Example: Class

```
// Class Declaration

public class Dog
{
        // Instance Variables
        String name;
        String breed;
        int age;
        String color;

// constructor used when no dimensions specified
        Dog()
        {
                Name=”ABC”;
                breed =”CGF”;
                age=5;
                color= “red”;
        }

 // constructor used when all dimensions specified
        Dog(String a, String b, int c, String d)
        {
                Name=a;
                breed =b;
                age=c;
                color= d;
        }



        // Constructor Declaration of Class
        public Dog(String name, String breed, int age, String color)
        {
                this.name = name;
                this.breed = breed;
                this.age = age;
                this.color = color;
        }

        // method 1
Public string setname(String n)
{name=n;
}
        public String getName()
        {
                return name;
	}

	// method 2
	public String getBreed()
	{
		return breed;
	}

	// method 3
	public int getAge()
	{
		return age;
	}

	// method 4
	public String getColor()
	{
		return color;
	}

	@Override
	public String toString()
	{
		return("Hi my name is "+ this.getName()+ ".\nMy breed,age and color are " +
			this.getBreed()+"," + this.getAge()+  ","+ this.getColor());
	}

	public static void main(String[] args)
	{
		Dog D = new Dog();// default
		Dog D1 = new Dog(“sdfs”, “dfgf”, 4, “sdfsdf”);// parameterised
		Dog D2 = new Dog("tuffy","papillon", 5, "white");
		Dog D3 = new Dog(D2); // copy of constructed
		Dog D4 = new Dog(), D5= new Dog(); //creating two objects
		System.out.println(D3.toString());
		System.out.println(D3.name);
	}
}
```

Program

```
// Class with methods
class Box {
double width;
double height;
double depth;
// display volume of a box
double volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}

class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
Double a=mybox1.volume();
// display volume of second box
Double b= mybox2.volume();
SOP(a);
SOP(b);
}
}

//Anonymous.java
class Calculation{

void fact(int n){
int fact=1;
for(int i=1;i<=n;i++){
fact=fact*i;
}
System.out.println("factorial is "+fact);
}

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

```
//Calculation c= new Calculation();
new Calculation().fact(5);//calling method with anonymous object
 }
}


// constructor overloading
class Demo{
      int  value1;
      int  value2;
      /*Demo(){
       value1 = 10;
       value2 = 20;
       System.out.println("Inside 1st Constructor");
     }*/

Demo(int a){
      value1 = a;
      System.out.println("Inside 2nd Constructor");
    }

    Demo(int a,int b){
    value1 = a;
    value2 = b;
    System.out.println("Inside 3rd Constructor");
   }

   public void display(){
      System.out.println("Value1 === "+value1);
      System.out.println("Value2 === "+value2);
  }
  public static void main(String args[]){
    Demo d1 = new Demo();
    Demo d2 = new Demo(30);
    Demo d3 = new Demo(30,40);
    d1.display();
    d2.display();
    d3.display();
 }
}
```

```
// constructor chaining: Consider a scenario where a base class is extended by a child. Whenever
an object of the child class is created, the constructor of the parent class is invoked first. This is
called Constructor chaining.
class Demo{
   int  value1;
   int  value2;
    Demo(){
      value1 = 1;
      value2 = 2;
      System.out.println("Inside 1st Parent Constructor");
   }
   Demo(int a, int b){
      value1 = a;
            value2=b;
      System.out.println("Inside 2nd Parent Constructor");
   }
  public void display(){
     System.out.println("Value1 === "+value1);
     System.out.println("Value2 === "+value2);
  }
  public static void main(String args[]){
     DemoChild d1 = new DemoChild();
     d1.display();
  }
}
class DemoChild extends Demo{
    int value3;
    int value4;
    DemoChild(){
    super(1,2);
     value3 = 3;
     value4 = 4;
    System.out.println("Inside the Constructor of Child");
    }
    public void display(){
      System.out.println("Value1 === "+value1);
      System.out.println("Value2 === "+value2);
      System.out.println("Value1 === "+value3);
      System.out.println("Value2 === "+value4);
   }
}


//copy constructor
//Java program to initialize the values from one object to another object.
class Student6{
    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i,String n){
    id = i;
    name = n;
    }
    //constructor to initialize another object
    Student6 (Student6 s){
    this.id = s.id;
    this.name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

++++++++++++++++++++++++++++++++++++

```
// Java program to read some values using Scanner class and print their mean.
import java.util.Scanner;
 public class ScannerDemo2
{
    public static void main(String[] args)
    {
        // Declare an object and initialize with predefined standard input object
        Scanner sc = new Scanner(System.in);
         // Initialize sum and count of input elements
        int sum = 0, count = 0;
         // Check if an int value is available
        while (sc.hasNextInt())
        {
            // Read an int value
            int num = sc.nextInt();
            sum += num;
            count++;
        }
        int mean = sum / count;
        System.out.println("Mean: " + mean);
    }
}
```

Example 3

```
// bufferReader for reading file
import java.io.*;
public class BufferedReaderExample {
   public static void main(String args[])throws Exception{
         FileReader fr=new FileReader("D:\\testout.txt");
         BufferedReader br=new BufferedReader(fr);
         int i;
         while((i=br.read())!=-1){
         System.out.print((char)i);
         }
         br.close();
         fr.close();
   }}
```

Example 4

```
public static void main(String[] args) {
   String input = null;
   int number = 0;
   try {
    BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));
       input = bufferedReader.readLine();
       number = Integer.parseInt(input);
   } catch (NumberFormatException ex) {
      System.out.println("Not a number !");
   } catch (IOException e) {
       e.printStackTrace();
}}
```

Example 5

```
// bufferereader for inputstream reader
import java.io.*;
public class UserInputInteger
{
       public static void main(String args[])throws IOException
       {
       InputStreamReader read = new InputStreamReader(System.in);
       BufferedReader in = new BufferedReader(read);
       int number;
               System.out.println("Enter the number");
               number = Integer.parseInt(in.readLine());
   }
}
```

Example 6

```
//reading and printing the data until the user prints stop using bufferereader
public class BufferedReaderExample{
public static void main(String args[])throws Exception{
     InputStreamReader r=new InputStreamReader(System.in);
     BufferedReader br=new BufferedReader(r);
     String name="";
     while(!name.equals("stop")){
      System.out.println("Enter data: ");
      name=br.readLine();
      System.out.println("data is: "+name);
     }
    br.close();
    r.close();
    }      }
```

Example 7

```
//scope of the prog
class Scope {
public static void main(String args[]) {
int x; // known to all code within main
x = 10; // global
if(x == 10)
{ // start new scope
int y = 20; // known only to this block // local var
// x and y both known here.
System.out.println("x and y: " + x + " " + y);
x = y * 2;
}
// y = 100; // Error! y not known here
// x is still known here.
System.out.println("x is " + x);
}}
```

Scope of the variable

```
class LifeTime {
public static void main(String args[]) {
int x;
for(x = 0; x < 3; x++) {
int y = -1; // y is initialized each time block is entered
System.out.println("y is: " + y); // this always prints -1
y = 100;
System.out.println("y is now: " + y);
}
}
}
```

Example 7

```
//Scope error
class ScopeErr {
public static void main(String args[]) {
int bar = 1;
{ // creates a new scope
int bar = 2; // Compile-time error – bar already defined!
}
}
}
```

```
// Matrixarray.java
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};
//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];
//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}
}}
```

```
// Passarray as input to method
class Test
{
// Driver method
public static void main(String args[])
{
int arr[] = {3, 1, 2, 5, 4};
// passing array to method m1
sum(arr);
}
public static void sum(int[] arr)
{
// getting sum of array values
int sum = 0;
for (int i = 0; i < arr.length; i++)
sum+=arr[i];
System.out.println("sum of array values : " + sum);
}
}

// Return array as output
class Test
{
// Driver method
public static void main(String args[])
{
int arr[] = m1();
for (int i = 0; i < arr.length; i++)
System.out.print(arr[i]+" ");
}
public static int[] m1()
{
// returning array
return new int[]{1,2,3};
}
}
```

Abstract

```
abstract class Bike{
  abstract void run();
}
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();
}
}
```

Abstrct Class example

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
      Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape
      () method
      s.draw();
      }
      }


//Simple Inheritance
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}

/// single Inheritance
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}

// multilevel inheritance
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
```

```
d.weep();
d.bark();
d.eat();
}}

/// hierarchical inheritance
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
*}}


// Aggregation in Java Has a relationship
public class Address {
String city, state, country;

public Address (String city, String state, String country) {
   this.city = city;
   this.state = state;
   this.country = country;
}
 }
public class Emp {
int id;
String name;
Address address;

public Emp(int id, String name,Address address) {
   this.id = id;
   this.name = name;
   this.address=address;
}

void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}

public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");

Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);

e.display();
e2.display();

}
}

// aggregation in java
class Operation{
 int square(int n){
  return n*n;
 }
}

class Circle{
 Operation op;//aggregation
 double pi=3.14;

 double area(int radius){
   op=new Operation();
   int rsquare=op.square(radius);//code reusability (i.e. delegates the method call).
   return pi*rsquare;
 }

 public static void main(String args[]){
   Circle c=new Circle();
   double result=c.area(5);
   System.out.println(result);
 }
}

// Method overloading Polymorphism

// 1) Method Overloading: changing no. of arguments
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
```

```
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}

// 2) Method Overloading: changing data type of arguments
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}

// In java, method overloading is not possible by changing the return type of the method only
because of ambiguity.

class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

Output: Compile Time Error: method add(int,int) is already defined in class Adder

```
// Can we overload java main() method?
class TestOverloading4{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
// But JVM calls main() method which receives string array as arguments only.
```

Method Overloading and Type Promotion

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);

  }
}


//Example of Method Overloading with Type Promotion if matching found
//If there are matching type arguments in the method, type promotion is not performed.

class OverloadingCalculation2{
  void sum(int a,int b){System.out.println("int arg method invoked");}
  void sum(long a,long b){System.out.println("long arg method invoked");}

  public static void main(String args[]){
  OverloadingCalculation2 obj=new OverloadingCalculation2();
  obj.sum(20,20);//now int arg sum() method gets invoked
  }
}

//Example of Method Overloading with Type Promotion in case of ambiguity
// If there are no matching type arguments in the method, and each method promotes similar
number of arguments, there will be ambiguity.

class OverloadingCalculation3{
  void sum(int a,long b){System.out.println("a method invoked");}
  void sum(long a,int b){System.out.println("b method invoked");}

  public static void main(String args[]){
  OverloadingCalculation3 obj=new OverloadingCalculation3();
  obj.sum(20,20);//now ambiguity
  }
}

// // Java program for Operator overloading Java does not support user-defined operator
overloading. The + operator can be used to as an arithmetic addition operator to add numbers. It
can also be used to concatenate strings.

class OperatorOVERDDN {

        void operator(String str1, String str2)
        {
                String s = str1 + str2;
                System.out.println("Concatinated String - "
                                                + s);
        }
```

```
	void operator(int a, int b)
	{
		int c = a + b;
		System.out.println("Sum = " + c);
	}
}

class Main {
	public static void main(String[] args)
	{
		OperatorOVERDDN obj = new OperatorOVERDDN();
		obj.operator(2, 3);
		obj.operator("joe", "now");
	}
}


// Method overloading in inheritance
class Animal {
  public void animalSound() {
    System.out.println("The animal makes a sound");
  }
}

class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class Dog extends Animal {
  public void animalSound() {
    System.out.println("The dog says: bow wow");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

// method overriding

```
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
  void run(){System.out.println("Bike is running safely");}
    public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```

// Java Runtime Polymorphism Example: Shape –method overriding

```
class Shape{
void draw(){System.out.println("drawing...");}
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle...");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle...");}
}
class Triangle extends Shape{
void draw(){System.out.println("drawing triangle...");}
}
class TestPolymorphism2{
public static void main(String args[]){
Shape s;
s=new Rectangle();
s.draw();
s=new Circle();
s.draw();
s=new Triangle();
s.draw();
}
}
```

Output:

```
drawing rectangle...
drawing circle...
drawing triangle...
```

// Java Runtime Polymorphism with Data Member

```
class Bike{
 int speedlimit=90;
}
class Honda3 extends Bike{
 int speedlimit=150;

 public static void main(String args[]){
  Bike obj=new Honda3();
  System.out.println(obj.speedlimit);//90
}

 // Abstract class
abstract class Animal {
  public abstract void animalSound();  // Abstract method (does not have a body)
  public void sleep()  // Regular method
 {
    System.out.println("Zzz");
  }
}
class Pig extends Animal {
  public void animalSound() {
    System.out.println("The pig says: wee wee");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

// abstract method in an abstract class

```
abstract class Sum{
   public abstract int sumOfTwo(int n1, int n2);
   public abstract int sumOfThree(int n1, int n2, int n3);

   public void disp(){
	System.out.println("Method of class Sum");
   }
}

class Demo extends Sum{

   public int sumOfTwo(int num1, int num2){
	return num1+num2;
```

```
   }
   public int sumOfThree(int num1, int num2, int num3){
         return num1+num2+num3;
   }
   public static void main(String args[]){
         Sum obj = new Demo();
         System.out.println(obj.sumOfTwo(3, 7));
         System.out.println(obj.sumOfThree(4, 3, 19));
         obj.disp();
   }
}
```

Output:

```
10
26
```

```
// abstract method in interface
interface Multiply{
   //abstract methods
   public abstract int multiplyTwo(int n1, int n2);
  int multiplyThree(int n1, int n2, int n3);
   }

class Demo implements Multiply{
   public int multiplyTwo(int num1, int num2){
      return num1*num2;
   }
   public int multiplyThree(int num1, int num2, int num3){
      return num1*num2*num3;
   }
   public static void main(String args[]){
      Multiply obj = new Demo();
      System.out.println(obj.multiplyTwo(3, 7));
      System.out.println(obj.multiplyThree(1, 9, 0));
   }
}
```

Output:

```
21
0
```

// a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

```
class Parent
{
    // private methods are not overridden
    private void m1() { System.out.println("From parent m1()");}
```

```
  protected void m2() { System.out.println("From parent m2()"); }
}
 class Child extends Parent
{
   // new m1() method unique to Child class
   private void m1() { System.out.println("From child m1()");}
    // overriding method with more accessibility
   @Override
   public void m2() { System.out.println("From child m2()");}
     }
class Main
{
   public static void main(String[] args)
   {
      Parent obj1 = new Parent();
      obj1.m2();
      Parent obj2 = new Child();
      obj2.m2();
   }
}
```
Output :

```
From parent m2()
From child m2()
```

// Final methods can not be overridden

```
class Parent
{
   // Can't be overridden
   final void show() {  }
}
 class Child extends Parent
{
   // This would produce error
   void show() {  }
}
```
Output :

```
13: error: show() in Child cannot override show() in Parent     void show() {  }
```

// Static methods can not be overridden(Method Overriding vs Method Hiding) : When you defines a static method with same signature as a static method in base class, it is known as method hiding.

```
class Parent
{
   // Static method in base class which will be hidden in subclass
   static void m1() { System.out.println("From parent static m1()");}
```

```
    // Non-static method which will be overridden in derived class
    void m2() { System.out.println("From parent non-static(instance) m2()"); }
}

class Child extends Parent
{
    // This method hides m1() in Parent
    static void m1() { System.out.println("From child static m1()");}

    // This method overrides m2() in Parent
    @Override
    public void m2() { System.out.println("From child non-static(instance) m2()");}

}

// Driver class
class Main
{
    public static void main(String[] args)
    {
        Parent obj1 = new Child();

        // As per overriding rules this should call to class Child static overridden method. Since
static method can not be overridden, it calls Parent's m1()
        obj1.m1();

        // Here overriding works and Child's m2() is called
        obj1.m2();
    }
}
Output :

From parent static m1()
From child non-static(instance) m2()

// Static blocks

class Test
{
    // static variable
    static int a = 10;
    static int b;

    // static block
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String[] args)
    {
       System.out.println("from main");
       System.out.println("Value of a : "+a);
       System.out.println("Value of b : "+b);
    }
}
```

Output:

```
Static block initialized.
from main
Value of a : 10
Value of b : 40
```

// Static variables

```
class Test
{
    // static variable
    static int a = m1();

    // static block
    static {
        System.out.println("Inside static block");
    }

    // static method
    static int m1() {
        System.out.println("from m1");
        return 20;
    }

    // static method(main !!)
    public static void main(String[] args)
    {
       System.out.println("Value of a : "+a);
       System.out.println("from main");
    }
}
```

Output:

```
from m1
Inside static block
Value of a : 20
from main
```

//final variable

// A final variable that is not initialized at the time of declaration is known as blank final variable. If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. E.g. PAN CARD number of an employee.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

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

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

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

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

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

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Is final method inherited?

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

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

//we initialize blank final variable but only in constructor. For example:

```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
 }
}
```

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

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

// Can we declare a constructor final: No, because constructor is never inherited.
// *static* indicates that the particular member belongs to a type itself, rather than to an instance of that type.
// From the memory perspective, static variables go in a particular pool in JVM memory called Metaspace

```
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }

//method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

```
    // static Method
    public static void setNumberOfCars(int numberOfCars) {
        Car.numberOfCars = numberOfCars;
    }
```

usage of java this keyword.

1. **this can be used to refer current class instance variable.**
2. **this can be used to invoke current class method (implicitly)**
3. **this() can be used to invoke current class constructor.**
4. **this can be passed as an argument in the method call.**
5. **this can be passed as argument in the constructor call.**
6. **this can be used to return the current class instance from the method.**

Program without this pointer

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
THIS.rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

```
o/p
0 null 0.0
0 null 0.0
```

// Program using this

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

// Calling default constructor from parameterized constructor:

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

```
A a=new A(10);
}}
```

// Calling parameterized constructor from default constructor:

```
class A{
A(){
this(5);
System.out.println("hello a");
}
A(int x){
System.out.println(x);
}
}
class TestThis6{
public static void main(String args[]){
A a=new A();
}}
```

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2{
  void m(S2 obj){
  System.out.println("method is invoked");
  }
  void p(){
  m(this);
  }
  public static void main(String args[]){
  S2 s1 = new S2();
  s1.p();
  }
}
```

// return as a statement from the method

```
class A{
A getA(){
return this;
}
void msg(){System.out.println("Hello java");}
}
class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
```

super with variables

```
class Vehicle
{
   int maxSpeed = 120;
}

/* sub class Car extending vehicle */
class Car extends Vehicle
{
   int maxSpeed = 180;

   void display()
   {
      /* print maxSpeed of base class (vehicle) */
      System.out.println("Maximum Speed: " + super.maxSpeed);
   }
}

/* Driver program to test */
class Test
{
   public static void main(String[] args)
   {
      Car small = new Car();
      small.display();
   }
}
```

super with methods

```
class Person
{
   void message()
   {
      System.out.println("This is person class");
   }
}

/* Subclass Student */
class Student extends Person
{
   void message()
   {
      System.out.println("This is student class");
   }
```

```
    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}

/* Driver program to test */
class Test
{
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

super with constructors

```
class Person
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

/* subclass Student extending the Person class */
class Student extends Person
{
    Student()
    {
        // invoke or call parent class constructor
        Super();
        System.out.println("Student class Constructor");
    }
}

/* Driver program to test*/
class Test
{
        public static void main(String[] args)
        {
            Student s = new Student();
        }
    }
```

// super with variables

```
class Vehicle
{
    int maxSpeed = 120;
}

class Car extends Vehicle
{
    int maxSpeed = 180;

    void display()
    {
            System.out.println("Maximum Speed: " + super.maxSpeed);
    }
}

class Test
{
    public static void main(String[] args)
    {
            Car small = new Car();
            small.display();
    }
}
```

// super with methods:

```
    class Person
    {
            void message()
            {
                    System.out.println("This is person class");
            }
    }

    class Student extends Person
    {
            void message()
            {
```

```
		System.out.println("This is student class");
	}

	// Note that display() is only in Student class
	void display()
	{
		// will invoke or call current class message() method
		message();

		// will invoke or call parent class message() method
		super.message();
	}
}

class Test
{
	public static void main(String args[])
	{
		Student s = new Student();
		// calling display() of Student
		s.display();
	}
}
```

// super with constructors

```
class Person
{
	Person()
	{
		System.out.println("Person class Constructor");
	}
}

class Student extends Person
{
	Student()
	{
		// invoke or call parent class constructor
		super();
		System.out.println("Student class Constructor");
	}
}

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

```
                    Student s = new Student();
          }
}
```

This is file **Protection.java**:

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **Derived.java**:

```
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = "4 + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **SamePackage.java**:

```
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}

package p2;
```

```
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **OtherPackage.java**:

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This show the value in the same package. So no error

This is file **Derived.java**:

```
package p1;
```

```
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = "4 + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is inheritance so o/p

This is file **SamePackage.java**:

```
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Same o/p if store in same package.

```
package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **OtherPackage.java**:

```
package p2;
```

```
//import p1.*;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}

package One;
public class Alpha {
   //member variables
   private   int iamprivate = 1;
             int iampackage = 2;  //package access
   protected int iamprotected = 3;
   public    int iampublic = 4;

   //methods
   private void privateMethod() {
      System.out.println("iamprivate Method");
   }
   void packageMethod() { //package access
      System.out.println("iampackage Method");
   }
   protected void protectedMethod() {
      System.out.println("iamprotected Method");
   }
   public void publicMethod() {
      System.out.println("iampublic Method");
   }

   public static void main(String[] args) {
      Alpha a = new Alpha();
      a.privateMethod();   //legal
      a.packageMethod();   //legal
      a.protectedMethod(); //legal
      a.publicMethod();    //legal

      System.out.println("iamprivate: " + a.iamprivate);    //legal
      System.out.println("iampackage: " + a.iampackage);    //legal
      System.out.println("iamprotected: "+ a.iamprotected"); //legal
```

```
        System.out.println("iampublic: " + a.iampublic);     //legal
    }
}
output
iamprivate Method
iampackage Method
iamprotected Method
iampublic Method
iamprivate: 1
iampackage: 2
iamprotected: 3
iampublic: 4
```

Package Access Level

```
package One;
public class DeltaOne {
    public static void main(String[] args) }
        Alpha a = new Alpha();
        //a.privateMethod();  //illegal
        a.packageMethod();    //legal
        a.protectedMethod();  //legal
        a.publicMethod();     //legal
        //System.out.println("iamprivate: " //  + a.iamprivate);   //illegal
        System.out.println("iampackage: "   + a.iampackage);   //legal
        System.out.println("iamprotected: " + a.iamprotected); //legal
        System.out.println("iampublic: "   + a.iampublic);    //legal
    }
}

output
iampackage Method
iamprotected Method
iampublic Method
iampackage: 2
iamprotected: 3
iampublic: 4
```

Subclass Access Level

```
package two;
import One.*;
public class AlphaTwo extends Alpha {
    public static void main(String[] args) {
        Alpha a = new Alpha();
        //a.privateMethod();   //illegal
        //a.packageMethod();   //illegal
        //a.protectedMethod(); //illegal
        a.publicMethod()       //legal

        //System.out.println("iamprivate: "  //   + a.iamprivate);  //illegal
        //System.out.println("iampackage: " //   + a.iampackage);   //illegal
        //System.out.println("iamprotected: "// + a.iamprotected);  //illegal
        System.out.println("iampublic "  + a.iampublic);      //legal

        AlphaTwo a2 = new AlphaTwo();
        a2.protectedMethod();   //legal
        System.out.println("iamprotected: " + a2.iamprotected); //legal
    }
}
```

```
o/p
iampublic Method
iampublic: 4
iamprotected Method
iamprotected: 3
```

World Access Level

```
package Two;
import One.*;
public class DeltaTwo {
    public static void main(String[] args) {
        Alpha alpha = new Alpha();
        //alpha.privateMethod();   //illegal
        //alpha.packageMethod();   //illegal
        //alpha.protectedMethod(); //illegal
        alpha.publicMethod();      //legal
        //System.out.println("iamprivate: " //   + a.iamprivate); //illegal
        //System.out.println("iampackage: " //   + a.iampackage); //illegal
        //System.out.println("iamprotected: "//   + a.iamprotected);//illegal
        System.out.println("iampublic: "  + a.iampublic);        //legal
    }
}
```

```
output
iampublic Method
iampublic: 4
```

1. Mypackage.java

```
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}

/* import 'MyClass' class from 'names' myPackage */

Package mypackage1;
import myPackage.MyClass;

public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable with a value
      String name = "GeeksforGeeks";
      // Creating an instance of class MyClass in the package.
      MyClass obj = new MyClass();
      obj.getNames(name);
   }
}
```

Std. package import

```
            import static java.lang.System.*;
            class StaticImportExample{
              public static void main(String args[]){

            system.out.println(“:sdfs”);// anonymous object access
               out.println("Hello");//Now no need of System.out
               out.println("Java");

             }
            }
```

```
            ClassOne.java
package package_name;

public class ClassOne {
public void methodClassOne() {
System.out.println("Hello there its ClassOne");
}
}
```

```
ClassTwo.java
package package_one;

public class ClassTwo {
public void methodClassTwo(){
System.out.println("Hello there i am ClassTwo");
}
}
```

```
Testing.java
import package_one.ClassTwo;
import package_name.ClassOne;

public class Testing {
public static void main(String[] args){
ClassTwo a = new ClassTwo();
ClassOne b = new ClassOne();
a.methodClassTwo();
b.methodClassOne();
}
}
```

Example of package that import the packagename.*

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

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Example of package by import package.classname

```
//save by A.java

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

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Example of package by import fully qualified name

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

Subpackage.java

```
package com.javatpoint.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

Pacakageclass.java

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

Package p=Package.getPackage("java.lang");

System.out.println("package name: "+p.getName());
System.out.println("Specification Title: "+p.getSpecificationTitle());
System.out.println("Specification Vendor: "+p.getSpecificationVendor());
System.out.println("Specification Version: "+p.getSpecificationVersion());

System.out.println("Implementaion Title: "+p.getImplementationTitle());
System.out.println("Implementation Vendor: "+p.getImplementationVendor());
System.out.println("Implementation Version: "+p.getImplementationVersion());
System.out.println("Is sealed: "+p.isSealed());
  }
}
```

1. **private access modifier.java**

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Privateconstructor.java

```
class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();//Compile Time Error
 }
}
```

Default.java

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

```
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

Protected.java

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

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Public.java

```
//save by A.java

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

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Javaaccessmodifier.java

```
class A{
protected void msg(){System.out.println("Hello java");}
}

public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
 public static void main(String args[]){
   Simple obj=new Simple();
   obj.msg();
   }
}
```

Interface.java

```
import java.io.*;

interface Vehicle {

	// all are the abstract methods.
	void changeGear(int a);
	void speedUp(int a);
	void applyBrakes(int a);
}

class Bicycle implements Vehicle{

	int speed;
	int gear;

	// to change gear
	@Override
	public void changeGear(int newGear){

		gear = newGear;
	}

	// to increase speed
	@Override
	public void speedUp(int increment){

		speed = speed + increment;
	}

	// to decrease speed
	@Override
	public void applyBrakes(int decrement){

		speed = speed - decrement;
	}
	public void printStates() {
		System.out.println("speed: " + speed
			+ " gear: " + gear);
	}
}

class Bike implements Vehicle {

	int speed;
	int gear;

	// to change gear
	@Override
	public void changeGear(int newGear){

		gear = newGear;
	}

	// to increase speed
	@Override
	public void speedUp(int increment){

		speed = speed + increment;
	}

	// to decrease speed
	@Override
	public void applyBrakes(int decrement){

		speed = speed - decrement;
	}

	public void printStates() {
		System.out.println("speed: " + speed
			+ " gear: " + gear);
	}

}

class GFG {

	public static void main (String[] args) {

		// creating an inatance of Bicycle
		// doing some operations
		Bicycle bicycle = new Bicycle();
		bicycle.changeGear(2);
```

```
                bicycle.speedUp(3);
                bicycle.applyBrakes(1);

                System.out.println("Bicycle present state :");
                bicycle.printStates();

                // creating instance of bike.
                Bike bike = new Bike();
                bike.changeGear(1);
                bike.speedUp(4);
                bike.applyBrakes(3);

                System.out.println("Bike present state :");
                bike.printStates();
        }
}
```

Inheritance in interface.java

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
```

Drawable	Drawable
Rectangle	Circle

```
}
```

TestInterface1.java

```
//Interface declaration: by first user
interface Drawable {
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
```

```
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
```

Multipleinheritance.java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Interface inheritance and Default Method in Interface

```
interface TestInterface
{
	// abstract method
	public void square(int a);

	// default method
	default void show()
	{
	System.out.println("Default Method Executed");
	}
}

class TestClass implements TestInterface
{
	// implementation of square abstract method
	public void square(int a)
	{
		System.out.println(a*a);
	}

	public static void main(String args[])
	{
		TestClass d = new TestClass();
		d.square(4);

		// default method executed
		d.show();
	}
}
```

Interface inheritance with static methods

```
interface Printable{
void print();
default void msg(){System.out.println("default method");}
static int cube(int x){return x*x*x;}
}
interface Showable extends Printable{
void show();
}
```

```
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
obj.msg();
//obj.cube(3);
System.out.println(Printable.cube(3));
 }
}
```

Static method in interface

```
interface Vehicle {

   default void print() {
      System.out.println("I am a vehicle!");
   }

   static void blowHorn() {
      System.out.println("Blowing horn!!!");
   }
}

interface FourWheeler {

   default void print() {
      System.out.println("I am a four wheeler!");
   }
}

class Car implements Vehicle, FourWheeler {

   public void print() {
      Vehicle.super.print();
      FourWheeler.super.print();
      Vehicle.blowHorn();
      System.out.println("I am a car!");
   }
}

public class Java8Tester {

   public static void main(String args[]) {
      Vehicle vehicle = new Car();
      vehicle.print();
   }
}
```

marker or tagged interface.java

```
import java.util.Scanner;

public class Student implements Cloneable {
   int age;
   String name;
   public Student (String name, int age){
      this.age = age;
      this.name = name;
   }
   public void display() {
      System.out.println("Name of the student is: "+name);
      System.out.println("Age of the student is: "+age);
   }
   public static void main (String args[]) throws CloneNotSupportedException {
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter your name: ");
      String name = sc.next();
      System.out.println("Enter your age: ");
      int age = sc.nextInt();
      Student obj = new Student(name, age);
      Student obj2 = (Student) obj.clone();
      obj2.display();
   }
}
```

Nestedinterface.java

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

nested interface which is declared within the class

```
class A{
  interface Message{
   void msg();
  }
}

class TestNestedInterface2 implements A.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  A.Message message=new TestNestedInterface2();//upcasting here
  message.msg();
 }
}
```

nested interface which is declared within the class

```
interface Library {
   void issueBook(Book b);
   void retrieveBook(Book b);
   public class Book {
      int bookId;
      String bookName;
      int issueDate;
      int returnDate;
   }
}
public class Sample implements Library {
   public void issueBook(Book b) {
      System.out.println("Book Issued");
   }
   public void retrieveBook(Book b) {
      System.out.println("Book Retrieved");
   }
   public static void main(String args[]) {
      Sample obj = new Sample();
      obj.issueBook(new Library.Book());
      obj.retrieveBook(new Library.Book());
   }
}
```

a class inside the interface?

```
interface Library {
   void issueBook(Book b);
   void retrieveBook(Book b);
   public class Book {
      int bookId;
      String bookName;
      int issueDate;
      int returnDate;
   }
}
public class Sample implements Library {
   public void issueBook(Book b) {
      System.out.println("Book Issued");
   }
   public void retrieveBook(Book b) {
      System.out.println("Book Retrieved");
   }
   public static void main(String args[]) {
      Sample obj = new Sample();
      obj.issueBook(new Library.Book());
      obj.retrieveBook(new Library.Book());
   }
}
```