to set the data and to get the data.

Example:-

```
class Encapsulation
{       private int sid;
        private int sname;
        //mutator methods
        public void setSid(int x)
        {       this.sid=sid;    }
        public void setSname(String sname)
        {       this.sname=sname;    }
        //Accessor methods
        public int getSid()
        {       return sid;      }
        public String getSname()
        {       return  sname;          }
};
```

To access encapsulated use fallowing code:-

```
class Test
{       public static void main(String[] args)
        {       Encapsulation e=new Encapsulation();
                e.setSid(100);
                e.setSname("ratan");
                System.out.println(e.getSid());
                System.out.println(e.getSname());
        }
};
```

Main Method:-

Public static void main(String[] args)

Public ---→To provide access permission to the jvm declare main with public.

Static ---→To provide direct access permission to the jvm declare main is static(with out creation of object able to access main method)

Void ---→ don't return any values to the JVM.

String[] args ---→ used to take command line arguments(the arguments passed from command prompt)

String ---→ it is possible to take any type of argument.

[ ] ---→represent possible to take any number of arguments.

Modifications on main():-

1) Modifiers order is not important it means it is possible to take **public static** or **static public .**
   Example :- ***public static void main(String[] args)***
   static public void main(String[] args)
2) the fallowing declarations are valid
   string[] args String []args String args[]
   Example:- ***static public void main(String[] args)***
   static public void main(String []args)
   static public void main(String args[])
3) instead of args it is possible to takeany variable name (a,b,c,… etc)
   Example:- ***static public void main(String... ratan)***
   static public void main(String... a)
   static public void main(String... anushka)
4) for 1.5 version insted of String[] argsit is possible to take String… args(only three dots represent variable argument)
   Example:- ***static public void main(String... args)***
5) the applicable modifiers on main method.
   a. public b. static c. final d.strictfp e.synchronized
   in above five modifiers public and static mandatory remaining three modifiers optional.
   Example :- ***public static final strictfp synchronized void main(String... anushka)***

Which of the fallowing declarations are valid:-

1. **public static void main(String... a)** --->valid
2. **final strictfp static void mian(String[] Sravya)** --->invalid
3. **static public void mian(String a[])** --->valid
4. **final strictfp public static main(String[] rattaiah)** --->invalid
5. **final strictfp synchronized public static void main(String... nandu)**--->valid

Example-1:-

```
class Test
{       final strictfp synchronized static public  void main(String...ratan)
        {
                System.out.println("hello ratan sir");
        }
};
```

Example-2:-

```
class Test1
{       final strictfp synchronized static public  void main(String...ratan)
        {       System.out.println("Test-1");
        }
};
class Test2
{       final strictfp synchronized static public  void main(String...ratan)
        {       System.out.println("Test-2");
        }
};
class Test3
{       final strictfp synchronized static public  void main(String...ratan)
        {System.out.println("Test-3");
        }
};
```

In above two example execute all [.class] files to check the output.

Example-3:-*main method VS inheritance*

```
class Parent
{       public static void main(String[] args)
        {
        System.out.println("parent class");
        }
};
class Child extends Parent
{       public static void main(String[] args)
        {
        System.out.println("child class");
        }
};
```

```
class Parent
{       public static void main(String[] args)
        {
        System.out.println("parent class");
        }
};
class Child extends Parent
{
};
```

In above two examples execute both Parent and Child [.class] files to check the output.

Example-4:-*main method VS overloading*

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

```
        {
        System.out.println("String[] parameter main method start");
        main(100);//1-argument ( int )  method calling
        }
        public static void main(int a)
        {
        main('r'); //1-argument ( char )  method calling
        System.out.println("int  main method-→"+a);
        }
        public static void main(char ch)
        {
        System.out.println("char main method-→"+ch);
        }
}
```

Strictfp modifier:-

- **a.** Strictfp is a modifier applicable for classes and methods.
- b. If a method is declared as strictfp all floating point calculations in that method will followIEEE754 standard. So that we will get platform independent results.
- c. If a class is declared as stictfp then every method in that class will follow IEEE754standard so we will get platform independent results.

Ex:- **strictfp class Test{ //methods///}** **--->all methods fallows IEEE754**
strictfp void m1(){} **---> m1() method fallows IEE754**

Native modifier:-

- a. Native is the modifier applicable only for methods.
- b. Native method is used to represent particular method implementations there in non-java code (other languages like C,CPP) .
- c. Nativemethods are also known as “foreign methods”.

Examples:-

```
public final int getPriority();
public final void setName(java.lang.String);
public static native java.lang.Thread currentThread();
public static native void yield();
```

Command Line Arguments:-

The arguments which are passed from command prompt is called command line arguments. We are passing command line arguments at the time program execution.

Example-1 :-

```
class Test
{
        public static void main(String[] ratan)
        {       System.out.println(ratan[0]+"  "+ratan[1]);//printing command line arguments
                System.out.println(ratan[0]+ratan[1]);
                //conversion of  String-int  String-double
                int a = Integer.parseInt(ratan[0]);
                double d = Double.parseDouble(ratan[1]);
                System.out.println(a+d);
        }
};
```

D:\>java Test 100 200
100 200
100200
300.0

Example-2:-

To provide the command line arguments with spaces then take that command line argument with in double quotes.

```
class Test
{
        public static void main(String[] ratan)
        {       //printing commandline arguments
                System.out.println(ratan[0]);
                System.out.println(ratan[1]);
        }
};
```

D:\>java Test *corejava ratan*
corejava
ratan
D:\>java Test *core java ratan*
core
java
D:\>java Test *"core java" ratan*
core java
ratan

Var-arg method:-

1. introduced in 1.5 version.
2. it allows the methods to take any number of parameters.

Syntax:-(only 3 dots)

```
Void m1(int… a)
```

The above m1() is allows any number of parameters.(0 or 1 or 2 or 3…….)

Example-1:-

class Test

```
{
        void m1(int... a){      System.out.println("Ratan");    }//var-arg method
        public static void main(String[] args)
        {
                Test t=new Test();
                t.m1();  //int var-arg method executed
                t.m1(10);//int var-arg method executed
                t.m1(10,20);//int var-arg method executed
        }
}
```

Example-2:-

```
class Student
{       void classRoom(int... fee)      {System.out.println("class room --> B.tech --> CSE");     }
        public static void main(String[] ratan)
        {
                Student s = new Student();
                s.classRoom();          //scholarship students
                s.classRoom(30000); //counselling fee students
                s.classRoom(100000,30000);     //NRI student with donation + counselling fee
                s.classRoom(100000,30000,40000);//NRI student donation+mediator fee+counselling
        }
}
```

Example-3:-printing var-arg values

```
class Test
{       void m1(int... a)
        {       System.out.println("Ratan");
                for (int a1:a)
                {System.out.println(a1);        }
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                t.m1();                 //int var-arg method executed
                t.m1(10);               //int var-arg method executed
                t.m1(10,20);            //int var-arg method executed
                t.m1(10,20,30,40);      //int var-arg method executed
        }
}
```

Example-4:-var-arg VS normal arguments

```
class Test
{       void m1(int a,double d,char ch,String... str)
        {       System.out.println(a+"  "+d+"   "+ch); //printing normal arguments
                for (String str1:str)//printing var-arg  by using for-each loop
                {       System.out.println(str1);
                }
        }
        public static void main(String... args)
```

```
        {       Test t=new Test();
                t.m1(10,20.5,'s');
                t.m1(10,20.5,'s',"aaaa");
                t.m1(10,20.5,'s',"aaaa","bbb");
        }
};
```

Note :-inside the method it is possible to declare only one variable-argument and that must be last argument otherwise the compiler will generate compilation error.

void m1(int... a) --->valid
void m2(int... a,char ch) --->invalid
void m3(int... a,boolean... b) --->invalid
void m4(double d,int... a) --->valid
void m5(char ch ,double d,int... a) --->valid
void m6(char ch ,int... a,boolean... b) --->invalid

Example-5 :- var-arg method vs overloading

```
class Test
{       void m1(int... a)
        {       for (int a1:a)
                {       System.out.println(a1);         }
        }
        void m1(String... str)
        {       for (String str1:str)
                {       System.out.println(str1);               }
        }
        public static void main(String[] args)
        {       Test t=new Test();
                t.m1(10,20,30);         //int var-arg method calling
                t.m1("ratan","Sravya");//String var-arg calling
                t.m1();//var-arg method vs ambiguity  [compilation error  ambiguous]
        }
}
```