Chapter - 3

Classes and Objects

Class 50
Access Specifiers 52
Access Specifier: default 52
Access Specifier: public 52
Access Specifier: protected 53
Access Specifier: private 53
Objects 53
Accessing Class members 54
Accessing private instance variables of class 56
A program on student class 57
Array of Objects 60
The this keyword 61
Passing Objects as Arguments to Methods 64
Returning Objects 67
Class Loading 70
Static Members 72
Static variables 72
Public static variables 77
Differences between static and non-static variables .. 78
Static Methods 79
Public static methods 81
Static block 82
Nested and Inner classes 85
Non-static nested classes or Inner classes 85
Static nested classes 86

Classes and Objects

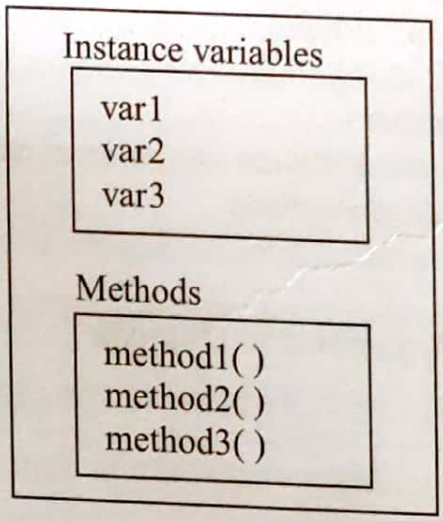
All the features of **OOP**(**O**bject **O**riented **P**rogramming) can be implemented using classes and objects. Classes are the basic building of **OOP**.

We understand classes and objects with real-time example. Let say, we want to construct a building of 3 floors. The first thing we do is, we approach an engineer, who gives the blue print (structure or design) of building. The blue print is known as **class**. After the blue print is prepared, which is then given to the constructor who actually construct the building on the land according to the blue print given by the engineer. This real building constructed is know as **object**. Therefore we can say an object is an implementation of class.

What is a class and object?

Class : A **class** is a collection of **instance variables** and **methods**.

Object : An object is an instantiation of class.



Class grouping instance variables and Methods

Class

A class is a combination of **instance variables** and **methods**. Instance variables are the varibles defined in the class and methods are the functions defined in the class.

Syntax:

```
class   classname
{
        access_specifier  type  instance_variable1[=value], . . . . . ;
        access_specifier  type  instance_variable2[=value], . . . . . ;
        :
        :
        access_specifier returntype  methodname1([type para1, . .])
        {
                // body of  method
        }
        access_specifier returntype  methodname2([type para1, . .])
        {
                // body of  method
        }
        :
        :
}
```

The java class have two types of members i.e. **Instance Varaibles** and **Methods**.

Instance Variables are the variables created inside the class.

Methods are the functions that are defined inside the class.

Note :

In CPP functions can be written inside or outside the class therefore function of class are called as member functions.

In Java functions should be defined only inside the class that is why in java functions of class are called as methods.

Access Specifiers

Access specifer also called as access modifier that explains to the **JVM**(**J**ava **V**irtural **M**achine) about the accessing permission of members of a class. The access specifier of a member (instance variable or method) specifies the scope of the member where it can be accessed. The access specifers are four. They are **default, public, protected** and **private.**

Access Specifier : default

If no access specifier is given for the member then java takes **default** as default access specifier for that member. The default members can be accessed inside and outside the class but in the same package. Outside the package the default members can not be accessed.

Example:

```
class A
{
        int x;          //  access specifier is default
        void sum( )  //  access specifier is default
        {
                :
        }
}
```

Access Specifier : public

The members declared with the access specifier public are said to be public members and such members can be accessed in the same and other classes regardless of their package as well as all its subclasses. Public members can be defined using **public** keyword.

Example:

```
class A
{
        public int x;
        public void sum( )
        {
                :
        }
}
```

Access Specifier : protected

The members declared with the access specifier protected are said to be protected members and such members can be accessed in the same class and in any other classes of same package and its subclasses of other packages. Protected members can be defined with the keyword **protected**.

Example:

```
class A
{
        protected int x;
        protected void sum( )
        {
                :
        }
}
```

Access Specifier : private

The members declared with the access specifer private are said to be private members and such private members should be accessed only with in the same class in which they are created and they cannot be accessed outside the class. Private members can be declared with the keyword **private**.

Example:

```
class A
{
        private int x;
        private void sum( )
        {
                :
        }
}
```

Objects

A class specification only declares the structure of objects and it must be instantiated in order to make use of the services provided by it. This process of creation of objects (variables) of the class is called **class instantiation**. It is the definition of an object that actually creates objects in the program by setting aside memory space for its storage. Hence, *a*

class is like a blueprint of an object and it indicates how the instance variables and methods are used when the class is instantiated. The necessary resources are allocated only when a class is instantiated.

Creation of classes is nothing but creation of userdefined datatype. We can use this new datatype (class) to create variables of the class type. However, creating objects of a class is a two-step process.

First, we must declare a reference variable of the class. This reference variable does not define an object, instead it creates a variable that stores reference(address) of an object.

Syntax:

```
classname obj;
```

Here **obj** is a reference variable of classname that can store reference(address) of object of classname.

Second, we must create an actual physical copy of the object and assign it to the reference variable. This can be done using the **new** operator.

Syntax:

```
obj=new classname();
```

The **new** operator allocates memory of **classname()** dynamically (at run-time) which is said to be an object and whose address is assigned to reference variable **obj**.

We can also create objects in single statement using the following syntax

```
classname obj=new classname();
```

In java objects are instantiated at runtime using new operator.

Accessing Class members

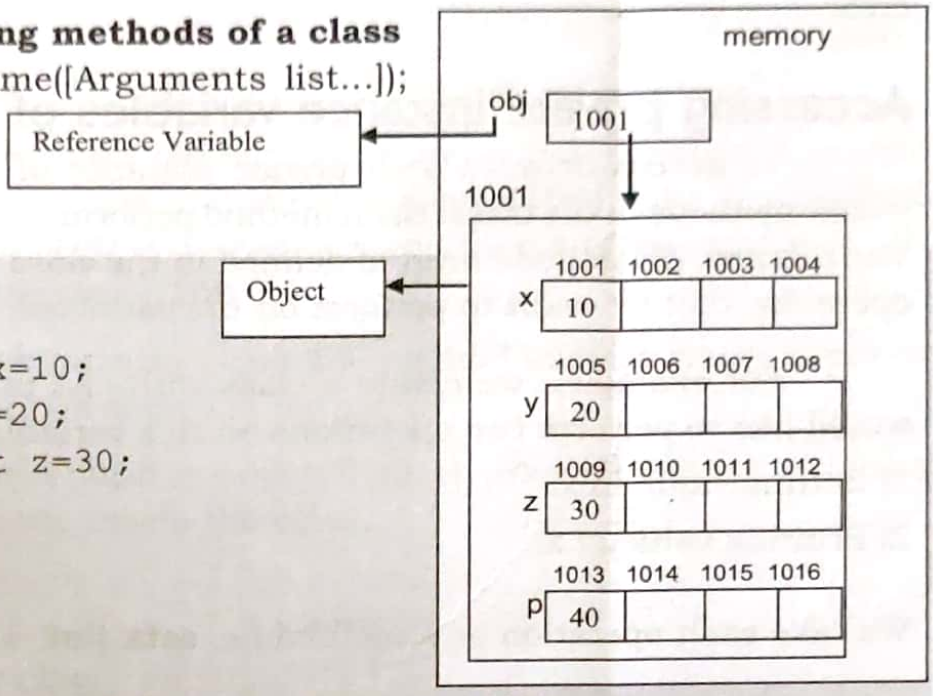
Once an object of a class has been created, there must be a provision to access its members. This can be achieved by using the member access operator . (dot).

1) **Syntax for accessing instance variable of a class**

Objectname.instancevariable;

1) **Syntax for accessing methods of a class**

Objectname.methodname([Arguments list...]);



Example1:

Bin>edit ex1.java

```
class sample
{
    private int x=10;
    public int y=20;
    protected int z=30;
    int p=40;
}
class ex1
{
    public static void main(String argv[])
    {
          sample obj=new sample();
          // System.out.print("\n obj.x=" +obj.x); //Error
          System.out.print("\n obj.y=" +obj.y);
          System.out.print("\n obj.z=" +obj.z);
          System.out.print("\n obj.p=" +obj.p);
    }
}
```

Bin>javac ex1.java

Bin>java ex1

obj.y=20

obj.z=30

obj.p=40

Explanation:

In the main() the statement **sample obj=new sample();** the **new** allocates memory of **sample()** which is the actual object and whose address 1001 is assigned to **obj** as show in the above memory. The private variable **x** is not accessed outside the class therefore it is commented, otherwise it is an error. Where as public, protected and default variables **y**, **z** and **p** can be accessed outside the class using objectname as a result we got the output as shown in above output.

Point to remember

1) A class with only private variables is no use and the compiler generates error.

Accessing private instance variables of class

To access or work with private instance of class we have to define public methods in the class. Each method performs operation on the variables. The number of methods need to defined in the class based on the number of operation that we want to perform on the variables.

For example, we create a class with one private variable **x** and we would like to perform two operations on this variable.

1) Setting value to x.

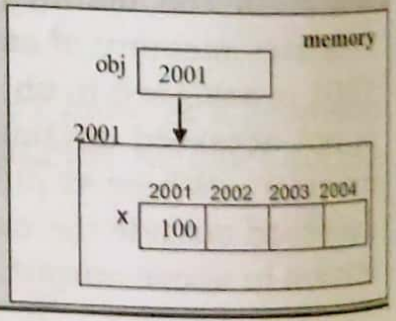
2) Printing value of x.

We take each operation as a method i.e. **setx (int a)** and **printx( )**.

Program

Bin> edit ex2.java

```
class sample
{
      private int x;
      public void setx(int a)   //sets value to x
      {
          x=a;
      }
      public void printx()      // prints value of x
      {
          System.out.print("\n x=" + x);
      }
}
class ex2
{
    public static void main(String argv[])
    {
          sample obj=new sample();
          obj.setx(100);
          obj.printx();
    }
}
```



Bin> java exp2.java
Bin>java exp2
x=100

Explanation:

In the main, the object **obj** is created and which contains **x** private variable of sample class. The statement **obj.setx(100)** calls the method present in the class and **100** is passed to **a** and then **a** is assigned to **x**. In this way **x** is set with the value **100**

The statement **obj.printx( )** calls the method in class where it prints the value of **x** on monitor as shown in output.

In this way if we want to perform any operations on private instance variables we have to define methods inside the class.

Example:

A program on student class.

Analysis

Before we write program on classes we have to decide classname, instance variables and methods of class.

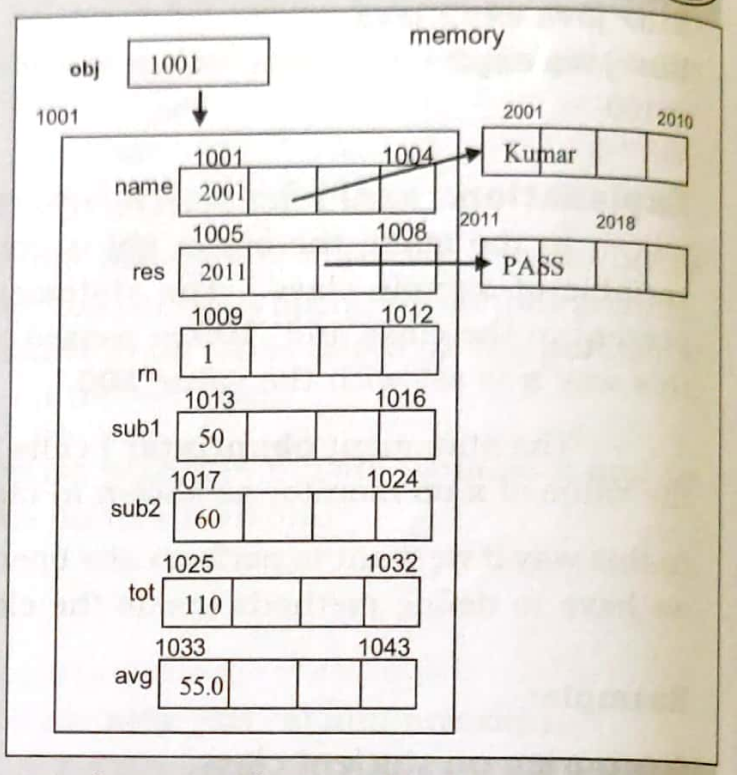
ClassName : Student

1. **Instance Variables :** name, rn, sub1, sub2, tot ,avg, res.
2. **Methods :**
   setstudent (String n,int r, int s1,int s2)
   sets values for name, rn, sub1, sub2.
   calculate () ⟶ calculates the tot, avg, res.
   printstudent() ⟶ prints the all variables.

Program
Bin>edit stud.java

```
class student
{
    private string name,res;
    private int rn,sub1,sub2,tot;
    private double avg;
    public void setstudent(String n,int r,int s1, int s2)
    {
```

```
            name=n;
            rn=r;
            sub1=s1;
            sub2=s2;
        }
    public void calculate()
    {
        tot=sub1+sub2;
        avg=tot/2.0;
    if(sub1>=35 && sub2>=35)
      res="PASS";
    else
      res="FAIL";
    }
public void printstudent()
{
    System.out.print("\nname="+name+"\n rn=" +rn +
        "\nsub1="+sub1+"\n sub2="+sub2+"\n tot="+
        tot+"\n avg="+avg+"\t res="+res);
}
}
class stud
{
    public static void main(String argv[])
    {
        student obj=new student();
        obj.setstudent("Kumar",1,50,60);
        obj.calculate();
        obj.printstudent();
    }
}
```



Bin> javac stud.java
Bin> java stud

name = Kumar
rn=1
sub1=50
suh2=60
tot=110
avg=55.0
res=PASS

Explanation:

In the main, **obj** is the object created to student class as shown in the memory which contains all instance variables of student class. The statement **obj.setstudent("Kumar",1,50,60)** calls the **setstudent()** method present in the class and which sets the values for name, rn, sub1, sub2 with "kumar", 1 , 50 and 60.

The statement **obj.calculate()** calls the calculate method in student class where it calculates tot, avg and res.

The statement **obj.printstudent()** prints the values of all instance variables of **obj** as shown in output.

Assignment

1. Write a program on employee class.

   Instance variables : name, empno, basic, hra, da, pf, it and netsal.

   Methods :

   reademployee() //read values for name, empno,basic.
   calculate( ) //calculate hra,da,pf,it,netsal.
   showemployee() //display values of all variables

   Conditions:

If basic>=5000 then	If basic<5000
hra=40% on basic.	hra=25% on basic.
da=30% on basic.	da=15% on basic.
pf=20% on basic.	pf=8% on basic.
it=10% on basic.	it=5% on basic.

2. Write a program to calculate area of a triangle using class.

   Instance variables : area, base, height

   Methods :

   readvalues( ) //should read base and height
   cal_area( ) //calculate area
   printvalues( ) //print all datamembers

3. Write a program for manipulating coordinates in Rectangle coordinate system. Represent points as objects. The class point must include members such as x and y (as instance variables), and add ( ), sub ( ), angle ( ), etc. (as methods).

Array of objects

In java, objects can not be defined as an array directly. But array of reference elements can be created. For each reference element and object memory can be allocated.

Syntax:

```
classname objectname[ ] = new classname[n];
                    or
classname objectname[ ];
objectname = new classname[n]
```

A program on array of objects

Bin>edit arrayobj.java

```
class sample
{
      int x;
}
class arrayobj
{
      public static void main(String argv[ ])
      {
            sample obj[ ];
            obj=new sample[2];
            obj[0]=new sample();
            obj[1]=new sample();
            obj[0].x=10;
            obj[1].x=20;
            System.out.print("\n obj[0].x="+ obj[0].x);
            System.out.print("\n obj[1].x="+ obj[1].x);
      }
}
```

Bin>javac arrayobj.java
Bin>java arrayobj

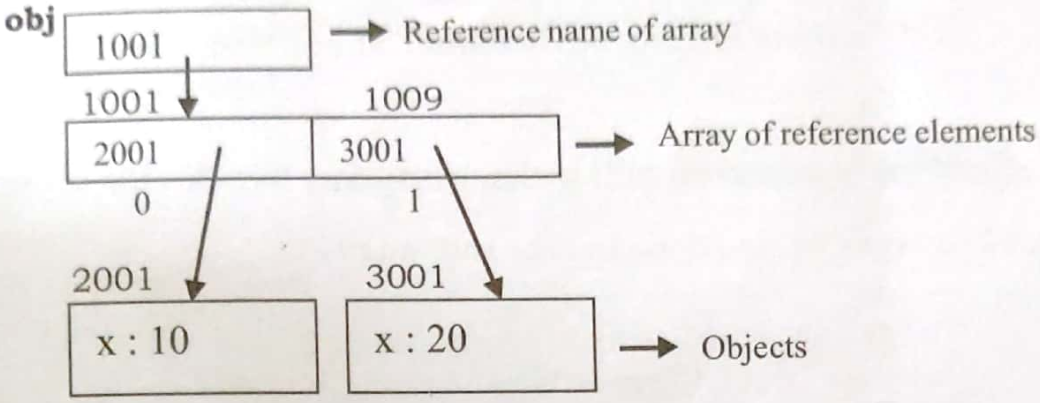
```
obj[0].x=10;
obj[1].x=20
```

Explanation:

In the above program, the statement **sample obj[ ];** creates a reference name of array of sample type. In the next statement **obj=new sample[2];** allocates two reference elements of sample type to **obj**. The statements

```
obj[0]=new sample();
obj[1]=new sample();
```

where **obj[0]** and **obj[1]** reference elements are allocated with an object of sample class memory as shown below:



The statements

```
obj[0].x=10;  //stores 10 into x of obj[0] object
obj[1].x=20;  //stores 20 into x of obj[1] object
```

In the print() of program, the statements obj[0].x gives 10 and obj[1].x gives 20.

The this keyword

Local variables of methods can have the same name as that of instance variable of class. This is possible because local variable is local to the function in which they are created where as instance variables are global to the same class i.e they can be accessed by any method of same class

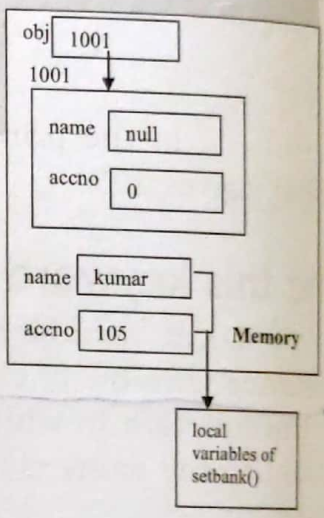
When local variables (or) parameters of methods is having the same name as that of instance variables of class then the java gives first preference to the local variables of same method as a result only local variables are accessed in the method and not the instance variables (global variables).

Example:

Bin>edit accounts.java

```
class bank
{
      private String name;
      private int accno;
      public void setbank(String name, int accno)
      {
            name = name;
            accno=accno;
```

```
    }
    public void show( )
    {
       System.out.print("\nname="+name+"\naccno="+accno);
    }
}
class accounts
{
    public static void main(String agrv[])
    {
          bank obj=new bank( );
          obj.setbank("kumar", 105);
          System.out.print("\nBank Account...");
          obj.show( );
    }
}
```



Bin>javac accounts.java
Bin>java accounts
Bank Account...
name=null
accno=0

Explanation:

In the main(), **obj** is the object instantiated to bank class as shown in memory. The statement **obj.setbank("kumar",105)** calls the **setbank(String name, int accno)** method inside the class. The values **"kumar"** , **105** passes to **name** and **accno** of **setbank ()**. In this case the local variables **name**, **accno** and instance variables of class are having same names. Therefore in the method only local variables are accessed but not the instance variables of class or object. As a result, the local variable values are again assigned to same local variables but not to the instance variables of object. The statement **obj.showbank()** displays name=null and accno=0 of **obj**.

To overcome the above problem or drawback we can use ***this*** keyword.

The **this** keyword is a predefined reference variable in java which can store address of any object. When a method is called by an object, the address of object is assigned to ***this*** reference variable, and ***this*** reference variable can be used in the method to know the address of object or to access

the members of object or class explicitly. When instance variables of class and local variables of methods have same name then instance variables can be accessed using ***this*** reference variable explicitly.

When the method is terminated the ***this*** reference variable created is automatically deleted.

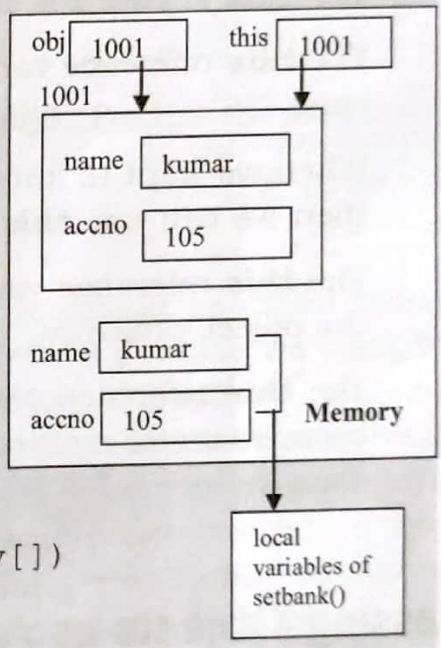
Solution for the above program using this reference variable.

Example:

Bin>edit accounts1.java

```
class bank
{
    private String name;
    private int accno;
    public void setbank( String name, int accno)
    {
          this.name = name;
          this.accno=accno;
    }

    public void showbank( )
    {
          System.out.print("\n name ="
          +name+"\n accno ="+accno);
    }
}
class accounts
{
    public static void main(String agrv[])
    {
          bank obj=new bank( );
          obj.setbank("kumar", 105);
          System.out.print("\n Bank Account...");
          obj.showbank( );
    }
}
```



Bin>javac accounts.java

Bin>java accounts

Bank Account...

name=kumar

accno=105

Explanation:

In the main(), **obj** is the object instantiated to bank class as shown in memory. The statement **obj.setbank("kumar",105)** calls the **setbank(String name, int accno)** method inside the class. The values **"kumar"**, **105** passes to **name** and **accno** of **setbank ()**. When the **setbank()** is called by **obj** whose address(**101**) is stored into ***this*** reference variable as shown in above memory. Now ***this*** reference variable is pointing to **obj**. Inside the **setbank()** the **name** and **accno** of **obj** is accessed using ***this*** reference variable. Therefore the local variables **name**(kumar) and **accno**(105) values are assigned to instance variables of object as shown in memory. When the **setbank()** is terminated the ***this*** reference variable is automatically deleted from memory. The statement **obj.showbank()** displays **name** and **accno** of **obj** as shown in output.

Points to Remember

1) The **this** keyword is predefined reference variable in Java.
2) The **this** reference variable should be used only inside the methods of class.
3) When we want to know the address of called object inside the method then we can use **this** reference variable.
4) The **this** reference variable should be used to access the members of the object explicitly.
5) The **this** reference variable should not be used in the static methods because static methods can be called without using object but using class name.

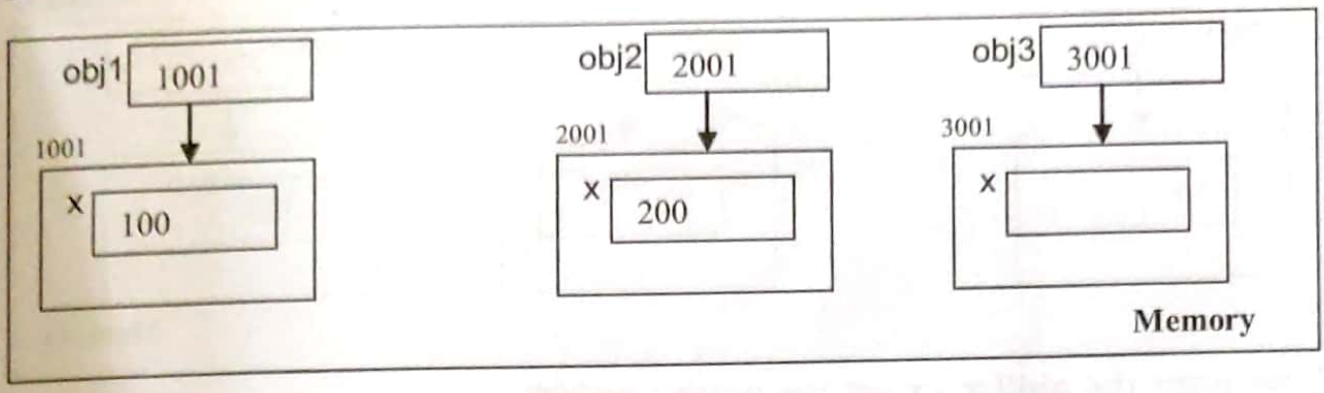
Passing Objects as Arguments to methods

To the methods we can pass arguments of any type such as int, char, float, double and so on. Even objects of classes can also be passed to methods similar to other data types.

Example: A program to print the addition of two objects contents.

Logic:

Let say there are three objects **obj1**, **obj2** and **obj3** where obj1.x=100, obj2.x=200 and obj3.x is nothing as shown memory below.



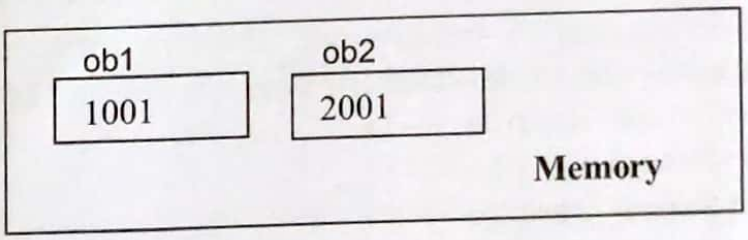
The class for the objects is:

```
class sample
{
        private int x;
        public void sum(sample ob1, sample ob2)
        {
                x=ob1.x+ob2.x;
        }
}
```

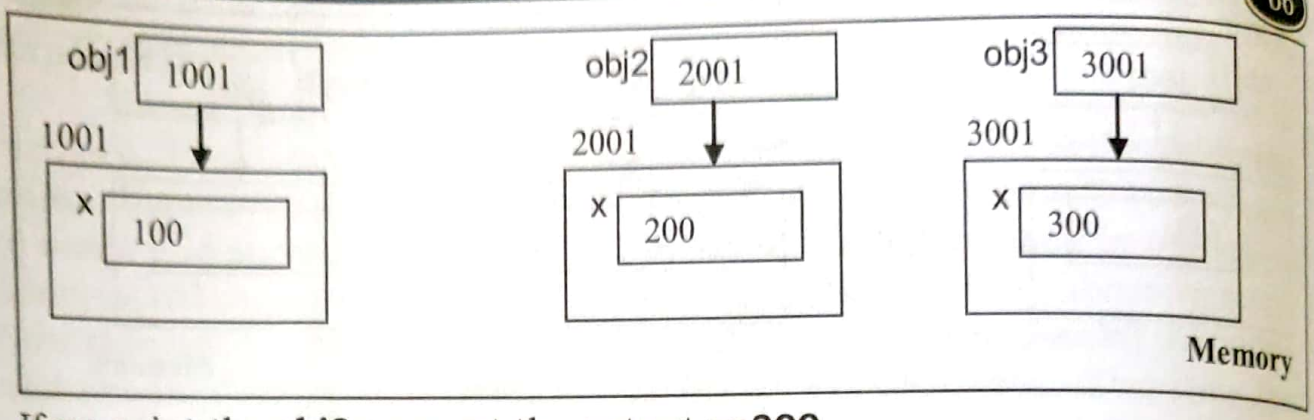
Now we want to add **obj1**, **obj2** values and store in **obj3**. For this the statement we write..

```
obj3.sum(obj1,obj2);
```

sum( ) is the method of sample class, therefore it should be called using object name and we are using **obj3** to call the method because we want to store the results in **obj3**. The calling object **obj3** is passed **implicitly**(internally) to the **sum()** and whose members(instance variables) can be accessed directly without using object name. Where as other objects **obj1**(1001) and **obj2**(2001) are passed **explicitly**(externally) to the parameters(**ob1**,**ob2**) and using these explicit objects(**ob1**,**ob2**) members should be accessed using object name. The memory of **ob1** and **ob2** is..



Now in the **sum( )** method, the statement **x=ob1.x + ob2.x;** adds **x** value of **ob1** (100) and **x** value of **ob2** (200) and result stores in **x** of implicit object **obj3**. Therefore result in memory is..



If we print the **obj3.x** we get the output **x=300**.

Program

Bin>edit sum1.java

```
class sample
{
    private int x;
    public void setx(int a)
    {
          x=a;
    }
    public void showx( )
    {
          System.out.print(" x = " + x );
    }
    public void sum(sample ob1, sample ob2)
    {
          x=ob1.x+ob2.x;
    }
}
class sum1
{
    public static void main(String  argv[])
    {
          sample obj1,obj2,obj3;
          obj1=new sample ( );
          obj2=new sample ( );
          obj3=new sample ( );
          obj1.setx(100);
          obj2.setx(200);
          obj3.sum(obj1,obj2);
          System.out.print("\n obj1.....");
```

```
            obj1.showx ( );
            System.out.print("\n obj2.....");
            obj2.showx ( );
            System.out.print("\n obj3.....");
            obj3.showx ( );
    }

}
```

Bin> javac sum1.java

Bin> java sum1

obj1.....x =100

obj2.....x =200

obj3.....x =300

Points to remember:

- ➔ One object should be selected to call the method and remaining objects should be passed explicitly.
- ➔ The calling object is passed implicitly.
- ➔ Implicit object members can be accessed in the method directly without using object name.
- ➔ Explicit objects members should be accessed in the method using object name.
- ➔ Any changes made to the implicit object in the method are permanent.
- ➔ Any changes made to the explicit objects are also permanent because passing objects is pass by reference.

Returning objects

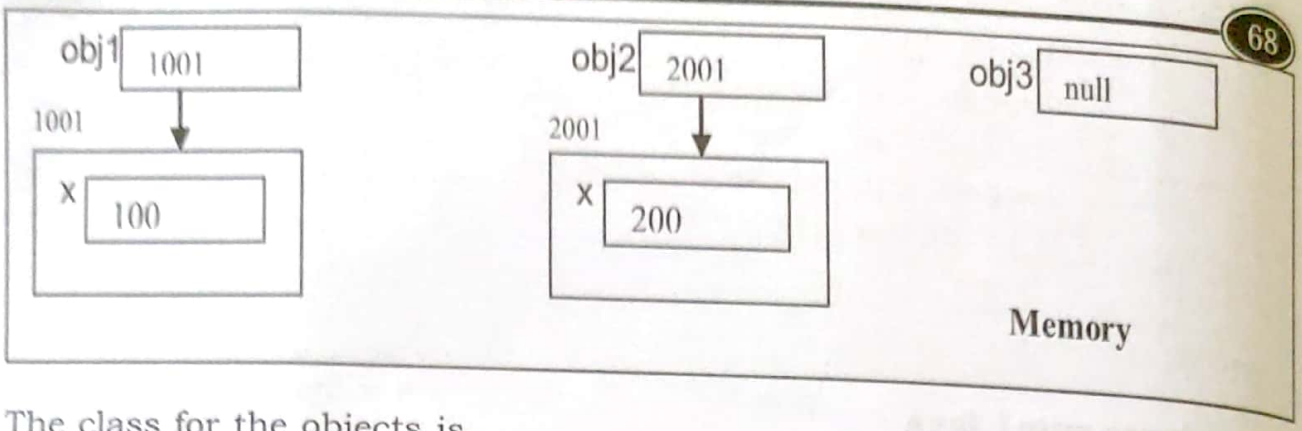
Methods not only can receive objects as parameters but it can also return objects and the return type of method should be of object's class type(class name).

Example:

A program to print addition of two objects by returning object.

Logic:

Let say there are three objects **obj1**, **obj2** and **obj3** where obj1.x=100, obj2.x=200 and obj3 is nothing as shown memory below.



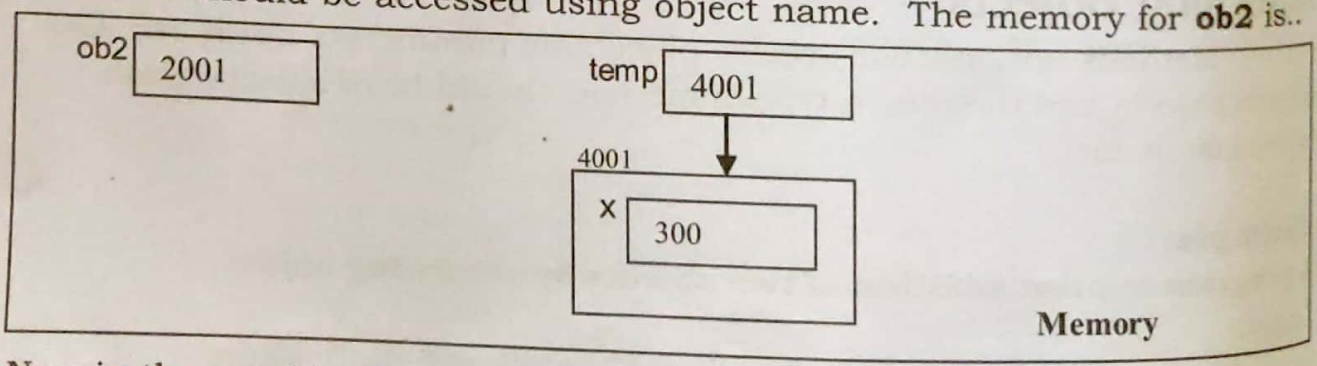
The class for the objects is...

```
class sample
{
        private int x;
        public sample sum(sample ob2)
        {
                sample temp=new sample();
                temp.x=x+ob2.x;
                return temp;
        }
}
```

Now we want to add **obj1**and **obj2** values and store in **obj3**. For this the statement we write is..

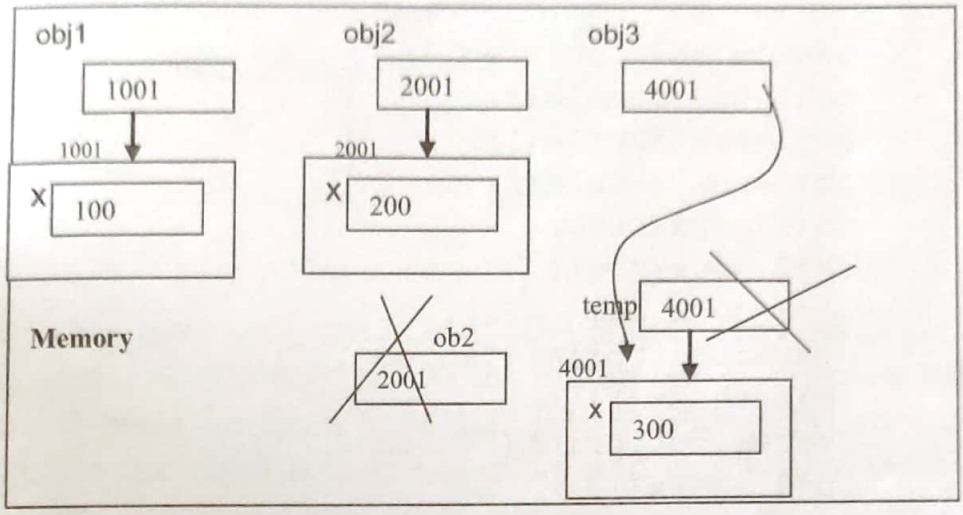
```
obj3=obj1.sum(obj2);
          4001
```

sum( ) is the method of sample class, therefore it should be called using object name and we are using **obj1** to call the method, because we want to store the results in **obj3** which is not passing to **sum( )**. The calling object **obj1** is passed **implicitly**(internally) to the **sum()** and whose members can be accessed directly without using object name. Where as other object(**obj2**) is passed **explicitly**(externally) to the parameter(**ob2**) and explicit object data members should be accessed using object name. The memory for **ob2** is..



Now in the **sum( )**, a **temp** object is created to store the results of calculation. The statement **temp.x=x + ob2.x;** adds **x** of implicit object(**obj1**) (100) and **x** of

ob2 (200) and result stores in **x** of **temp**. Finally **temp(4001)** object is returned to **obj3** and then **ob2**, **temp** reference variables are deleted. Therefore result in memory is..



The **sum( )** is returning the object (**temp**) of **sample** class, therefore return type of sum() method is **sample**.

If we print the **obj3.x** we get the output **x=300**.

Program

Bin>edit sum2.java

```
class sample
{
    private int x;
    public void setx(int a)
    {
          x=a;
    }
    public void showx()
    {
          System.out.print(" x =" + x );
    }
    public sample sum(sample ob2)
    {
          sample temp=new sample();
          temp.x=x+ob2.x;
          return temp;
    }
}
```

```
class sum2
{
    public static void main(String  argv[])
    {
          sample obj1,obj2,obj3;
          obj1=new sample();
          obj2=new sample();
          obj3=new sample();
          obj1.setx(100);
          obj2.setx(200);
          obj3=obj1.sum(obj2);
          System.out.print("\n obj1…");
          obj1.showx();
          System.out.print("\n obj2…");
          obj2.showx();
          System.out.print("\n obj3…");
          obj3.showx();
    }
}
```

Bin> javac sum2.java

Bin> java sum2
obj1...x=100
obj2...x=200
obj3...x=300

Class Loading

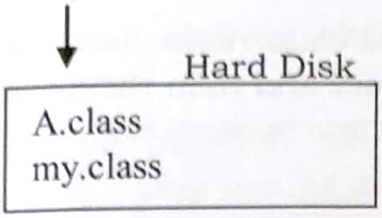
consider the java program

my.java

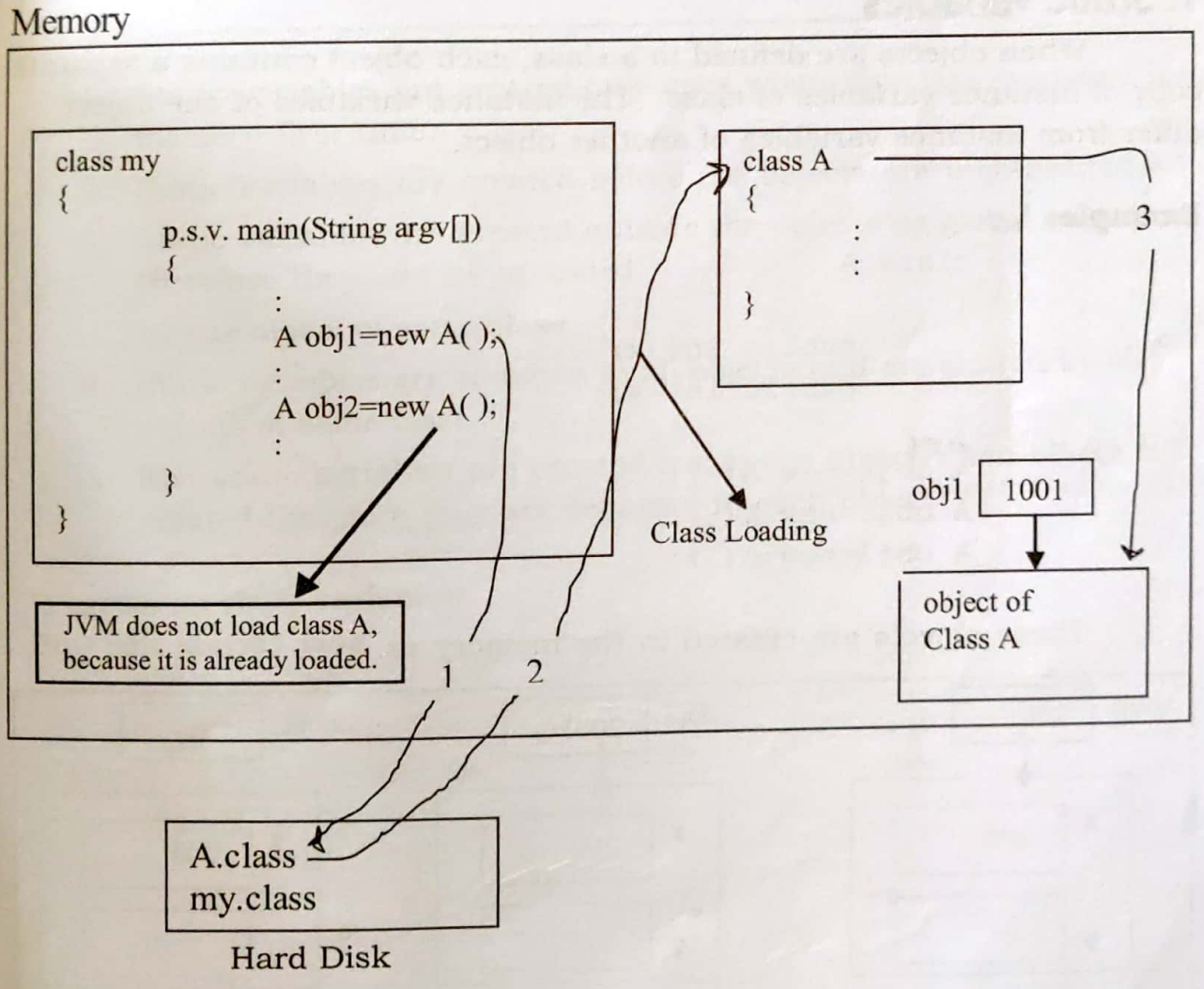
```
class A
{
      :
}
class my
{
      p.s.v. main(String argv[ ])
      {
            :
            A obj1=new A( );
            :
            A obj2=new A( );
            :
      }
}
```

When the above program is compiled we get two class files, they are **A.class** and **my.class**. These class files are stored on hard disk.

Bin>javac my.java



When the above program is executed only the **my.class** file is loaded into memory and main() is started. At the statement A obj1=**new A()**; the jvm first loads the **A.class** file from hard disk into ram(memory) and then object is instantiated to the **class A** in the memory and is allocated to obj1.



At the statement **A obj2=new A( );** the jvm does not load the **A.class** into memory instead the **class A** which is already loaded is used to create the object of **class A** and allocates to **obj2**.

In this way when jvm come across the class name first time then it loads the .class file from hard disk into memory and then the class is used. *This process of loading classes from hard disk into memory is known as* ***class loading***.

Static Members

Static members in Java are of three types

1. Static Variables
2. Static Methods
3. Static Block

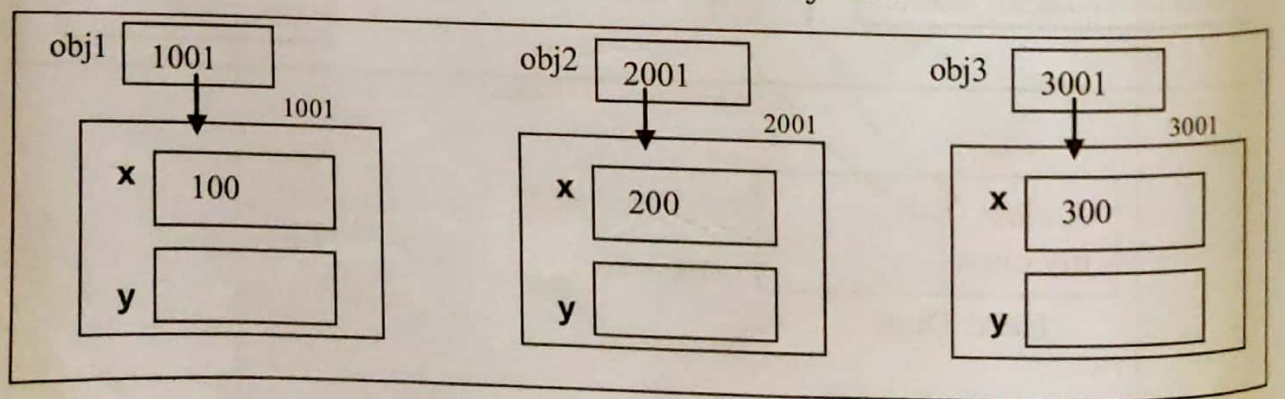
1. Static Variables

When objects are defined to a class, each object contains a separate copy of instance variables of class. The instance variables of one object differ from instance variables of another object.

Example:

```
class A
{
      public int x;
      public int y;
}
A obj1=new A( );
A obj2=new A( );
A obj3=new A( );
```

These objects are created in the memory as..



```
obj1.x=100; // stores 100 into x of obj1
obj2.x=200; // stores 200 into x of obj2
obj3.x=300; // stores 300 into x of obj3
```

The **obj1**, **obj2** and **obj3** contains a copy of **x** and **y** variables as shown in the memory.

In some situations we want to have common variables for all objects of same class. This can be achieved by declaring the instance variables of class as **static** and such varaibles are called as **static variables**.

Syntax:

```
class classname
{
        access static type  variable1[=value1], . . ;
                :
                :
}
```

1. Static variables are created only once when the class is loaded into memory first time.
2. Static variables are created before the objects are instantiated.
3. Static variables are created outside the objects as global variables therefore they can be accessed

   by any object of same class.
4. Static variables are common to all objects and are shared by all objects of same class.
5. Non-static variables are created inside the object when object is created therefore they are separate to each object.

Program on static variables:

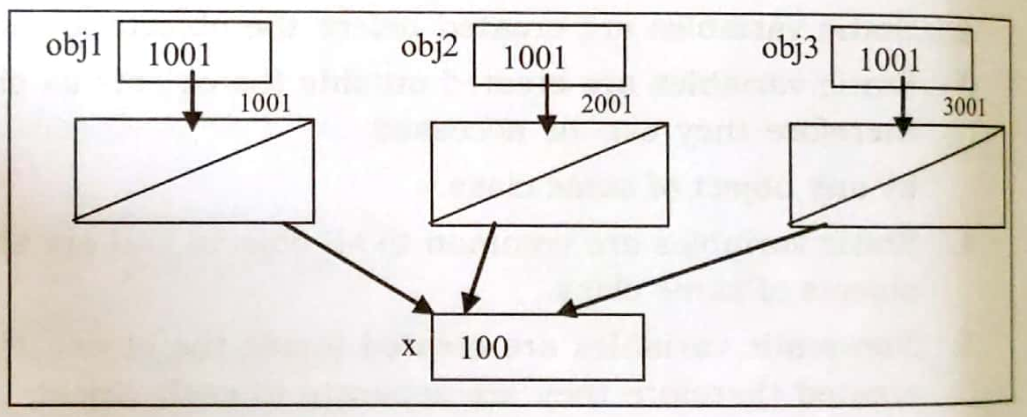
Bin>edit static1.java

```
class sample
 {
    static int x;
 }
 class static1
 {
    public static void main(String argv[] )
    {
          sample obj1=new sample();
```

```
        sample obj2=new sample();
        sample obj3=new sample();
        obj1.x=100;
        System.out.print("\n obj1.x="+obj1.x);
        System.out.print("\n obj2.x="+obj2.x);
        System.out.print("\n obj3.x="+obj3.x);
        obj3.x=200;
        System.out.print("\n After obj3.x=200");
        System.out.print("\n obj1.x="+obj1.x);
        System.out.print("\n obj2.x="+obj2.x);
        System.out.print("\n obj3.x="+obj3.x);
    }
}
```

Bin>javac static1.java
Bin>java static1

obj1.x=100
obj2.x=100
obj3.x=100
After obj3.x=200
obj1.x=200
obj2.x=200
obj3.x=200



Explanation:

In the **main()**, at the statement **sample obj1=new sample();** the jvm loads **sample** class into memory and then allocates the memory to static variable **x** and there after memory of **sample** object is allocated to **obj1**.

At the statement **sample obj2=new sample();** the jvm does not loads sample class into memory, because it is already loaded and memory of **sample** class object is allocated to **obj2** and similarly to **obj3**.

We can see in the memory that **x** static variable is common to all objects. The statement **obj1.x=100;** assigns **100** to **x** by the **obj1**.

The statements

```
System.out.print("\n obj1.x="+obj1.x);
System.out.print("\n obj2.x="+obj2.x);
System.out.print("\n obj3.x="+obj3.x);
```

prints **x** value as **100** for all objects as shown in above output.

The statement **obj3.x=200;** assigns **200** to **x** by the **obj3**.

The statements

```
System.out.print("\n obj1.x="+obj1.x);
System.out.print("\n obj2.x="+obj2.x);
System.out.print("\n obj3.x="+obj3.x);
```

prints **x** value as **200** for all objects as shown in above output.

In this way, static variables are created outside the objects as global and common variables and can be accessed by all objects of same class.

Another Example

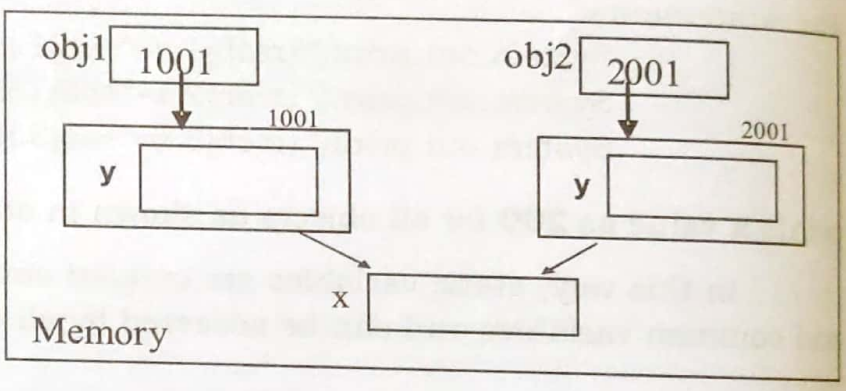
Bin>edit static2.java

```
class A
{
      private static int x;
      private int y;
      public void setvalues(int a,int b)
      {
            x=a;
            y=b;
      }
      public void showvalues( )
      {
            System.out.print("\n static x="+x);
            System.out.print("\n non-static y="+y);
      }
}
class static2
{
      public static void main(String argv[])
      {
```

```
            A obj1=new A();
            A obj2=new A();
            obj1.setvalues(100,200);
            System.out.print("\n using obj1…");
            obj1.showvalues( );
            System.out.print("\n using obj2…");
            obj2.showvalues( );
        }
    }
```

Bin>javac static2.java
Bin>java static2

```
using obj1...
static x=100
non-static y=200;
using obj2...
static x=100
non-static y=0
```



Explanation:

When the above program is executed, in the main(), at the statement **A obj1=new A();** the jvm loads the **A** class into the memory and after the jvm allocates the memory to static variable **x** and then memory of **A** class object containing non-static variable **y** is allocated to **obj1**.

At the statement **A obj2=new A();** the jvm does not loads the **A** class into the memory because it is already loaded and memory of **A** class object containing non-static variable **y** is allocated to **obj2.**

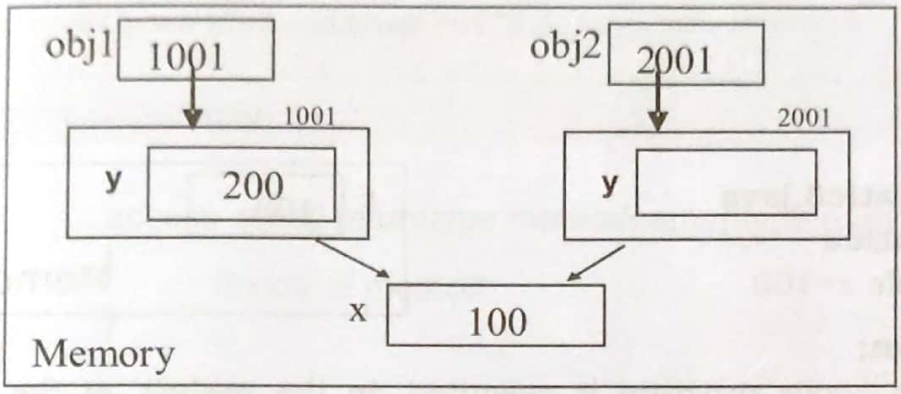
Therefore **x** is common to **obj1** and **obj2** where as **y** is separate to each object.

The statement,

obj1.setvalues(100,200); calls

```
public void setvalues(int a, int b)
{
   x=a;
   y=b;
}
```

As method is called by **obj1** object where **x** static variable is assigned with **a (100)** and **y** non-static of **obj1** is assigned with **b (200),** and the memory looks like this



The statement **obj1.showvalues()**; calls showvalues() which displays the values of **obj1**.

static x=100

non-static y=200

And the statement **obj2.showvalues()**; calls showvalues() which displays the values of **obj2**

static x=100

non-static y= 0 (As no value is assigned to **y** of obj2).In this way, it is proved that **x** is common variable for **obj1** and **obj2** where as **y** is separate to objects **obj1** and **obj2**.

Public static variables

If static variables are declared as **public** then such **public static variables** can be accessed using **classname** and **.(dot)** without using object name. This is possible because static variables are created at class loading and before the objects are created.

Syntax:

```
classname.publicstaticvariable
```

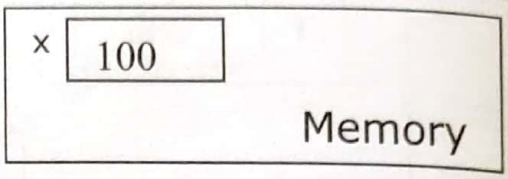
Example program:

Bin>edit static3.java

```
class sample
{
    public static int x;
}
class static3
{
```

```
    public static void main(String argv[])
    {
          sample.x=100;
          System.out.print("\n sample.x="+sample.x);
    }
}
```

Bin>javac static3.java
Bin>java static3
sample.x=100



Explanation:

When above program is executed, in the **main()**, at the statement **sample.x=100;** first the **sample** class is loaded into the memory and immediatly static variable **x** is created and is stored with **100**. Therefore **x** variable exist in memory without the existance of object it can be accessed using classname.

The statement **System.out.print("\n sample.x="+sample.x);** prints **x** value on screen.

Points to remember:

1) Static variables are also called as **class variables** because they can be accessed using class name and are created outside the object.

Differences between static and non-static variables.

1) Static variables are created outside the objects	1) Non-static variables are created inside the objects.
2) Static variables are created before the objects are created at the time of class loading.	2) Non-static variables are created when objects are instantiated.
3) Static variables are global to all objects of same class.	3) Non-static variables are local to each object.
4) Static variables are common to all objects.	4) Non-static variables are separate to each object.
5) Public static variables can be accessed using classname.	5) Non-static variables should be accessed using object name.
6) Static variables are known as class variables.	6) Non-static variables are known as instance variables.

2. Static methods

Methods of a class can be declared as static and such static methods can access only static members of the class and can not access non-static members(instance variables and methods).

Syntax:

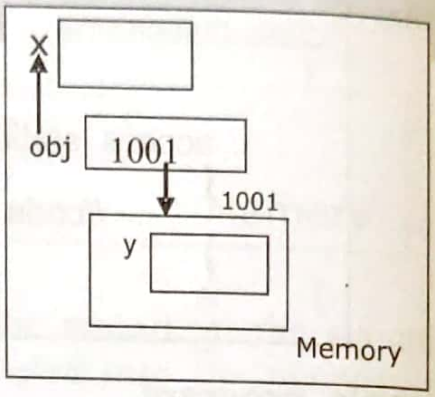
```
class classname
{
        :
    access static returntype methodname([type para1, ...])
    {
        //code of method
    }
}
```

Example program

Bin>edit static4.java

```
class sample
{
    private static int x;
    private int y; // y is non-static
    public void setvalues(int a, int b)
    {
        x=a;
        y=b;
    }
    public void showvalues() //non-static method can access
    {                        // both static and non-static.
        System.out.print("\n static x="+x);
        System.out.print("\n non-static y="+y);
    }
    public static void showstatic()
    {
        System .out.print ("\n static x="+x);
        //System.out.print("\n non-static y="+y); //Error:
        //y is non-static and can`t access in static method.
    }
}
class static4
{
    public static void main(String argv[])
    {
```

```
        sample obj=new sample();
        obj.setvalues(100,200);
        obj.showvalues();
        obj.showstatic();
    }
}
```



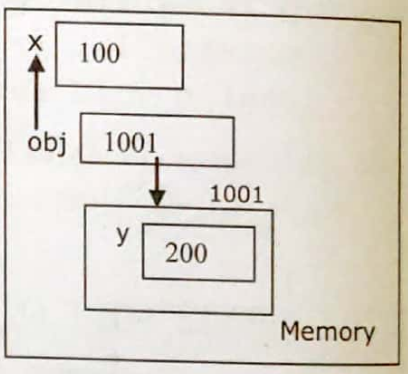
Bin>javac static4.java
Bin>java static4
static x=100
non-static y=200
static x=100

Explanation:

When above program is executed, in the main(), at the statement **sample obj=new sample();** the jvm loads the sample class into memory and allocates memory of static variable x and then the memory of sample object is allocated to obj as shown in above memory.

The statement obj.setvalues(100,200) calls

```
public void setvalues ( int a,  int b )
{
        x=a;
        y=b;
}
```



Which assign **a(100)** to static **x** and **b(200)** to non-static **y** as shown in above memory.

The statement **obj.showvalues()** calls.

```
public void showvalues()
{
        System.out.print("\n static x="+x);
        System.out.print("\n non-static y="+y);
}
```

Which displays
static x=100
non-static y=200
As **showvalues()** is a non-static method therefore it can access both static and non-static members.

Where as **obj.showstatic()** calls

```
public static void showstatic()
{
        System.out.print("\n static x="+x);
        //System.out.print("\n non-static y="+y); //Error: non-static
                                    // y can't access in static method.
}
```

Which displays

static x=100

Public static methods

If static methods are declared as public then such public static methods can be accessed without using the object name but using the classname and dot(.)

Syntax:

```
classname.methodname([arguments list...]);
```

Example program

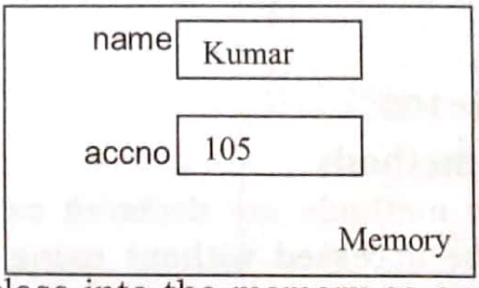
Bin>javac static5.java

```
class bank
{
    private static String name;
    private static int accno;
    public static void setvalues(String s, int ano)
    {
          name=s;
          accno=ano;
    }
    public static void showvalues()
    {
          System.out.print("\n Name=" + name);
          System.out.print("\n Accno=" + accno);
    }
}
class static5
{
    public static void main(String argv[])
    {
          bank.setvalues("kumar",105);
          bank.showvalues();
    }
}
```

```
Bin> javac static5.java
Bin> java static5
Name=Kumar
Accno=105
```

Explanation:

When the above program is executed, in the main(), at the statement bank.setvalues("kumar",105);



The jvm loads the bank class into the memory as a result all static variables are created and then **setvalues("kumar",105)** method is called

```
public static void setvalues(String s, int ano)
{
        name=s;
        accno=ano;
}
```

Where "kumar" is passed to **s** and 105 is passed to **ano.** Inside the method **s** and **ano** is assigned to **name** and **accno** as shown in memory.

The statement **bank.showvalues()** calls

```
public static void showvalues()
{
        System.out.print("\n Name=" + name);
        System.out.print("\n Accno=" + accno);
}
```

Which displays

```
Name=kumar
Accno=105
```

3. Static block

In java apart from static variables and static methods a new concept of static introduced and that is **static block**.

A class can have a static block which is used to initialize the static variables of class. A static block can access only static members of same

class and is automatically executed only once when class is loaded into the memory first time.

Syntax:

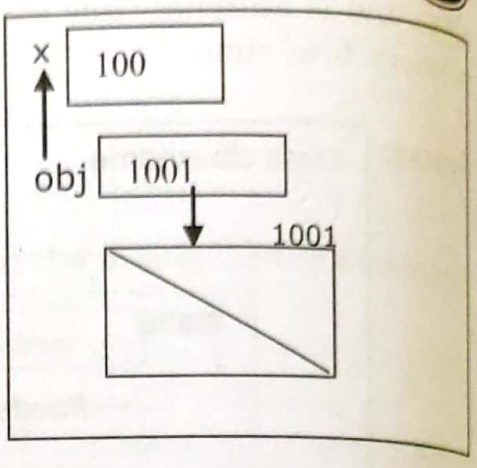
```
class classname
{
        :
        :
    static
    {
        ://code of static block
        :
    }
}
```

Program:

Bin> edit static6.java

```
class sample
{
    static int x;
    static
    {
        System.out.print("\n static block executing...");
        x=100;
        System.out.print("\n x="+x);
        System.out.print("\n static block terminating...");
    }
    public void show()
    {
        System.out.print("\n show()...");
        System.out.print("\n x="+x);
    }
}
class static6
{
    public static void main(String argv[ ]    )
    {
        System.out.print("\n*****************");
        sample obj=new sample();
        System.out.print("\n*****************");
        obj.show();
    }
}
```

```
Bin>javac static6.java
Bin>java static6
        ****************
        static block executes...
        x=100
        static block terminates
        ****************
        show( )...
        x=100
```



Explanation:

When the above program in executed, in the main(), at the statement **sample obj=new sample();** the jvm loads the sample class into the memory and creates **x** static variable and the static block executes as a result **x** is initialized with **100** and prints the output

```
static block executing...
x=100
static block terminating...
```

After executing the static block the memory of sample object is allocated to obj as shown in memory. The statement **obj.show();** displays **x=100** on the monitor.

Nested and Inner classes

In java, it is possible to define a class within another class and such classes are known as nested classes. These nested classes are of two types.

1. Non-Static nested classes or Inner classes
2. Static nested classes

1. Non-static nested classes or inner classes

A class defined in another class is known as inner class and such inner classes can be used only within in the scope of outer class i.e. inner class can't be accessed outside the outer class. The inner class can access the members of outer class directly without using the object name.

Syntax:

```
class outer_class
{
        .       :
    class inner_class
    {
            :
    }
            :
}
```

The inner class is not having static modifier therefore it is known as non-static nested class.

Example:

Bin>edit nestclass1.java

```
class A
{
   private int x=100;
   public void show()
   {
        B objb=new B();
        objb.display();
   }
   class B
   {
        public void display()
        {
             System.out.print("\n x= " +x);
        }
   }
```

```
}
class nestclass1
{
    public static void main(String argv[])
    {
          A obja=new A();
          obja.show();
    }
}
```

Bin>javac nestclass1.java

Bin>java nestclass1

x=100

Explanation:

In the above program **class B** is the inner class of **class A.** The inner class can access the members of outer class directly with out using the object name, therefore the **class B** is printing the instance variable(**x**) of **class A**, which is a valid access.

In the main(), the statement **A obja=new A();** creates the object **obja** and then the statement **obja.show();** calls the **show()** of **class A**. In the **show()** method the statement **B objb=new B();** creates **objb** object of **class B** and calls **display()** method which prints the value of **x** i.e **100** and this is a valid access.

2. Static nested classes

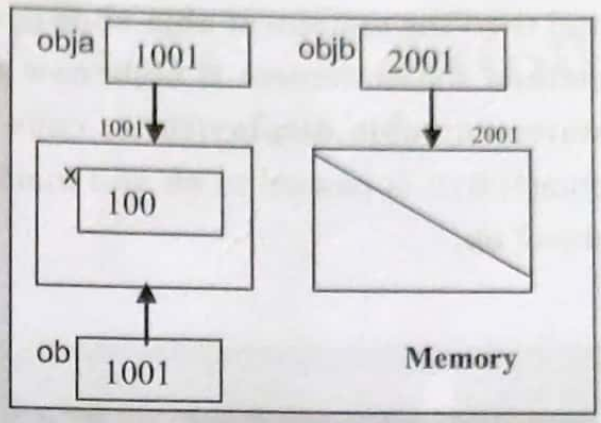
A class defined in another class with a static modifier is known as static nested class and such class scope is limited to outer class i.e. static nested class can not be accessed outside the outer class. The static nested class can not access the members of outer class directly but it should access indirectly using the object name.

Syntax:

```
class outer_class
{
            :
      static class nested_class
      {
            :
      }
            :
}
```

Example:

Bin>edit nestclass2.java




```
class A
{
    private int x=100;
    public void show()
    {
          B objb=new B();
          objb.display(this);
    }
    static class B
    {
          public void display(A ob)
          {
                System.out.print("\n ob.x= " + ob.x);
          }
    }
}
class nestclass2
{
    public static void main(String argv[])
    {
          A obja=new A();
          obja.show();
    }
}
```

Bin>javac nestclass2.java

Bin>java nestclass2

ob.x=100

Explanation:

In the above program **class B** is the static nested class of **class A.** The static nested class can access the members of outer class using the object name, therefore the **class B** is printing the instance variable(**x**) of **class A** using the object **ob.**

In the main(), the statement **A obja=new A();** creates the object **obja** and then the statement **obja.show();** calls the **show()** of **class A**. In the **show()** method the statement **B objb=new B();** creates **objb** object of **class B**. The statement **objb.display(this);** calls **display(A ob)** where **this**(address in obja(1001)) is passed to **ob** and inside the method the **x** is printed using the object **ob**.