**What is an Object?**



Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in *fields* (variables in some programming languages) and exposes its behavior through *methods* (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.

**What is a class?**

A classis simply a representation of a type of object. It is the blueprint/ plan/ template that describe the details of an object. A class is the blueprint from which the individual objects are created. Classis composed of three things: a name, attributes, and operations.

public class Student  
{  
}

Now we will add 2 variables to the class which are the student's name and student number.

public class Student  
{  
   String studentName;  
   int studentNumber;  
}

Next we must add methods for getting and setting these variables.

public class Student  
{  
   String studentName;  
   int studentNumber;  
   
   public void setStudentName(String s)  
   {  
      studentName = s;  
   }  
   
   public void setStudentNumber(int i)  
   {  
      studentNumber = i;  
   }  
   
   public String getStudentName()  
   {  
      return studentName;  
   }  
   
   public int getStudentNumber()  
   {  
       return studentNumber;  
   }  
}

## Instantiating an object

Now that we have finished writing the class we must create the main program that will instantiate the object. We will call the main program class TestClass. It should look just like how every other java program starts off.

public class TestClass  
{  
   public static void main(String[] args)  
   {  
   }  
}

Now we will instantiate the object. To do this we declare a reference to the object called stu and set it equal to a newly created object in memory.

public class TestClass  
{  
   public static void main(String[] args)  
   {  
      Student stu = new Student();  
   }  
}

## Using the object

Now we can call the methods from the stu object to set the values of its variables. You can't set the values of the variables in an object unless they are declared as public. This makes it safer to work with variables because they can't be changed by mistake by another object.

public class TestClass  
{  
   public static void main(String[] args)  
   {  
      Student stu = new Student();  
      stu.setStudentName("John Smith");  
      stu.setStudentNumber(12345);  
      System.out.println("Student Name: " + stu.getStudentName());  
      System.out.println("Student Number: " + stu.getStudentNumber());  
   }  
}

**Features of 100% OOP Languages:**

1. Encapsulation
2. Abstraction
3. Constructor
4. Inheritance
5. Polymorphism

**1)Encapsulation:**The wrapping up of data and functions into a single unit (called class) is known as encapsulation. Encapsulation is the process of binding together the methods and data variables as a single entity. This keeps both the data and functionality code safe from the outside world. It hides the data within the class and makes it available only through the methods.Java provides different accessibility scopes (public, protected, private, default) to hide the data from outside. **For example,**we can create a class **"Check"** which has a variable **"amount"** to store the current amount. Now to manipulate this variable we can create methods. For example to set the value of amount create **setAmount()** method and to get the value of amount create **getAmount()** method . Here is the code for "**Check"** class:

**class**Check{  
  private int amount=0;  
  **public int**getAmount(){  
   return amount;  
  }  
  **public void**setAmount(**int**amt){  
   amount=amt;  
  }   
}

**public class**Mainclass{  
  **public static void**main(String[] args){  
   int amt=0;  
   Check obj= **new**Check();  
   obj.setAmount(200);  
   amt=obj.getAmount();   
   System.out.println("Your current amount is :"+amt);  
   }  
}

Here the data variable "amount" and methods **setAmount()** and  **getAmount()** are enclosed together with in a single entity called the "Check" class. These two methods are used to manipulate  this variable i.e. set and get the current value of amount.

**Example:2**

class Encapsulation

{

private int height;

private int width;

private int length;

public void init()

{

height=100;

width=100;

length=100;

}

public void display()

{

System.out.println("height="+height);

System.out.println("lenghth="+length);

System.out.println("width="+width);

}

public static void main(String arr[])

{

Encapsulation encap=new Encapsulation()

encap.init();

encap.display();

}

}

**2)Abstraction:** Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The function that operate on these data are sometimes called methods or member functions.

**3)Constructor:** A constructor is a ‘special’ member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

The constructor functions have some special characteristics. These are:

1) They should be declared in the public section.

2) They are invoked automatically when the objects are created.

3) They do not have return types, not even void and therefore, and they cannot return values.

4) They cannot be inherited, though a derived class can call the base class constructor.

5) The name of the constructor must be same as the class name.

**Example:**

class Area{

Area(){

System.out.println(“No shape”);

}

Area(int a){

System.out.println(“The are of the Squre=”+(a\*a));

}

Area(double a){

System.out.println(“Circles area”Math.PI\*a\*a);

}

public static void main(String arr[])

{

Area a=new Area();

Area a1=new Area(4);

Area a2=new Area(4.0);

}

}

**NO DESTRUCTORS IN JAVA**

**GARBAGE COLLECTION:** In java there is a garbage collector, which is called automatically after the execution of the program and frees the memory. When no reference to the object exists that object is assumed to be no longer needed and the memory occupied by the object is released by the Garbage Collector.

**4)Inheritance:** Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order. As the name suggests, inheritance means to take something that is already made. It is one of the most important feature of Object Oriented Programming. It is the concept that is used for reusability purpose. Inheritance is the mechanism through which we can derived classes from other classes. The derived class is called as child class or the subclass or  we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class. To derive a class in java the keyword extends is used. To clearly understand the concept of inheritance you must go through the following example.

The concept of inheritance is used to make the things from general to more specific e.g. When we hear the word vehicle then we got an image in our mind that it moves from one place to another place it is used for traveling or carrying goods but the word vehicle does not specify whether it is two or three or four wheeler because it is a general word. But the word car makes a more specific image in mind than vehicle, that the car has four wheels . It concludes from the example that car is a specific word and vehicle is the general word. If we think technically  to this example then vehicle is the super class (or base class or parent class) and car is the subclass or child class because every car has the features of it's parent (in this case vehicle) class.

Example:

class a

{

public void display()

{

System.out.println("Hello");

}

}

public class testing2 extends a

{

public static void main(String args[])

{

testing2 t1 = new testing2();

t1.display();

}

}

**Types of Inheritance:**

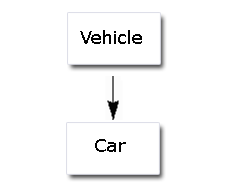
1)Single Inheritance

2)Multilevel Inheritance

3)Hierarchical Inheritance

4)Multiple Inheritace

**1)Single Inheritance:** When a  subclass is derived simply from it's parent class then this mechanism is known as simple inheritance. In case of simple inheritance there is only a sub class and it's parent class. It is also called single inheritance or one level inheritance.



**Syntax:**

class a

{

}

class b extends a

{

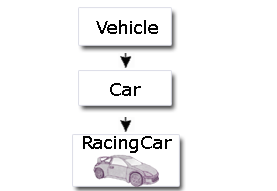
}

Example:

**class**A {  
  **int**x;  
  **int**y;  
  **void**get(**int**p, **int**q){  
  x=p; y=q;   }  
  **void**Show(){  
  System.out.println(x);

System.out.println(y);  
  }  
}  
  
**class**B **extends**A{  
  **public static void**main(String args[]){  
  B a = **new**B();  
  a.get(5,6);  
  a.Show();  
  }  
}

**2)Multilevel Inheritance:** It is the enhancement of the concept of inheritance. When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance. The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above ( parent ) class.  Multilevel inheritance can go up to any number of level.

**Syntax:**

Class a

{

}

Class b extends a

{

}

Class c extends b

{

}

Example:

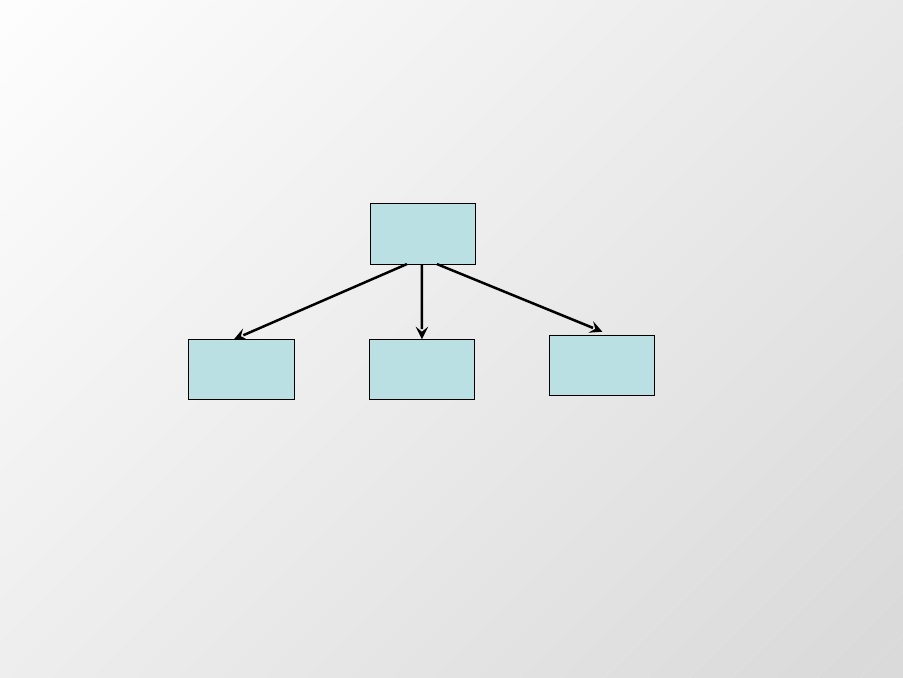
**class**A {  
  **int**x;  
  **int**y;  
  **void**get(**int**p, **int**q){  
  x=p; y=q;  
  }  
  **void**Show(){  
  System.out.println(x);

System.out.println(y);  
  }  
}  
**class**B **extends**A{  
  **void**Showb(){  
  System.out.println("B");  
  }  
}  
  
**class**C **extends**B{  
  **void**display(){  
  System.out.println("C");  
  }  
  **public static void**main(String args[]){  
  C a = **new**C();  
  a.get(5,6);  
  a.Show();

a.Showb();

a.display();  
  }  
}

**3)Hierarchical Inheritance:** In hierarchical type of inheritance, **one class is extended by many subclasses**. It is **one-to-many**relationship.

**Syntax:**

**Class a**

**{**

**}**

**Class b extends a**

**{**

**}**

**Class c extends a**

**{**

**}**

**Example-1:**

**class** one *//Super class*

{

int x=10,y=20;

**void** display()

{

System.out.println("This is the method in class one");

System.out.println("Value of X= "+x);

System.out.println("Value of Y= "+y);

}

}

**class** two **extends** one *//Sub class -1 of class one*

{

**void** add()

{

System.out.println("This is the method in class two");

System.out.println("X+Y= "+(x+y));

}

}

**class** three **extends** one*//Sub class-2 of class one*

{

**void** mul()

{

System.out.println("This is the method in class three");

System.out.println("X\*Y= "+(x\*y));

}

}

*/\* Main class \*/*

**class** Hier

{

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

{

two t1=**new** two(); *//Object of class two*

three t2=**new** three(); *//Object of class three*

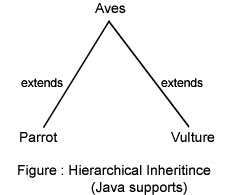
t1.display(); *//Calling method of class one using class two object*

t1.add(); *//Calling method of class two*

t2.mul(); *//Calling method of class three*

}

}

**Example-2:**

**class** Aves

{

**public** **void** fly()

{

System.out.println("Generally, aves fly");

}

}

**class** Parrot **extends** Aves

{

**public** **void** eat()

{

System.out.println("Parrot eats fruits and seeds");

}

}

**class** Vulture **extends** Aves

{

**public** **void** vision()

{

System.out.println("Vulture can see from high altitudes");

}

}

**public** **class** FlyingCreatures

{

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

{ *// all the following code is composition for FlyingCreatures*

Parrot p1 = **new** Parrot();

p1.eat(); *// calling its own member*

p1.fly();

*// calling super class member by inheritance*

Vulture v1 = **new** Vulture();

v1.vision(); *// calling its own member*

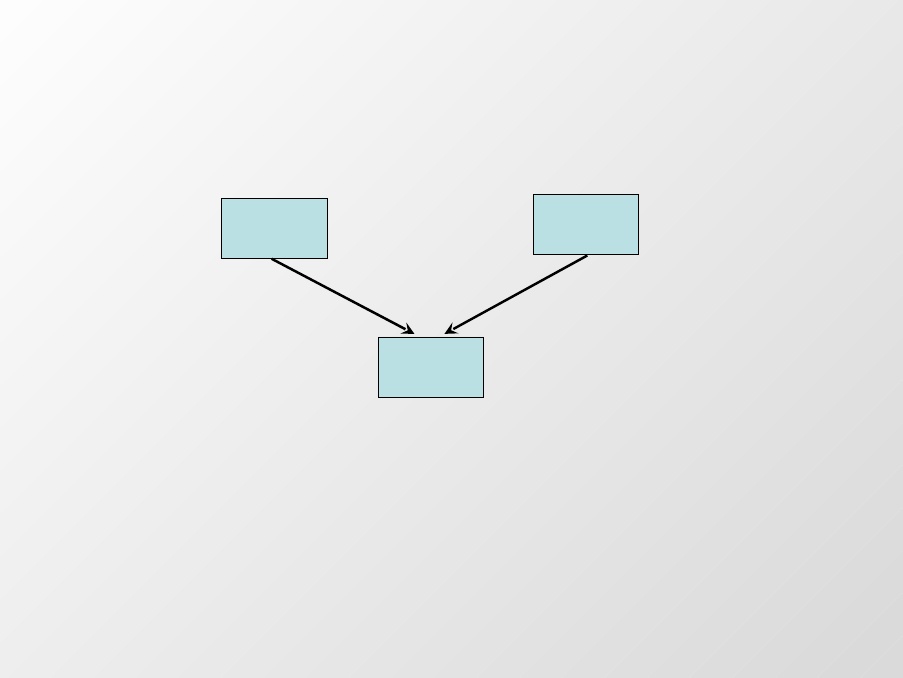
v1.fly(); *// calling super class member by inheritance*

}

}

In the above code, **Aves**class is extended by two classes – **Parrot**and **Vulture**. Both classes can make use of the methods of Aves. Even though the Parrot and Vulture are subclasses of the same class Aves, but still they cannot make use of each other members. Parrot and Vulture are known as "**siblings**". Siblings are disjoint and they cannot make use of other members as between them no inheritance is involved (like two sons of a father; one son's property cannot be shared by other but both can share the property of father). Following is the schematic representation of the classes involved.

**4)Multiple Inheritance:** The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritance. Java does not support multiple inheritance but the multiple inheritance can be achieved by using the interface.

**In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class.**

**This keyword:** The keyword **this**is useful when you need to refer to instance of the class from its method. The keyword helps us to avoid name conflicts. As we can see in the program that we have declare the name of instance variable and local variables same. Now to avoid the confliction between them we use **this** keyword. Here, this section provides you an example with the complete code of the program for the illustration of how to what is **this** keyword and how to use it.

In the example, **this.x** refers to the instance variable x while x refers to the local variable declared in Showb() method. We have made a program over **this**. After going through it you can better understand.

**Example:**

class A

{

int x=20;

}

public class testing2 extends A

{

public void Showb()

{

int x=30;

System.out.println(x);

System.out.println(this.x);

}

public static void main(String args[])

{

testing2 obj1 = new testing2();

obj1.Showb();

}

}

Output:

30

20

**Super Keyword:** The **super** is a keyword defined in the java programming language. Keywords are basically reserved words which have specific meaning relevant to a compiler in java programming language likewise the**super** keyword indicates the following :   
  
-- The super keyword in java programming language refers to the superclass of the class where the super keyword is currently being used.  
--  The super keyword as a standalone statement is used to call the constructor of the superclass in the base class.

**Example:**

class A

{

int x;

public A(int hardik)

{

x = hardik;

}

}

public class testing2 extends A

{

public testing2(int temp)

{

super(temp);

}

public void Showb()

{

int x=30;

System.out.println(x);

System.out.println(this.x);

}

public static void main(String args[])

{

testing2 obj1 = new testing2(1000);

obj1.Showb();

}

}

**Static keyword:** There will be times when you will want to define a class member that will be used independently of any object of that class. Normally a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.

**Methods declared as static have several restrictions :**

They can only call other **static** methods

They must only access **static**data.

They cannot use **this**or **super** in anyway (**Super** is a keyword used in Inheritance ).

**Example-1:**

public class testing2

{

public static void display()

{

System.out.println("This is static method");

}

public static void main(String args[])

{

display(); // Calling the method without declaring the object of the class

}

}

**Example-2:**

**class** ExampleStatic {

**static** **int** a = 5;

**static** **int** b;

**static** **void** setValMeth(**int** x) {

**System**.out.println("x = " + x);

**System**.out.println("a = " + a);

**System**.out.println("b = " + b);

}

**static** {

**System**.out.println("Static block initialized.");

b = a \* 5;

}

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

setValMeth(50);

}

}

In this example discussed all methods and variables are declared as **static**. So as an when you execute this code the 1st static declaration gets executed, then 2nd  and so on. As soon as the ExampleStatic class is loaded, all of the static statements are run. First, a is set to 5, then the static block executes (printing a message), and finally, b is initialized to a \* 5 or 25. Then main( ) is called, which calls meth( ), passing 50 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.   
**Note** It is illegal to refer to any instance variables inside of a static method.

**Output:**

Static block initialized.

x = 50

a = 5

b = 25

**What Is an Interface?**

The Tutorial wants to describe you a code that helps you in creating a Interface in java. Interface is defined as group of method, that implements a empty body. An example of Radio Tuner, when a listener switch on Radio Tuner, the tuner act  as interface between the electrical wing circuit inside the radio and you. Usually the Java do not support multiple inheritance, Interface in java is used for multiple inheritance.

To implement the interface the implements keyword is used.

**Example-1:**

interface area

{

public void display\_area();

}

interface perimeter

{

public void display\_perimeter();

}

public class rectangle implements area,perimeter

{

public void display\_area()

{

System.out.println("This is area interface");

}

public void display\_perimeter()

{

System.out.println("This is perimeter interface");

}

public static void main(String args[])

{

rectangle obj1 = new rectangle();

obj1.display\_area();

obj1.display\_perimeter();

}

}

**Example-2:**

interface A{

void display();

}

interface B extends A{

void show();

}

class C implements B{

void display(){

System.out.println(“interface A”);

}

void show(){

System.out.println(“Interface B”);

}

}

class D

{

public static void main(String arr[])

{

C c=new C();

c.display();

c.show();

}

}

**Few Facts about Interface:**

* An Interface is nothing but a prototype.
* An Interface can not have any non-abstract method.
* Interfaces are fully abstract.
* No Class can inherit an Interface, but can implement.
* In Java Multiple Inheritance can be implemented through Interfaces. That is one class implement more than one interfaces.
* One Interface can inherit another Interface.

**Abstract Class & Methods:**

The word abstract literally means : **INCOMPLETE.**

An ***abstract class*** is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An ***abstract method*** is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

* All the methods in an Interface are abstract.

abstract void display();

If a class includes abstract methods, the class itself *must* be declared abstract, as in:

public abstract class GraphicObject {

// declare fields

// declare non-abstract methods

abstract void draw();

}

**Example-1:**

abstract class A

{

abstract void callme();

void callmetoo()

{

System.out.println("This is a concrete method.");

}

}

class B extends A

{

void callme()

{

System.out.println("B's implementation of callme.");

}

}

public class testing2

{

public static void main(String args[])

{

B b = new B();

b.callme();

b.callmetoo();

}

}

**Output:**

B’s implementation of callme.

This is concrete method.

**Few Facts about Abstract class:**

* We can not declare abstract constructors
* We can not declare abstract static methods
* Any sub class of an abstract class must either implement all the abstract methods in the super class or be itself a abstract method
* We can not create any object of the abstract class.
* Inheritance of the abstract class is possible.

**Example-2:**

abstract class A

{

abstract void show();

void display()

{

System.out.println(“In class A”);

}

}

abstract class B extends A

{

abstract void show1();

void display1()

{

System.out.println(“In class B”);

}

}

class C extends B

{

void show()

{

System.out.println(“Definiton of show”);

}

void show1()

{

System.out.println(“Definition of show1”);

}

}

public class Ex

{

public static void main(String arr[])

{

C c=new C();

c.display();

c.display1();

c.show();

c.show1();

}

}

**Output:**

In class A

In class B

Definition of show

Definition of show1

**Example-3:**

abstract class Figure

{

double dim1;

double dim2;

Figure(double a, double b)

{

dim1 = a;

dim2 = b;

}

abstract double area();

}

class Rectangle extends Figure

{

Rectangle(double a, double b)

{

super(a, b);

}

double area()

{

System.out.println("Inside Area for Rectangle.");

return dim1 \* dim2;

}

}

class Triangle extends Figure

{

Triangle(double a, double b)

{

super(a, b);

}

double area()

{

System.out.println("Inside Area for Triangle.");

return dim1 \* dim2 / 2;

}

}

public class testing2

{

public static void main(String args[])

{

Rectangle r = new Rectangle(9, 5);

Triangle t = new Triangle(10, 8);

System.out.println("Area is " + r.area());

System.out.println("Area is " + t.area());

}

}

**Polymorphism:**Polymorphism allows one interface to be used for a set of actions i.e. one name may refer to different functionality. Polymorphism allows a object to accept different requests of a client (it then properly interprets the request like choosing appropriate method) and responds  according to the current state of the runtime system, all without bothering the user.

There are two types of polymorphism :

1. **Compile-time polymorphism**
2. **Runtime Polymorphism**

**1.Compile-time polymorphism:**

In **compiletime** Polymorphism, method to be invoked is determined at the compile time. Compile time polymorphism is supported through the **method overloading**concept in java.

Method overloading means having multiple methods with same name but with different signature (number, type and order of parameters).

**Example:**

**class**A{  
  **public void**fun1(**int**x){  
  System.out.println("The value of class A is : " + x);  
  }  
  **public void**fun1(**int**x,int y){  
  System.out.println("The value of class B is : " + x + " and " + y);  
  }  
}  
**public class**polyone{  
  **public static void**main(String[] args){  
  A obj=new A();

  obj.fun1(2);

   obj.fun1(2,3);   
  }  
}

**Output:**

**The value of class A is : 2  
The value of class B is : 2 and 3**

**2.Run Time Polymorphism:** In **rumtime** polymorphism, the method to be invoked is determined at the run time. The example of run time polymorphism is **method overriding**. When a subclass contains a method with the same name and signature as in the super class then it is called as method overriding.

**Example-1:**

**class**A{  
  **public void**fun1(**int**x){  
   System.out.println("int in Class A is : "+ x);  
  }  
}  
  
**class**B **extends**A{  
  **public void**fun1(**int**x){  
   System.out.println("int in Class B is : "+ x);  
  }  
}  
  
**public class**polytwo{  
  **public static void**main(String[] args){  
   A obj;  
   
   obj= **new**A(); // line 1  
   obj.fun1(2);  // line 2 (prints "int in Class A is : 2")  
   
   obj=**new**B();  // line 3  
   obj.fun1(5);  // line 4 (prints ""int in Class B is : 5")  
  }  
}

**Output:**

**int in Class A is : 2  
int in Class B is : 5**

In the above program, **obj** has been declared as A type. In line 1, object of class A is assigned. Now in the next line, fun1(int) of class A will be called. In line 3, obj has been assigned the object of class C so fun1(int) of class C will be invoked in line 4. Now we can understand that same name of the method invokes different functions, defined in different classes, according to the current type of variable **"obj"**. This binding of  method code to the method call is decided at run time.

**Example-2:**

class Animal {

  void whoAmI() {

    System.out.println("I am a generic Animal.");  
  }  
}  
class Dog extends Animal {  
 void whoAmI() {  
    System.out.println("I am a Dog.");  
  }  
}  
class Cow extends Animal {  
  void whoAmI() {  
    System.out.println("I am a Cow.");  
  }  
}

class Snake extends Animal {  
  void whoAmI() {  
    System.out.println("I am a Snake.");  
  }  
}  
class RuntimePolymorphismDemo {  
  public static void main(String[] args) {  
    Animal ref1 = new Animal();

ref1.whoAmI();  
ref1 = new Dog();

ref1.whoAmI();

ref1=new cow();

ref1.whoAmI();

ref1=new snake();

ref1.whoAmI();

  }

}

**Output:**

I am a generic Animal.  
I am a Dog.  
I am a Cow.  
I am a Snake.

In the example, there are four variables of type Animal (e.g., *ref1*, *ref2*, *ref3*, and *ref4*). Only*ref1* refers to an instance of *Animal* class, all others refer to an instance of the subclasses of*Animal*. From the output results, you can confirm that version of a method is invoked based on the actually object's type.