OOPs concepts In Java

[March 21, 2017](http://bhanucorejava.blogspot.co.uk/2017/03/class-object-method.html)

**Class Object Method**

Class:- When we say class which means it represents generic term through which we can indicate the group of object.

In other word it is imaginary world or blueprint of objet, when we say human being, we won’t make out anything until we say some individual name.

When we say animal we won’t make out anything until we say some individual animal name. without saying animal name, we can just imagine groups of animal object.

e.g: Human being,Vehicle, Animal

Human being represents group of human objects (Ram, Shyam,Mohan)

Vehicle:

Vehicle represents groups of vehicle objects (Car, Bike, Truck)

Animal:-

Animal represents groups of animal objects (Cow, Bear, Ant)

Object:- Object Has properties and behaviors

For e.g Ram has properties (hight, weight,color) and Behaviors like (Walk, Talk, Eat, Sleep)

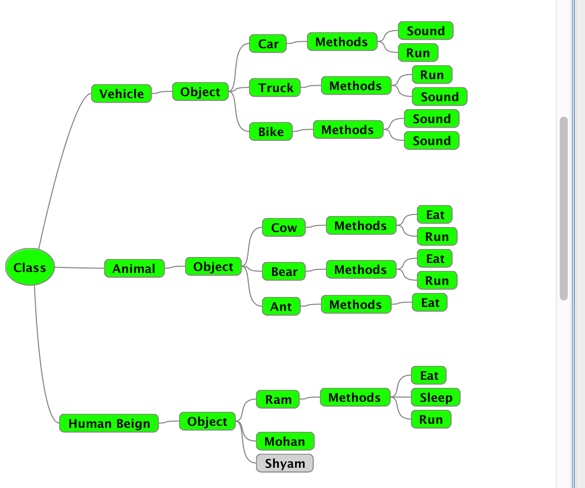
Cow Has Properties (color, weight, horn) and Behaviors like (Eat, Walk)

Car Has properties (Four wheel, Engine, Weight ) and Behaviors like (Run, Sound)

Method:- Behaviors or Action of object is Method

e.g Walk, Talk, Eat, Sleep,Run, Sound

Pictorial Representational

[](https://4.bp.blogspot.com/-i9aroc5vlT4/WNFDFX7H6oI/AAAAAAAAGuk/xThCwMMXn4occaQ-10xWyvGtX3FzJOxEwCPcB/s1600/TestClass.jpg)

**Structure Of Class**

Example of Class, Method and Object

package coreJavaLearning.basicOfJava;

public class Example1 {

int i;

int j;

public void test(int k){

System.out.println("I am method");

}

public static void main(String[] args) {

Example1 obj = new Example1();

obj.test(5);

}

}

**Point to remember**

* First Line of class is always package Name
* Public is access of Class
* int i, int j is class variables

**public void test(int k)**

* Method first String "Public" is access to method
* "int k" is local variable to method
* void is return type of method

**public static void main(String[] args)**

* Main method is always static method
* Return type of main method is always void
* Main method return type is always Array of String

**Static And Non Static Member Of Class**

Static Members of class are accessed by class Name, Since static members are class

Non Static members of class are accessed by object. Non Static members are object members

package coreJavaLearning.basicOfJava.staticAndNonStatic;

public class Example1 {

int i;

static int j;

public void test1() {

}

public void test2() {

}

public static void test3() {

}

public static void main(String[] args) {

Example1 obj = new Example1();

obj.test1();

obj.test2();

System.out.println(obj.i);

Example1.test3();

System.out.println(Example1.i)

Here we will get compile time error, Since we are trying to call non static variables through class name

System.out.println(Example1 .i);

We can't call non static method though class reference. if we try to do that we will get compile time error.     since non static members are object members

Example.test2();

}

}

In This Example we calling test1(), test2() through object reference

Example1 obj = new Example1();

obj.test1();

obj.test2();

test3() we are calling through Class reference

Example1.test3();

The above statement will be applicable for all non static and static members of class

package coreJavaLearning.basicOfJava.staticAndNonStatic;

public class Example3 {

int i;

static int j;

// We can't access non static members for static method

public static void test1(){

// Here we will get compile time error

int a = i;

}

// We can access non static and static members for non static method

public void test2(){

int a = i;

}

}

**Return type of Java**

The data type of the return value must match the method's declared return type We can't return an integer value from a method whose declaration type is void.

package coreJavaLearning.basicOfJava.returnType;

public class Example1 {

public void test1() {

}

public int test2() {

return 3;

}

public double test3() {

return 3.99;

}

public boolean test4() {

return true;

}

public char test5() {

return 'a';

}

public String test6() {

return "Test";

}

public Example1 test7() {

return new Example1();

}

public int[] test8() {

return new int[7];

}

In test1() method when we try to return integer data we will get compile time error. since method declaration type is void

In test2() method when we try to return String data we will get compile time error. since method declaration type is Integer

In test3() method when we try to return String data we will get compile time error. since method declaration type is double

In test4() method when we try to return void data we will get compile time error. since method declaration type is boolean

In test7() method we are returning object since method declaration is class type.

object declaration syntax

Example1 obj = new Example1(); that's why we are returning new Example1() for method test7()

test8() method we are returning array object since method declaration is array type.

object declaration syntax for array

int[] a = new int[7]; that's why we are returning new int[7] for method test8()

**Local And Global Variable**

**Local Variables:-**

* We write local variable within method , function and block.
* Local variables are local in nature, we can't access from outside method, function and block
* It is possible to have local variables with the same name in different functions.

**Global Variables:-**

* We write global variables outside method, function and block.
* We can't create duplicate global variable.
* We can access global variable by any method, function and block.

package coreJavaLearning.basicOfJava.LocalAndGlobalVariable;

public class Example1 {

int i;

int j;

char c;

public void test(int a, int b) {

}

public void test1() {

int a = 10;

int b = 20;

}

public static void main(String[] args) {

Example1 obj = new Example1();

System.out.println(obj.i);

/\*

Here when we can't access local variable through object reference

Since local variable can't be access outside method

(int i,int j,char c) are global variables. and we can access them from any method

Where as (int a, int b) are local variables. and we can't access from outside method

\*/

System.out.println(obj.a);

}

}

**Constructors in Java**

We Have two types of constructors

·      Default constructor

·      Parameterized constructors

A constructor with no argument is called as Default constructor

A constructor with argument is called as Parameterized constructors

**package** coreJavaLearning.basicOfJava.constructorInJava;

**public** **class** Example1 {

**int** i;

**int** j;

     //Default constructor

     Example1() {

     }

     //Parameterized constructors

     Example1(**int** i) {

     }

     //Parameterized constructors

     Example1(**int** p, **int** k) {

      i = p;

      j = k;

     }

     //Parameterized constructors

     Example1(**boolean** i) {

     }

**void** test() {

         System.***out***.println("the value of i is:-" + i + " value of j is :-" + j);

     }

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

         Example1 obj = **new** Example1();

         Example1 obj1 = **new** Example1(5);

         Example1 obj2 = **new** Example1(3, 4);

         Example1 obj3 = **new** Example1(**true**);

         obj2.test();

     }

}

**Points to Remember**

Constructor name should be same as class name.

Constructor will not return anything.

Constructor is used to initialized the global variables.

Constructor is used to supply different kind of data to object.

Java constructor is invoked at the time of object creation.  
We can’t keep return type for constructor.

**Q: Is it possible to Create object of default constructor when we have only parameterized constructor in class**

Answer: - NO (when we explicitly create parameterized constructors in class then java compiler will not keep by default constructor in class)

**Example**

package coreJavaLearning.basicOfJava.constructorInJava;

public class Example1 {

int i;

int j;

//Here we are initializing global variable through constructors

Example1(int p, int k) {

      i = p;

      j = k;

}

void test() {

System.out.println("the value of i is:-" + i + " value of j is :-" + j);

}

public static void main(String[] args) {

Example1 obj2 = new Example1(3, 4);

// When we call test method we will get the value of i is:-3 value of j is :-4

obj2.test();

}

}

**Access Modifier**

 We have four types of Access Modifiers

1.    Private

2.    Public

3.    Protected

4.    Default

Access Level of each type

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Type** | **Within Class** | **Outside Class** | **Within Same Package** | **Outside Package(Through Inheritance)** |
| Public | Yes | Yes | Yes | Yes |
| Private | yes | No | No | No |
| Protected | yes | Yes | Yes | Yes |
| Default | yes | Yes | Yes | No |

Example

package coreJavaLearning.basicOfJava.AccessType;

public class Example1 {

public int i;

private int j;

protected int k;

int d;

public void test1() {

}

private void test2() {

}

protected void test3() {

}

void test4() {

}

// Within same class we can access all types

public static void main(String[] args) {

Example1 obj = new Example1();

obj.test1();

obj.test2();

obj.test3();

obj.test4();

System.out.println(obj.i);

System.out.println(obj.j);

System.out.println(obj.k);

System.out.println(obj.d);

}

}

**Points to Remember**

1.    From Other Class in same package We can access (Public , Protected and Default)

2.    From Other Package we can access Public and Protected through Inheritance

package coreJavaLearning.basicOfJava.testAccessType;

import coreJavaLearning.basicOfJava.AccessType.Example1;

public class TestAcessType extends Example1{

            public static void main(String[] args) {

                        TestAcessType obj = new TestAcessType();

                        obj.test1();

                        obj.test3();

                        // Here We are not able to access test2() and test4()

            }

}

**This in Java**

**This in Java**

This is one of the java keywords and it helps in referring to the current object. When we use the dot(.) operator on this keyword, then we can access the member variables of the current object. this keyword can also be used to call one constructor from another constructors of the same class.

**Use of This Keyword**

•      It can be used to refer current class instance variable.

•      this() can be used to invoke current class constructor.

•      It can be used to invoke current class method (implicitly)

•      It can be passed as an argument in the method call.

•      It can be passed as argument in the constructor call.

•      It can also be used to return the current class instance.

**package** coreJavaLearning.basicOfJava.thisInJava;

**public** **class** Example1 {

**int** i;

**int** j;

            Example1() {

                        System.***out***.println("constructor call with default");

            }

            // Here This will call constructor with two arguments

            Example1(**int** i) {

**this**(7, 8);

                        System.***out***.println("constructor call with 1 parameters");

            }

            // Here This will call constructor with one argument

            Example1(**int** i, **int** j) {

**this**();

                        System.***out***.println("constructor call with 2 parameters");

            }

            // When we don't wright this here then global variables i and j  value would be zero

            // Since compiler will not understand which value should be initialized

**public** **void** test(**int** i, **int** j) {

**this**.i = i;

**this**.j = j;

            }

**public** **void** test1(**int** i, **int** j) {

**this**.i = i;

**this**.j = j;

            }

**public** **void** test(Example1 obj) {

                        System.***out***.println(obj);

            }

**public** **void** test3(**int** obj) {

                        test(**this**);

            }

**public** **void** dispaly() {

                        System.***out***.println("the value of i is:-" + i + " the value of j is:" + j);

            }

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

                        Example1 obj = **new** Example1(4);

                        obj.test(3, 4);

                        obj.dispaly();

                        obj.test3(5);

                        /\*

OutPut

constructor call with default

constructor call with 2 parameters

constructor call with 1 parameters

the value of i is:-3 the value of j is:4

coreJavaLearning.basicOfJava.thisInJava.Example1@7852e922

                         \*/

            }

}

**package** coreJavaLearning.basicOfJava.thisInJava;

**public** **class** Example2 {

**public** **void** test(Example2 obj) {

                        System.***out***.println(obj);

            }

            // Here I am using this as an argument to method call

            // Since test method required argument of class type

**public** **void** test3(**int** obj) {

                        test(**this**);

            }

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

            }

}

**package** coreJavaLearning.basicOfJava.thisInJava;

**public** **class** Example3 {

**int** i;

**int** j;

**public** **void** test1(){

                        System.***out***.println(**this**.i);

            }

            // Here I am using this for method call

            // Since This represents current class object

            // This is available only for non static members

**public** **void** test2(){

**this**.test1();

            }

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

            }

}

**Method Overloading In Java**

Method Overloading will allow us to create more than one methods with same name by changing the method arguments.

Method Overloading is called as compile time polymorphisms.

Arguments list can be different in following ways

1) Numbers of parameters to method

2) Data Type of parameters

3) Sequence Type of parameters

**1) Numbers of parameters to method example**

public class Example1 {

public void test1(int i) {

}

public void test1(int i, int j) {

}

public void test1(int i, int j, int k) {

}

}

**2) Data Type of parameters**

public class Example1 {

public void test1(int i, double d) {

}

public void test1(int i, int j) {

}

public void test1(int i, int j, boolean k) {

}

}

**3) Sequence parameters to method**

public class Example1 {

public void test1(int i, double d) {

}

public void test1(double i, int d) {

}

}

On Run time Method call happens based on the parameters supplied to method

public class Example1 {

public void test1(int i) {

System.out.println("Method with one argument");

}

public void test1(int i, int j) {

System.out.println("Method with two arguments");

}

public void test1(int i, int j, int k) {

System.out.println("Method with three arguments");

}

public static void main(String[] args) {

Example1 obj = new Example1();

obj.test1(4);

obj.test1(2, 3);

obj.test1(2, 3, 4);

}

}

**Output**

Method with one argument

Method with two arguments

Method with three arguments

Q: Is it possible to overload method just by changing return type.

Answer:-No, since compiler will not understand which method should be call on rum time

public class Example1 {

public void test1(int i) {

System.out.println("Method with one argument");

}

public int test1(int i) {

System.out.println("Method with one argument");

return 3;

}

}

The above example is not possible

**Method overriding In Java**

**Points to Note:-**

1.    Method Overriding is the the feature of java which allow us to create same method in parent and child class with same name and with same arguments.

2.    Method Overriding is the the ability of java which will make sure method call will happen from a class for which we have created the object. Not from referenced class.

3.    At compile time method call happens from reference class.

4.    At Run time method call happens from object class.

5.    Method Overriding is possible only by inheritance.

6.    Method Overriding we also call it as run time polymorphism.

7.    Method Overriding is the the feature of java which allow us to create same method in parent and child class with same name and with same arguments.

**For Point 1 and 2:**

public class ParentClass {

public void test() {

System.out.println("From ParentClass test()");

}

public int test1() {

System.out.println("From ParentClass test1()");

return 3;

}

}

public class ChildClass extends ParentClass {

public void test() {

System.out.println("From ChildClass test()");

}

public int test1() {

System.out.println("From ChildClass test1()");

return 3;

}

public static void main(String[] args) {

ParentClass obj = new ChildClass();

obj.test();

obj.test1();

}

}

**OutPut**

From ChildClass test()

From ChildClass test1()

**At compile time method call happens from reference class.**

public class ParentClass {

public void test() {

System.out.println("From ParentClass test()");

}

public int test1() {

System.out.println("From ParentClass test1()");

return 3;

}

public void test2() {

System.out.println("From ParentClass test()");

}

}

public class ChildClass extends ParentClass {

public void test() {

System.out.println("From ChildClass test()");

}

public int test1() {

System.out.println("From ChildClass test()");

return 3;

}

public static void main(String[] args) {

ParentClass obj = new ChildClass();

obj.test();

obj.test1();

obj.test2();

}

}

**Here test2() is getting called from reference class, though test2() is not available in object class**

**At Run time method call happens from object class. if that method is not present in child class then only call will happens from parent class.**

public class ParentClass {

public void test() {

System.out.println("From ParentClass test()");

}

public int test1() {

System.out.println("From ParentClass test1()");

return 3;

}

public void test2() {

System.out.println("From ParentClass test()");

}

}

public class ChildClass extends ParentClass {

public void test() {

System.out.println("From ChildClass test()");

}

public int test1() {

System.out.println("From ChildClass test()");

return 3;

}

public static void main(String[] args) {

ParentClass obj = new ChildClass();

obj.test();

obj.test1();

obj.test2();

}

}

**OutPut**

From ChildClass test()

From ChildClass test1()

From ParentClass test2()

**Inheritance in Java**

Points to Note:

* Through inheritance child class will acquire all non static members of class.
* We can't inherit private member of class.
* We can't inherit static members of class. since static members are class members.
* Final Members can not be inherit.

**Single Level Inheritance**

public class ParentClass {

public void test() {

System.out.println("From ParentClass test()");

}

public int test1() {

System.out.println("From ParentClass test1()");

return 3;

}

public void test2() {

System.out.println("From ParentClass test2()");

}

}

public class ChildClass extends ParentClass {

public static void main(String[] args) {

ParentClass obj = new ChildClass();

obj.test();

obj.test1();

obj.test2();

}

}

**Private Members can not be inherited, since we can not inherit test() method from child class**

public class ParentClass {

public void test() {

System.out.println("From ParentClass test()");

}

public int test1() {

System.out.println("From ParentClass test1()");

return 3;

}

public void test2() {

System.out.println("From ParentClass test2()");

}

private void test3() {

System.out.println("From ParentClass test2()");

}

}

public class ChildClass extends ParentClass {

public static void main(String[] args) {

ParentClass obj = new ChildClass();

obj.test();

obj.test1();

obj.test2();

}

}

**We can't inherit static members of class. since static members are class members.When we create test4() in child class , we will get compile time error.**

public class ParentClass {

public void test() {

System.out.println("From ParentClass test()");

}

public int test1() {

System.out.println("From ParentClass test1()");

return 3;

}

public void test2() {

System.out.println("From ParentClass test2()");

}

private void test3() {

System.out.println("From ParentClass test3()");

}

public static void test4() {

System.out.println("From ParentClass test4()");

}

}

public class ChildClass extends ParentClass {

public void test4(){

System.out.println("I am from ChildClass test4()");

}

public static void main(String[] args) {

ParentClass obj = new ChildClass();

obj.test();

obj.test1();

obj.test2();

}

}

**Final Members can not be inherit. when we try to do that we will get compile time error**

public class ParentClass {

public void test() {

System.out.println("From ParentClass test()");

}

final public int test1() {

System.out.println("From ParentClass test1()");

return 3;

}

public void test2() {

System.out.println("From ParentClass test2()");

}

private void test3() {

System.out.println("From ParentClass test3()");

}

}

public class ChildClass extends ParentClass {

public int test1() {

System.out.println("From ChildClass test1()");

return 3;

}

public static void main(String[] args) {

ParentClass obj = new ChildClass();

obj.test();

obj.test1();

obj.test2();

}

}

**Multilevel inheritance**

**package** constructorsInjava;

**public** **class** SuperParent {

**public** **void** test6(){

                        System.***out***.println("I am from SuperParent test6()");

            }

}

**import** constructorsInjava.SuperParent;

**public** **class** ParentClass **extends** SuperParent{

**public** **void** test() {

                        System.***out***.println("From ParentClass test()");

            }

**public** **int** test1() {

                        System.***out***.println("From ParentClass test1()");

**return** 3;

            }

**public** **void** test2() {

                        System.***out***.println("From ParentClass test2()");

            }

}

**public** **class** ChildClass **extends** ParentClass {

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

                        ParentClass obj = **new** ChildClass();

                        obj.test();

                        obj.test1();

                        obj.test2();

                        obj.test6();

            }

}

**Output:**

From ParentClass test()

From ParentClass test1()

From ParentClass test2()

I am from SuperParent test6()

**Interface in Java**

      Interface has only unimplemented methods.

      Interface members are by default (Public static final).

      We call interface as 100 % abstract class.

      Multiple inheritance is possible in case of interface.

      We cannot create object of interface, since all the members are unimpeded.

      We cannot create constructor of interface. Since object creation for interface is not possible.

      We cannot create object of class. Since members are unimplemented.

**Structure of interface**

**public** **interface** Example1 {

**public** **final** **static** **int** ***i***=90;

**int** ***j*** =80;

**public** **void** test1();

**public** **void** test2();

**public** **void** test3();

**public** **void** test4();

}

**Implementation of interface.**

When we implement interface we have to write implementation of all the methods in child class.

**public** **interface** Example1 {

**public** **final** **static** **int** ***i***=90;

**int** ***j*** =80;

**public** **void** test1();

**public** **void** test2();

**public** **void** test3();

**public** **void** test4();

}

**public** **class** TestInterface **implements** Example1{

            @Override

**public** **void** test1() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**public** **void** test2() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**public** **void** test3() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**public** **void** test4() {

                        // **TODO** Auto-generated method stub

            }

}

**Real Time Example of Interface**

**package** constructorsInjava;

**public** **interface** RateOfInterest {

**public** **void** rateofInterest();

}

**package** constructorsInjava;

**public** **class** SBI **implements** RateOfInterest{

            @Override

**public** **void** rateofInterest() {

      System.***out***.println("SBI gives 6 %");

            }

}

**package** constructorsInjava;

**public** **class** HDFC **implements** RateOfInterest{

            @Override

**public** **void** rateofInterest() {

                        System.***out***.println("HDFC gives 7 %");

            }

}

**package** constructorsInjava;

**public** **class** AXIS **implements** RateOfInterest{

            @Override

**public** **void** rateofInterest() {

                        System.***out***.println("AXIS gives 7 %");

            }

}

**package** constructorsInjava;

**public** **class** TestInterface {

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

                        SBI sbi = **new** SBI();

                        HDFC hdfc = **new** HDFC();

                        AXIS axis = **new** AXIS();

                        sbi.rateofInterest();

                        hdfc.rateofInterest();

                        axis.rateofInterest();

            }

}

**Out Put:**

SBI gives 6 %

HDFC gives 7 %

AXIS gives 7 %

**Multiple inheritance through interface example.**

In Interface we can inheritance multiple interface by writing interface as comma separated after**implements keyword.**

This is not possible in case of inheritance. We cannot write inheritance as comma separated after **extends keyword.**

**public** **interface** A {

**public** **void** test1();

}

**public** **interface** B {

**public** **void** test2();

}

**public** **interface** C {

**public** **void** test3();

}

**public** **class** TestMultipleInheritanceOfInterface **implements** A,B,C{

            @Override

**public** **void** test3() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**public** **void** test2() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**public** **void** test1() {

                        // **TODO** Auto-generated method stub

            }

}

**Abstract Class in Java**

      Abstract class will have implemented and unimpeded methods.

      We Cannot create of Abstract class.

      We Cannot write constructor in Abstract class.

      To make class Abstract class we need to have at least one method as Abstract method.

      We can create Reference of Abstract class and object of child class.

**Basic Structure**

**public** **abstract** **class** ExampleAbstarct1 {

**public** **abstract** **void** test1();

**public** **void** test2(){

                        System.***out***.println("I am implmented method test2() from abstract class");

            }

**public** **void** test3(){

                        System.***out***.println("I am implmented method test3() from abstract class");

            }

**public** **abstract** **void** test4();

**abstract** **void** test5();

}

**public** **class** TestAbstractClass **extends** ExampleAbstarct1{

            @Override

**public** **void** test1() {

                        System.***out***.println("I am from child class implementation test1() ");

            }

            @Override

**public** **void** test4() {

                        System.***out***.println("I am from child class implementation test4()");

            }

            @Override

**void** test5() {

                        System.***out***.println("I am from child class implementation ttest5() ");

            }

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

                        ExampleAbstarct1 obj = **new** TestAbstractClass();

                        obj.test1();

                        obj.test2();

                        obj.test3();

                        obj.test4();

                        obj.test5();

            }

}

**Out put**

I am from child class implementation test1()

I am implmented method test2() from abstract class

I am implmented method test3() from abstract class

I am from child class implementation test4()

I am from child class implementation ttest5()

**The key technical differences between an**[**abstract class**](http://us2.php.net/abstract)**and an**[**interface**](http://us2.php.net/interface)**are:**

       Methods and members of an abstract class can be defined with *any visibility*, whereas all methods of an interface must be defined as public (they are defined public by default).

       When inheriting an abstract class, a *concrete* child class *must define the abstract methods*, whereas an abstract class can extend another abstract class and abstract methods from the parent class don't have to be defined.

       Similarly, an interface extending another interface is *not responsible for implementing methods* from the parent interface. This is because interfaces cannot define any implementation.

       A child class can only *extend a single class* (abstract or concrete), whereas an interface can extend or a class can *implement multiple other interfaces*.

       A child class can define abstract methods with the *same or less restrictive visibility*, whereas a class implementing an interface must define the methods with the exact same visibility (public)

**Methods and members of an abstract class can be defined with *any visibility*, whereas all methods of an interface must be defined as public (they are defined public by default).**

**Here we are creating abstract class with all access type except private.**

**public** **abstract** **class** ExampleAbstarct1 {

**public** **abstract** **void** test1();

**protected** **abstract** **void** test4();

**abstract** **void** test5();

}

**When we extend by concrete class, by default all the access will not be public**

**public** **class** TestAbstractClass **extends** ExampleAbstarct1{

            @Override

**public** **void** test1() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**protected** **void** test4() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**void** test5() {

                        // **TODO** Auto-generated method stub

            }

}

**Here we are creating interface class with all access type except private.**

**public** **interface** Example1 {

**public** **void** test1();

**void** test2();

            // When we try to create method with protected access we will get compile time error.

            // Since by default access of methodsa are public

**protected** **void** test3();

}

**When we implement interface by default child class will get all method access as public. If we try to change access type, we will get compile time error.**

**public** **class** TestInterface **implements** Example1{

            @Override

**public** **void** test1() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**public** **void** test2() {

                        // **TODO** Auto-generated method stub

            }

}

**When inheriting an abstract class, a *concrete* child class *must define the abstract methods*, whereas an abstract class can extend another abstract class and abstract methods from the parent class don't have to be defined.**

When we extend one abstract class from other abstract class we don’t need to implement the abstract methods.

Same is not applicable when we extend from concrete class. Concrete class has to implement all unimplemented methods.

**public** **abstract** **class** AbstarctClass1 {

**public** **abstract** **void** test1();

**public** **abstract** **int** test2();

**public** **abstract** **void** test3();

}

**public** **abstract** **class** AbstarctClass2 **extends** AbstarctClass1{

}

**Concrete class has to implement all unimplemented methods. If we will not implement all unimplemented methods, then we will get compile time error.**

**public** **class** ConcreteClass **extends** AbstarctClass1{

            @Override

**public** **void** test1() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**public** **int** test2() {

                        // **TODO** Auto-generated method stub

**return** 0;

            }

            @Override

**public** **void** test3() {

                        // **TODO** Auto-generated method stub

            }

}

**When Interface extends other interface then method implementation by other interface is not required. Since interface cannot have implemented method**

**Point to remember:**

      Interface extends interface.

      Class implements interface

**public** **interface** Interface1 {

**public** **void** test1();

**public** **void** test2();

**public** **void** test3();

}

**public** **interface** Interface2 **extends** Interface1{

}

**// Here Class has to implement all the unimplemented methods of interface**

**public** **class** ConcreteClass **implements** Interface1{

            @Override

**public** **void** test1() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**public** **void** test2() {

                        // **TODO** Auto-generated method stub

            }

            @Override

**public** **void** test3() {

                        // **TODO** Auto-generated method stub

            }

}