OOPS CONCEPT

17/10/2017

Types of programming models

There are 3 types of programming model.

1) Process Oriented Programming model (POP)
2) Object Oriented Programming model (OOP) → Java
3) Aspect Oriented Programming model (AOP) → Spring Hibernet

The main building blocks of POP are → Global function

The main building blocks of OOP are → Object & Classes.

* Drawback of POP model-

1) Ownership maintainence is not there.
2) Which function is being called wrongly by which data member cannot be found easily.
3) Modifying the function can be complex. i.e can't be done.
4) Data integrity is not maintained.

=> The drawbacks of POP model is overcomed by OOP model-

* OOPS → The building blocks for OOP model is Object and classes.

Class is a combination of States and behaviours.

1) Ownership maintainence is there because member of class input as well as method has to be used only on that particular class object.

2) One class member cannot be accessed by other class.

Father of OOP → "Grady Booch"

First OOPL → "Small Talk"

Till now there is no 100% OOPL

JAVA is 99% OOPL because of a drawback i.e There is no support for multiple inheritance in the case of classes.

SHOT ON MI A2

* NOTE-

=> Java is considered as an "OOPL" but not a "procedure oriented programming language" because there are some drawbacks in POP language that is ownership maintainence is not there and global functionality mechanism is there and data integrity maintainence is not there and mainly which function is being called wrongly we cannot find out by which data member so there is no chance that highly popular programming language cannot be considered as POP languages.

=> To overcome the drawbacks of POP model, they have provided one more programming model that is called as OOPL.

=> In object oriented programming language there is a ownership maintainence that one class and its members has to be used by same class members only. (i.e object of the same class) and there is a restriction that one class and its member cannot be used by another class and for a class definition, we can create infinite number of object and that object can share the complexity task and make performance of application as high performance.

A part from the above principles, there are 4 more important principles and that are called as OOPS principle

1) Encapsulation.
2) Abstraction
3) Inheritence
4) Polymorphism.

1) Encapsulation-

It is the process of wrapping of data and total information of application can be collected or encapsulated in the class definition.

2) Abstraction-

It is the process of hiding internal information by providing external functionalities.

There are two concepts for Abstraction.

1) Abstract classes and methods
2) Interfaces

3) Inheritence-

It is used to maintain parent and child relationship among the multiple classes. It is used for reusing the properties of one class to another class directly. It is also a process of defining a new class by inheriting the properties of existing class.

4) Polymorphism-

Poly → many
morphism → forms
many + forms

It is the process of behaving one source differently at different situation.

+ => Polymorphism behaviour

eg:-

10 + 20 => addition (30)

"10" + "20" => concatination (1020)

* Object exhibits 3 properties-

1) State
2) Behaviour
3) Identity

1) State represents, how many set of inputs required for developing java application.
2) Behaviour represents, the functionality what you are performing with objects.
3) Identity represents as reoganization or representation of object for what purpose/functionality we can use.

Logically we can represent only state and behaviour of the object but we cannot represent identity of the object logically then state can be represented by using variables of the classes and behaviour can be represented by using method of the classes.

NOTE-

In any language supports for inheritance principle then only it is called as Object Oriented programming language

eg:- C++, java, etc.

If any language not supports for inheritance principle then it is called as Object based programming language

eg:- Java script (Scripting language), Angular JS.

Class	Object
1) It is a combination of state and behaviour of an object	1) It is an instance of a class definition.
2) Class is an expanded concept of structure because inside the class definition, we can use both inputs as well as functionalities.	2) Object is a realtime entity or it is called as runtime entity because the memory for object will be created at the time of execution of java application.
3) Class is a logical representation of object. There is no memory allocation for class definition.	3) Object is a physical representation of a class and its member
4) Class is a blue print / template of an object	4) Object and instance both are interchangable.
5) Syntax- <modifiers> class <className> { State + behaviour }	5) Syntax- ClassName refvar = new ClassName();

eg:-

```
Class Calculator {
    int [] number;
    Char [] symbols;
    String monitor;

    void add () { };
    void sub () { };
    void mul () { };
    void div () { };
```

State (i.e. variables)

behaviours (i.e methods)

* Requirement-

Collect 10 student details regarding Name of student, SID value, Contact number and display each and every student details.

Now,

Student → object

Input (State) → Name, SID, Contact, YOP } variables

Operation (behaviour) → display student details } method

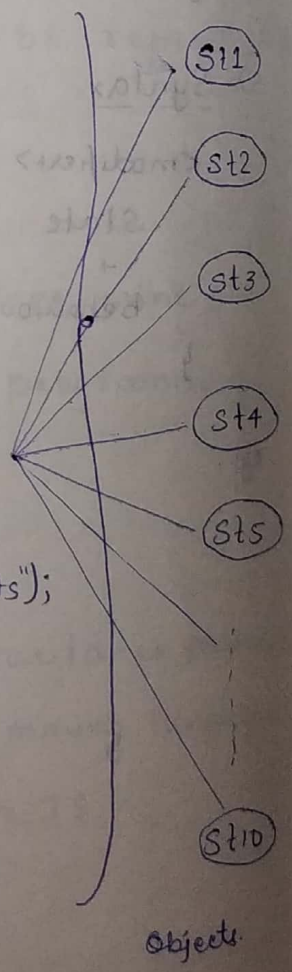
So,

defining class

```
class Student {
    String Name;
    int SID;
    long phone;
    int YOP;

    +

    void display(){
        System.out.println("Displaying Students");
        System.out.println(" Name);
        System.out.println(SID);
        System.out.println(phone);
        System.out.println(YOP);
```

state

behaviour



NOTE-

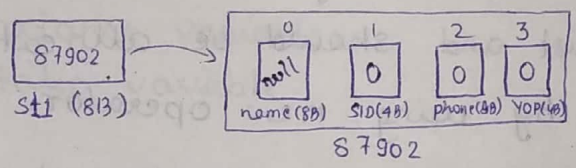
When only class is defined, its a logical operation representation.
i.e - Memory allocation will not happen.

When object is defined, its a physical operation representation.
i.e - Memory allocation will happen.

Internal operation while object creation-

```
Student st1 = new Student();
```

1) It is allocating 8B of memory for reference variable and assign's the default value (i.e null).



"JVM has to get information about class and its members". For that JVM will perform a process called as "Class Loading" by reading .class file

2) JVM will verify whether class is loaded or not.

3) If class loading already done then it will ignore.

4) In case if class loading has not happned, then JVM will perform class loading.

5) At the time of class loading, if any static variable is there then it will allocate memory.

6) By using new operator, JVM will create new memory for actual object.

7) At the time of object creation, Instance variable are allocating memory inside object memory.

8) The actual object address will be assigned to the reference variable.

NOTE-

→ While creating object of a class, we are assigning actual object into reference variable. So, in this syntax, actual object is different and reference variable is different because actual object syntax is going to allocate a new memory location and assigning the members memory allocation but reference variable just collecting the object address

→ So, depending on multiple reference variable, we should not count how many objects are created but depending on actual object syntax we should calculate how many objects are created for specified class. Since every actual object syntax must and should be allocated in seperate memory by using new operator.

→ There is no chance to calculate No. of objects based on reference variable because multiple reference variable can take the same memory address.

→ For any class description, infinite number of objects can be created and there is no restriction that specified number of objects only we have to create for any class definition.

* Members of Class-

There are 5 members of class

1) Variables
2) Blocks
3) Constructors
4) Methods
5) Inner classes

1) Variables-

In OOPS concept, variable is a member of a class which is used for representing state of an object.

=> There are 2 types of variables

(i) Instance variable
(ii) Static variable.

ex-

```
Class Hello {

    int a = 10 ;            -> instance premitive variable
    String str = "JLC";     -> instance reference variable

    static int b = 30;      -> Static premitive variable

    static String str1 = "SD";-> Static reference variable

}
```

(i) Instance variable -

Variable declaration without static modifier in the class definition is known as Instance variable.

Every instance variable will allocate the memory at the time of object creation.

=> Instance variable can be accessed in following ways.

(a) With reference variable containing actual object.

```
Hello hi = new Hello();
hi.x
```

(b) By using dynamic object

```
new Hello().x;
```

=> Instance variable has multiple copies of memory depending on the number of objects.

(ii) Static variable-

Variable declaration with static modifier in the class definition is known as static variable.

Every static variable will allocate the memory at the time of class loading. Only once the memory will be allocated for the static variable since for every class, class loading will happen only once.

Multiple objects of the class will point to single memory of static variable.

```
Class Hello{
    Static int b;
}
```

=> Static variable can be accessed in following ways.

(a) Using class name

```
Hello.b;
```

(b) Using null reference variable

```
Hello h1= null;
```

(c) Using reference variable containing actual value

```
Hello h2 = new Hello();
```

(d) Directly using Object reference.

```
new Hello().b ;
```

23/10/2017

NOTE-

In class definition (class level variable)

If we declare and do initialization in the second line then it will throw error.

If we have the requirement to initialize in the second line then Block concept is used.

2) Blocks-

Block is a member of class which is used for initializing the class level variables in the next line.

These are also known as initialization blocks.

Block should be declared by using parenthesis and it has no name.

{ } → Block

ex-

```
Class Hello {
    int x;
    {
      x = 12;
    }
    static int y;
    {
      y = 50;
    }
}
```

→ use of parenthesis makes the syntax valid else it will throw error

if static int y; y = 50; } → invalid, as no use of parenthesis.

Block does not have any name so developer cannot call the block explicitely and JVM will call the blocks for execution

=> Blocks are classified in two types-

(i) instance initialization block

(ii) static initialization block.

(i) instance initialization block-

Block declaration within the class definition without the static modifier is called as Instance block.

Instance block will execute automatically for each and every object creation.

(ii) static initialization block-

Block declaration within the class definition with static modifier is called as static block.

Static block will execute automatically at the time of class loading.

NOTE-

=> Inside Instance block we can initialize both instance and static variable but there is no gurantee that static variable will be initialized or not.

→ If we access the static variable with object reference then only static variable will be initialized or else it will not be initialized in the instance block.

=> Inside Static block we cannot initialize instance variable We can only initialize static variable because the static block will be executed at the time of class loading and at that time no instance variable will allocate the memory and without memory allocation how we can initialize the actual value for instance variable.

ex.

```
Class Test{
    public static void main(String arg[]){
        Hello h1 = new Hello();
            System.out.println(h1.y);
            System.out.println(h1.x);
    }
}

Class Hello{
    int x;
    static int y;
    {
     System.out.println("Hello 1B");
        x = 10;
        y = 20;
    }
}
```

block.

NOTE-

Block is used for another purpose-

=> In any scenario, if we need to verify whether the specific class is loaded or not. For that static block execution we can use.

i.e. If static block execution there, we can conclude that class is loaded elseit is not loaded.

=> We can check how many number of object are created for any class definition based on the multiple times of execution of instance block of the corresponding class.

24/10/2017

* Local Variable-

Variable declared inside the member of the class (Method, constructor, or block) is known as local variable.

→ Local variables are not the member of the class. These are member to member of the class

→ Local variable scope is upto the member of the class where the declaration has happned.

→ Local variables cannot be provided default value by JVM. Before using local variable developer has to initialize the actual value for Local variable.

→ Only final modifier is allowed while declaring the local variable of the class.

→ Local variable cannot be refered by the object reference or class name because local variable scope is up to the member of the class only and it cannot be accessed in different member of the class then how it will access outside the class definition.

* Local Blocks-

Block declaration within the member of a class is known as Local Blocks.

→ Local blocks will be executed when corresponding member is executing.

→ If we declare any variable inside the local block then we can access that variable upto that local block only and we cannot access it in different local block also.

NOTE-

Local variable or Local Blocks are not member of class. These are member to member of a class.

Q- WAP to collect 5 student details with SID value, Name of Student, email of student, phone number of student, age of student and display 5 student details.

Ans-

```
Class Details {
    public static void main (String arg[]) {
        Student st1 = new Student();
            st1.sid = 123;
            st1.name = "Vikash";
            st1.email = "vikash@gmail.com";
            st1.phone = 9876543210;
            st1.age = 23;
        System.out.println("Student 1 Details");
            st1.display();
        Student st2 = new Student();
            st2.sid = 345;
            st2.name = "Ramu";
            st2.email = "Ramu@gmail.com";
            st2.phone = 7654321098;
            st2.age = 25;
        Student st3 = new Student();
            :
            :
        Student st4 = new Student();
            :
            :
        Student st5 = new Student();
    }
}
class Student {
    int sid, age;
    String name, email;
    long phone;
    void display(){
        System.out.println ("SID:"+sid);
        System.out.println ("Name:"+name);
        System.out.println ("Email:"+email);
        System.out.println ("Phone:"+phone);
        System.out.println ("Age:"+age);
}}
```

3) Constructor-

Constructor is a member of a class which is used for initializing the instance variable of the class while creating the object.

→ Constructor is a special type of method and it is mainly given for initializing different set of input values for multiple objects which are created for any class description at the time of object creation.

→ Constructor name must and should be same as class name and there is no return type for constructor.

→ Constructor will be executed automatically by the JVM while creating each and every object for any class.

```
Hello hi = new Hello();
```

↳ It's not a method
It's a constructor.

NOTE-

To know the class of member we can use the code on the command line.

```
> javap Hello
```

→ Whenever any class description doesnot contain constructor then by default JVM will provide default constructor to create the object of the class.

→ If explicitely constructor definition is provided then JVM will not provide any default constructor.

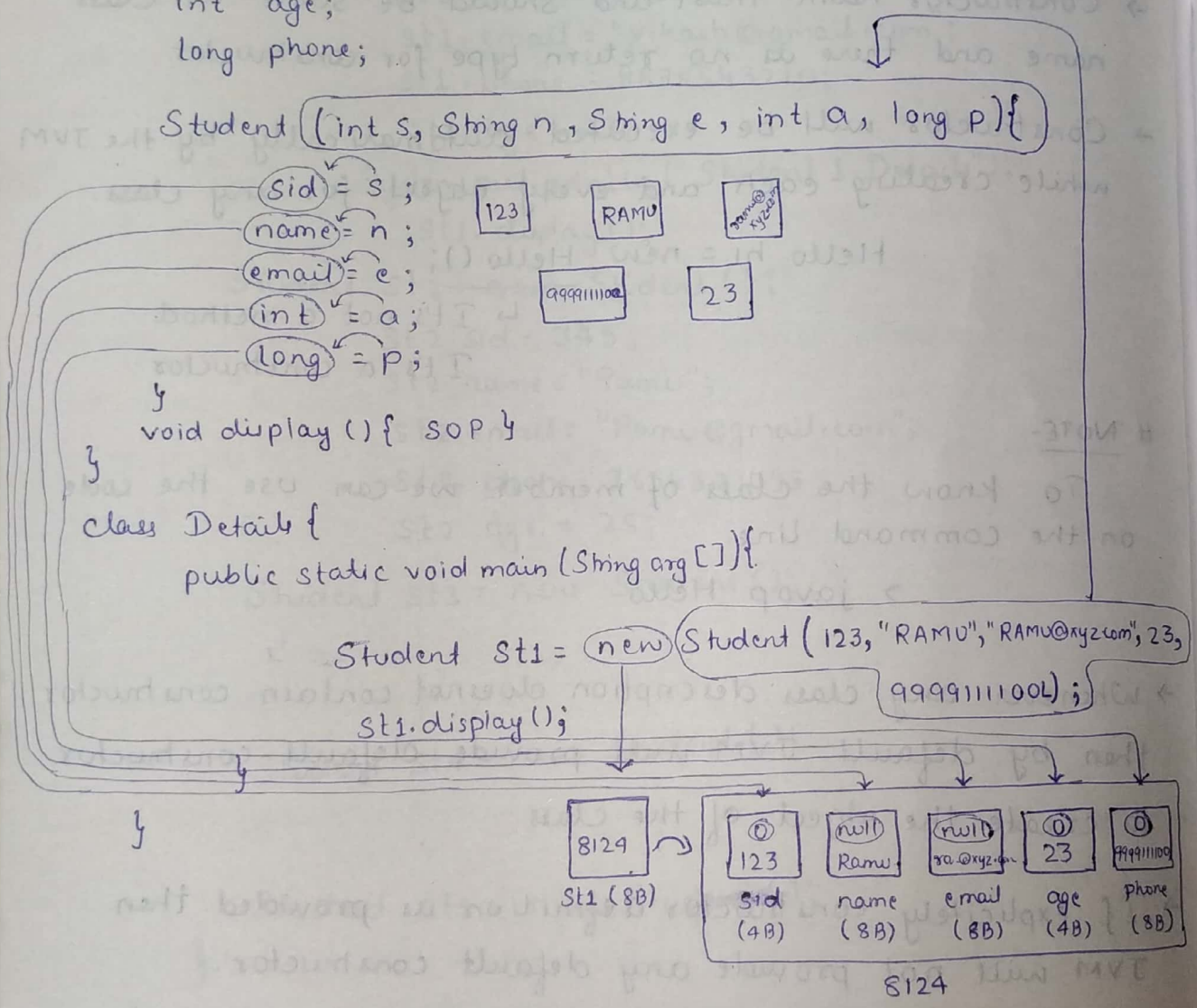
⇒ Constructors are classified in two types-

1) Default constructor
2) Parametrized constructor.

2) Parameterized constructor-

Memory allocation.

```
Class Student {
    int sid;
    String name;
    String email;
    int age;
    long phone;

    Student (int s, String n, String e, int a, long p){
        sid = s;
        name = n;
        email = e;
        int = a;
        long = p;
    }
    void display () { SOP }
}
Class Details {
    public static void main (String arg[]){
        Student St1 = new Student (123, "RAMU", "RAMU@ryz.com", 23,
                                   9999111100L);
        St1.display();
    }
}
```



NOTE-

→ Multiple constructors can be defined within the class definition either parameterized constructor or default constructor but between multiple constructor there should be a difference in the either count of parameters or datatypes of parameters or order of parameters.

→ In case, between multiple constructor if there is no difference in count of parameter or datatype of parameter or order of parameter then it will be considered as duplicate constructor definition.

→ Constructor's cannot be invoked explicitely then in any scenario if you want to call the constructor then before constructor name, provide "void" keyword, then it will be considered as normal method but not a constructor.

→ The main difference between constructor and method is we cannot invoke constructor explicitely. JVM will execute automatically but there is no automatic execution for method must and should called explicitely by developer and every constructor (matching constructor definition) will be called automatically at the time of every object creation.

→ Constructor is used for initializing instance variable of the class but not static variable because multiple object will point to single copy memory of static variable so, if we initialize static variable, values with constructor definition then which ever the last object is created and which value is provided for static static variable in constructor then it will affect for previous objects of the class also. So, this is a reason why we cannot initialize static variable with the help of constructor.

Static variable initialization with constructor.

ex.

```
Class Test {
    public static void main (String arg[]){
        System.out.println ("Main Started");
    Student st1 = new Student (123, "Ramu", "CUSAT");
    Student st2 = new Student (456, "Shyam", "JNTU");
    Student st3 = new Student (789, "Ghanshyam", "SVU");

        st1.display();
        st2.display();
        st3.display();
    }
}

Class Student {
    int sid;
    String name;
static String clgname;  ⟶  its a static so o/p will differ.
    Student (int s, String n, String cn){
        sid = s;
        name = n;
        clgname = cn;
    }
    void display(){
        System.out.println ("\n *** ___ ***");
        System.out.println ("SID :"+ sid +"\n Name :"+ name +"\n ClgName:"+
                                    clgname);
    }
}
```

O/P

```
*** ___ ***
SID : 123
NAME : RAMU
CLGNAME: SVU

*** ___ ***
SID: 456
NAME: SHYAM
CLGNAME: SVU
```

```
*** ___ ***
SID: 789
NAME: Ghanshyam
CLGNAME: SVU
```

here because of static variable the clgname variable is always pointing to the same memory allocation hence SVU value is called for all.

25/10/2017

* this keyword-

"this" keyword is a predefined keyword which is holding current object reference and by the use of "this" keyword we can access current class variable.

=> "this" keyword is known as final instance reference variable

In any class definition, if we have declare two variables with the same name, that is one at the local scope and one at the class level scope which is considered as duplicated variable declaration in the class definition. So, we can access the local variable directly and we can access the class level variable by using "this" keyword. Since "this" keyword contains the current object reference and "this" keyword is always used for accessing current class level variable.

Ex-

```
class Test{
    public static void main (String arg[]){
        Hello h1 = new Hello();
            h1.show();
    }
}

class Hello{
    int a = 99;
    void show(){
        System.out.println ("Hello show()");
        String a = "JLC";
        System.out.println (a);
        System.out.println (this.a);
    }
}
```

O/p.

```
Hello show()
JLC
99
```

=> This keyword can be used for 3 purposes-

1) Accessing current class level variable.

eg:- this.a
this.b
this.c

2) Accessing current class level method.

this.m1();
this.m2();
this.show();

3) Calling one constructor from another constructor definition.

this(); → default constructor
this(10); → one argumented constructor
this(10,20); → two argumented constructor.

* Constructor chaining process-

Calling one constructor from another constructor.

NOTE-

"This" keyword cannot be used inside the static context (Static block or static method) since it is an instance final reference variable.

ex-

```
class Student{
    int sid;
    String name;
    Student (int sid, String name){
        this.sid = sid;
        this.name = name;
    }
void display(){
    System.out.println("SID:"+sid+"\n Name:"+name);
    }
}
```

* Memory Allocation-

① Class Student {

int sid;

String name;

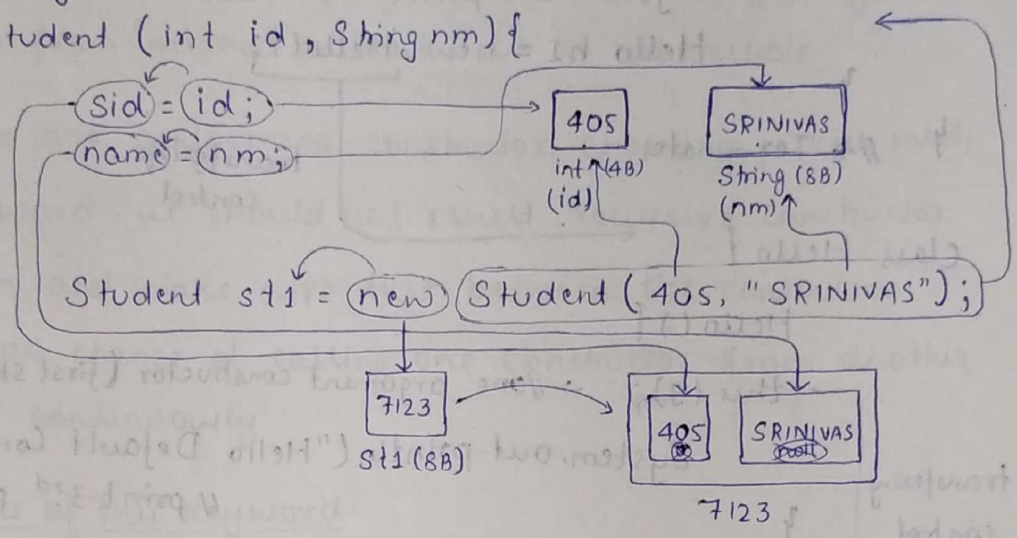
Local scope → Student (int id, String nm) {

Sid = id;

name = nm;

}

Object → Student st1 = new Student (405, "SRINIVAS");



② Class Student {

int sid;

String name;

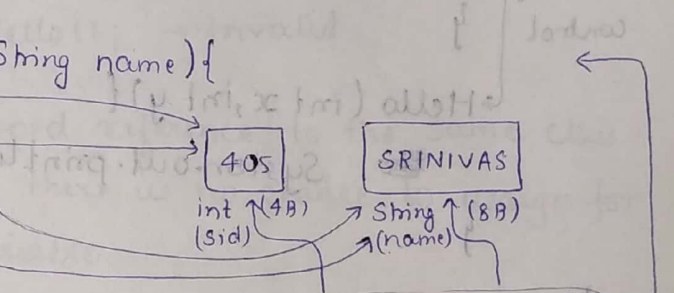
Local scope → Student (int sid, String name) {

Sid = sid;

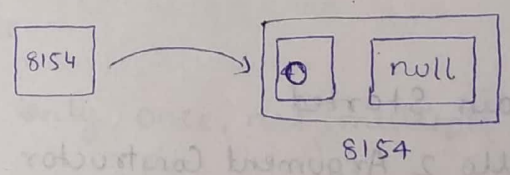
name = name;

}

Object → Student st2 = new Student (405, "SRINIVAS");



Since "sid" scope upto local so, object will take default value and will not load the parameter values.



③

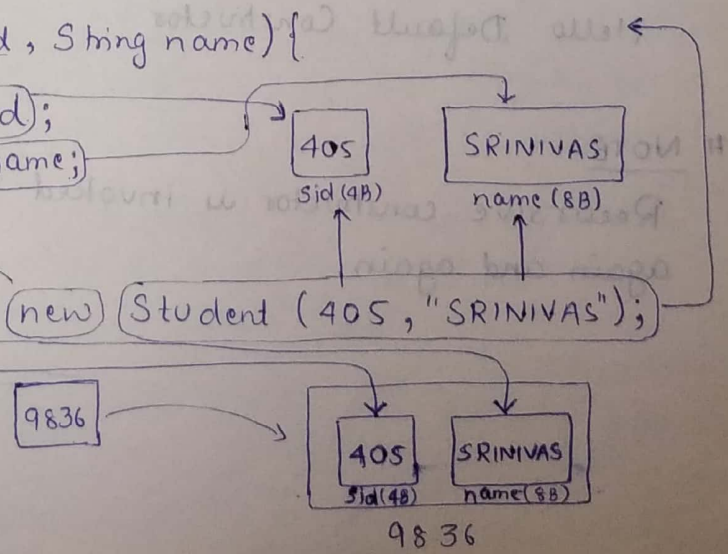
Local scope → Student (int sid, String name) {

this.sid = sid;

this.name = name;

}

Object → Student st3 = new Student (405, "SRINIVAS");



with this keyword, we can access class level variable in local scope with having same data type.

ex- Constructed chain process-

```
class Test {
    public static void main (String arg[]){
        System.out.println ("Main Started");
        Hello h1 = new Hello();
    }
}  //→ Terminates

class Hello {
    Hello () {
        this (9);  → //one argument constructor (first statement)
        System.out.println ("Hello Default Constructor");
                                    // print 3rd place
    }
    Hello (int x) {
        this (10,20);  // two argument constructor (first statement)
        System.out.println ("Hello 1 Argument Constructor");
                                    // print 2nd place
    }
    Hello (int x,int y){
        System.out.println ("Hello 2 Argument Constructor");
                                    // print 1st place
    }
}
```

transfering control

transfering control

transfering control

control goes to previous Calling Class

O/p

Main Started
Hello 2 Argument Constructor
Hello 1 Argument Constructor
Hello Default Constructor

NOTE-

Recursive constructor is invoked if constructor is called again and again.

NOTE-

-> Whenever we are calling one constructor from another constructor by "this" keyword, it should be 1st statement in another constructor definition or in case if you try to call constructor from method or block, it is not possible.

-> When we are performing constructor chaining process with "this" keyword, it should not result recursive constructor invocation and make sure that between the constructor there is no chance of calling one constructor from another constructor continuously.

* Limitations of "this" keyword-

1) this keyword is an instance final reference variable which is holding the current object reference and it cannot be modified.

this = new Hello(); -> invalid

2) We can assign "this" keyword reference to the same class reference variable only but there is no chance to assign for different class reference variable.

3) We can access this keyword by using same class name but not the other class name

4) Class loading is happening only once, not multiple times

Q. Prove that whether class is loading once or not by

```
Class Test {
        public static void main (String arg[]) throws Exception {
            System.out.println ("Main Started");
            System.out.println (Hello.a);
            System.out.println ("Delete Hello.class file & Enter");
            System.in.read();
            Hello h1 = new Hello();
            h1.show();
        }
}

Class Hello {
        static int a = 10;
            int b = 20;
            static {
                System.out.println ("Hello SB");
            }
            {
                System.out.println ("Hello IB");
            }
            void show(){
                System.out.println ("Hello Show()");
            }
}
```

O/P →

Main Started
Hello SB
10
Delete Hello.class file and press Enter
↵ (After deleting)

Hello IB
Hello Show()

* Method-

Methods are the members of the class to represent behaviour / functionality / operation of an object.

Syntax-

Syntax is called as Method declaration / Signature / Definition / Tempelate

```
< Modifiers>* < Method Return type > < Method Name > < Method parameter>*
{
  // body of the method
}
```

default method return type is one keyword i.e void.

1) Method declaration-

```
void show (){
    // Statement;
}
```

Once the method is declared, it will not be executed automatically by the JVM. So, even memory will also not be allocated. So, after method declaration we need to perform method calling.

2) Method calling-

```
< Method Name > < parameter i/p value if there >;
```

Based on method calling only, memory for the method definition is allocated on the top of the stack memory (part of JVM memory) and it will be executed with statement.

There is no automatic execution or invocation for the method which are declared inside the class like constructor. So, explicitely developer has to call the method for execution

=> Methods are classified into 2 types-

1) Instance Method
2) Static Method

1) Instance Method-

Method declaration without static modifier is known as Instance method. Every instance method will allocate memory at the time of object creation.

→ To access Instance Method.

```
class Hello{
    void m1(){
        System.out.println("m1 in Hello");
    }
}
```

a) By using reference variable containing actual object.

```
Hello h1 = new Hello();
h1.m1();
```

b) By using dynamic object.

```
new Hello().m1();
```

2) Static Method-

Method declaration with static modifier is known as Static Method. Every Static Method will allocate the memory at the time of class loading.

→ To access Static Method-

```
class Hello{
    static void m2(){
        System.out.println("Hello Show");
    }
}
```

a) By using class name directly.

```
Hello.m2();
```

b) By using null reference variable.

```
Hello ref = null;
ref.m2();
```

c) By using the reference variable containing actual object

```
Hello h1 = new Hello();
h1.m2();
```

d) By using the dynamic object.

```
new Hello().m2();
```

NOTE-

-> Instance Method, we can access both instance variable and static variable because at the time of object creation both variable will allocate the memory and we can access both variables inside. Since we are calling Instance method with object reference only.

-> Static Method, we can only access static variable and we cannot access the instance variable directly because Static method will be called at the time of class loading then the instance memory allocation will not happen at the time of class loading. So, we can access the instance variable inside static method by creating the object reference.

```
class Hello{
    int a = 90;
    static int b = 80;
        static void display(){
            System.out.println("Hello->display()");
        //  System.out.println(a); -> invalid
            System.out.println(new Hello().a); ->valid
            System.out.println(b);
}
```

=> Members of Method are of two types :-

1) Method Return type (mandatory)

2) Method parameter (optional)

1) Method Return type-

Method return type can be provided in 2 scenarios

Case (i) -

Once the method is declared to perform some operation we are going to get some resultant value That resultant value we need to display within the method definition and no need to return any value from the method to the caller of the method then declare the method return type as void which is considered as default method return type.

Case (ii) -

Once the method is declared to perform some operation we are going to get some result ant value. That resultant value we need to return to the caller of the method and the return value will be reused then in that case we need to provide the method return type as other than void.

Method return type should be the value datatype which is returning from the method

ex-

```
class Test {
    public static void main (String arg[]){
        Hello h1 = new Hello();
        int res = h1.add();
        System.out.println("Return value is :"+ res);
    }
}
class Hello {
    int add(){
        int a=10;
        int b=20;
        int res = a+b;
        return res;
    }
}
```

2) Method parameter- 26/10/2017

Method parameters should be used for reusing the logic of Method for different set of inputs when multiple times if we call the method definition.

Method parameter should be provided at the the time of declaring the method and the actual value for the parameter should be provided while

=> Method parameter is classified into types-

i) Formal parameter / Formal Argument

ii) Actual parameter / Actual Argument.

```
void add (int a, int b){
    // Statement          -> Formal parameter
    }

add (10, 20){
    // Statement          -> Actual parameter
    }
```

=> While calling any method we must and should check whether the Actual parameter value are matching with formal parameter or not in terms of

a) Count of parameter

b) Datatype of parameter

c) Order of parameter.

=> Any type of parameter either for method or constructor is considered as Local variable to that scope.

ex-

```
class Test{
    public static void main (String arg[]){
        Hello h1 = new Hello();
            h1.add (99,1);
    }
}
class Hello{
    void add (int a, int b){
        int res = a+b;
    System.out.println ("Result is :"+ res); } }
```

* Method Overloading-

It is the process of defining the same method for multiples with different set of input values to perform same operation with same method name.

But to perform method overloading of any specific method, difference in the parameter list is mandatory.

=> Parameter list difference should occur in terms of

a) Count of parameter

b) Datatype of parameter

c) Order of parameter

=> Rules needs to be followed for Method Overloading.

a) Method name should be same

b) Method parameter should be different in terms of

(i) Number/count of parameter

(ii) Datatype of parameter

(iii) Order of parameter

c) Method return type can be anything.

NOTE-

Only change in the return type between multiple method is not a method overloading and is considered as duplicate method

Method overloading must and should require difference in the parameter list.

* Limitations of Method Overloading-

1) Whenever we are calling a method in method overloading. First priority it will check for matching method. If in case matching method is not there then it will be called from compatibility parameter.

2) When ever it is checking for compatability parameter. If it matches for only one call method then that call method will be binded and will be executed.

In case for one call method, more than one method is matching then compiler is in confusion to call which method and it will throw ambigous problem.

3) So, whenever we perform method overloading among the multiple method, make sure that ambigous problem should not occur and if ambigous problem occur then there is no solution provided by java vendor.

NOTE-

Method overloading mechanism is achiving polymorphism principle of OOP principle.

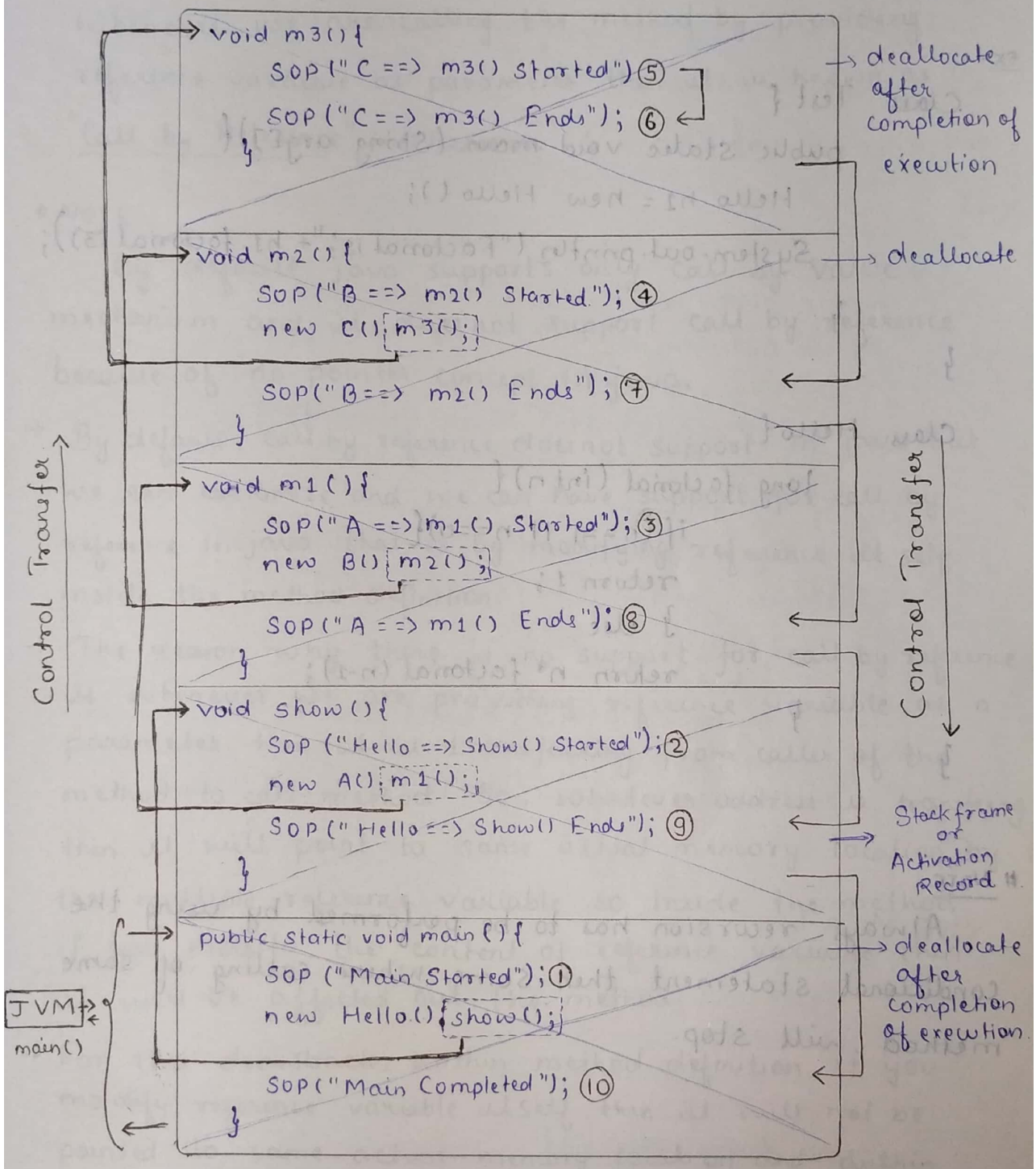
Q How memory will be allocated on the top of stack memory.

Code-

```
Class Test {
    public static void main (String arg[]){
        System.out.println ("Main Started");
        Hello h1 = new Hello ();
            h1.Show();

        System.out.println ("Main Completed");
    }
}

Class Hello{
    void show (){
        System.out.println ("Hello ==> Show() Started");
        new A().m1();
        System.out.println ("Hello ==> Show() Ends");
    }
}

Class A {
    void m1 (){
        System.out.println (" A ==> m1() Started");
        new B().m2();
        System.out.println ("A ==> m1() Ends");
    }
}

Class B {
    void m2() {
        System.out.println ("B ==> m2() started");
        new C().m3();
        System.out.println ("B ==> m2() Ends");
    }
}

Class C{
    void m3(){
        System.out.println (" C ==> m3() Started");
        System.out.println ("C ==> m3() Ends");
    }
}
```

=> Memory allocation-

Once the method definition is completed then it will delocate the stack memory



Output

Main Started
Hello ==> Show() Started
A ==> m1() Started
B ==> m2() Started
C ==> m3() Started
C ==> m3() Ends
B ==> m2() Ends
A ==> m1() Ends
Hello ==> show() Ends
Main Completed

* Recursion-

The process of calling one method from its definition only is called as Recursion Mechanism.

If we perform recursion concept without any conditional check then it throw error "stackoverflow error".

ex-

```
Class Test {
        public static void main (String arg[]){
            Hello h1 = new Hello ();
            System.out.println ("Factorial is:"+ h1.factorial (3));
        }
}

Class Hello {
        long factorial (int n){
                if (n==1 || n==0){
                return 1;
                } else
                return n* factorial (n-1);
        }
}
```

NOTE-

Always recursion has to be performed by using the conditional statement then some where calling of same method will stop.

* Call by reference and Call by value.

Whenever we are calling the method by providing primitive values as parameter then it is known as "Call by Value" mechanism.

Whenever we are calling the method by providing reference variable as parameter then it is known as "Call by Reference" mechanism.

NOTE

By default java supports only call by value mechanism and it does not support call by reference because of no pointer concept in java.

→ By default, call by reference does not supports in java but we can customize and we can have support for call by reference in java that is by modifying reference it self inside the method definition.

→ The reason why there is no support for call by reference is whenever we are providing reference variable as a parameter then it is transferring from caller of the method to call method. So, whatever address is transfering then it will point to same actual memory location by the multiple reference variable so inside the method if you modify the content of reference variable then it will be affected out the method.

→ For this drawback, within method definition, if you modify reference variable itself then it will not be pointed to same actual memory location and within the method if you modify the content then it will not be affected outside the method.

-> But in the case of C, C++, if you provide pointer variable in parameter from caller of method to called method then address will not be passed. It will send the context from the specified memory location. So there is a support of call by reference with pointer but not with reference variables and java does not have pointer and it is managing call by reference with reference variable.

ex-

```
class Test {
    public static void main (String arg[]){
    System.out.println("***** CBV *****");
        int x = 90;
        System.out.println("Main Started"+x);
        Hello h1 = new Hello();
        h1.show(x);
        System.out.println("Main Completed:"+x);
    System.out.println("**** CBR ****");
        Hai h = new Hai();
          h.a = 120;
        System.out.println("Main Started"+h.a);
        h1.display(h);
        System.out.println("Main Completed:"+h.a);
        }
    }
class Hello {
        void show(int x){
        System.out.println("Show() Starts:"+x);
        x = x+100;
        System.out.println("Show() Ends:"+x);
        }
        void display(Hai h){
        System.out.println("display() Starts:"+h.a);
        h.a = h.a+100;
        System.out.println("display() Ends"+h.a);
        }
    }
class Hai {
        int a;
    }
```

28/10/2017

* Var-Arrags-

It is the new feature added from Java 5 and it is used for declaring either method parameter or constructor parameter. Whenever we are not sure about the number of parameter input value or else to collect multiple value as the parameter input values.

Syntax-

datatype ... < Variable Name >;

↳ ellipsis

After using Var-Args

> We can directly provide primitive values
> No anonymous array declaration is required.
> Method can be called both ways. i.e directly or anonymous.

ex-

Without Var-Args declaration-

```
class VarArgs{
    public static void main (String[] args){
        Hello h1 = new Hello();
        h1.add (new int[] {1,2});          // anonymous array not directly
        h1.add (new int[] {10,20,30,40});  // anonymous array
    }
}

class Hello{
    void add (int arr[]){                  //non directly
        int res = 0
        for (int x: arr){
        res += x;
        }
        System.out.println ("Result is:"+res);
    }
}
```

Ex- with Var-Args declaration-

```
class VarArgs {
    public static void main (String ... args){
    Hello h1 = new Hello();
    h1.add (1, 2);                    // directly, no anonymous array
    h1.add (10,20,30,40);             // directly
    }
}

class Hello {
    Void add (int ... arr){           // using Var-Args for direct
      int res = 0;                      input.
      for (int x : arr){
        res += x;
      }
    }
}
```

NOTE-

> Internally Var-Args parameter is converted as the array declaration in the method at the time of compilation by the JVM.

> Whenever Var-Args is the parameter then it is a variable argument.
i.e - without providing any input value for the VAR-ARGS we can call the method.

i.e

```
h1.add ();
  ↳ blank / empty input value.
```

Rules to use Var-Args for Methods-

1) Var-Args must and should be declared as a parameter for method or constructor but not as the normal variable.

```
class Hello{
    int ... a;          -> invalid
    Static int ... b;   -> invalid.
}
```

2) Only one Var-Args should be taken in the parameter list

```
void show (int ... x , int ... y) {   -> invalid
```

3) Always Var-Args has to be placed as a last parameter in the parameter list.

```
void show (int ... x , int a, int b) {   -> invalid.
void show (int x, int a, int ... b) {   -> valid.
```

* Explaination to the default syntax-

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

> public- It is an Access Modifier which represents that we can access the main method anywhere in the application

> static- It is declared as static method because whenever jvm is envoking a method then it can be accessed by the class name and Object Reference is not required.

> void - The main method no need to return any value to the caller of the method and there is no use.

> main - It is the method name taken from low level language as application initializer or launcher.

> String- Command line arguments are the dynamic input value collection / collecting at the time of execution may be any type of value and all types of value will be collected by String datatype.

* Class Loading and Object Creation-

```
class Lab {
    public static void main (String arg[]){
    System.out.println ("Main Started");
    Hello h1 = new Hello();
    }
}

Class Hello{
    int a = 10;
    Static int b= 20;
    {
    System.out.println("Hello => IB");
    }
    static {
      System.out.println ("Hello => SB");
    }
}
```

1) Hello h1 = new Hello();

It will allocate 8B of memory for any reference variable assign default value with null.

null
8B (h1)

2) JVM will verify whether class is loaded or not.

3) If class is not loaded then JVM will perform class loading by reading Hello.class file created after compilation.

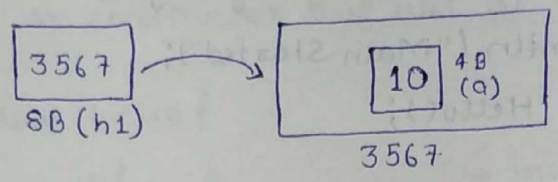
4) If class is already loaded then it will ignore.

5) At the time of class loading if static variables are there then they will allocate memory.

20	→ "Hello =>SB"	(1) execution.
4B(b)		

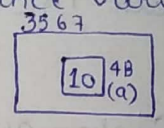
6) It will execute static block if it is there.

7) By using new operator, it will allocate memory for object inside the heap memory.



8) It will invoke the matching constructor to create the object. Invoking means memory is created for constructor definition but not executed

9) At the time of object creation, If Instance variables are there then it will allocate memory.

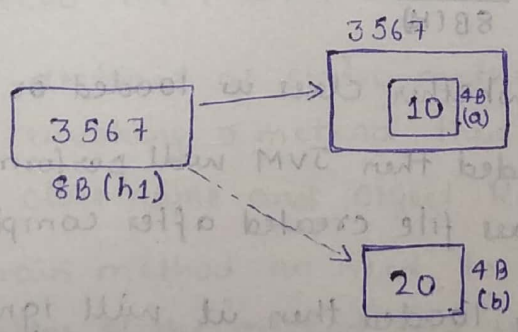


10) It will come to execute the constructor statement. But before execution first line of constructor, JVM will verify any instance block is there or not.

11) If ~~If~~ Instant Block is there then that will be executed ~~first~~ then only constructor statement will be executed

"Hello => 1B"

12) The actual object address has to assign to the reference variable.



Constructors	Methods
1) Constructor is used for initializing the Instance member of class with different set of input values at the time of object creation	1) Method is used for representing the behaviour or operation of the object.
2) Constructor name should be same as class name	2) Method name should be any name which will be a valid identifier.
3) Constructor donot have return type	3) Method must and should provide return type. Atleast void should be provided.
4) There is an automatic execution of constructor for every object creation	4) Method will not execute automatically by JVM.
5) Constructor cannot be called explicitely by developer	5) Method will be executed and will allocate memory based on explicit calling by developer.
6) Inside the constructor we can provide empty return statement	6) Inside the void method only we can provide return statement
7) Constructor cannot return any value to the caller	7) Methods can return value to the caller of the method by specifying the method return type other than void.
8) Constructor cannot provide with instance and static ~~blocks~~ scope	8) Method can be provided with instance and static scope.
9) Without returntype we can declare the constructor definition	9) Without return type we cannot declare the methods.

ex.

```
Class Hello{
    Hello(){} --> Constructor
    void Hello(String arg[]){} -> Method
}
```

30/10/2017

* Principle of OOPs →

1) Encapsulation
2) Abstraction
3) Inheritance
4) Polymorphism

1) Encapsulation-

Wrapping of total information about an object with the help of class definition.

Class is an Encapsulated data of an object.

2) Inheritance-

> Parent and child relationship should be maintained among the multiple classes is known as Inheritance

> The process of defining the new class by inheriting the property of existing class is known as Inheritance.

Inheritance is the major principle of OOPs concept

* Any language supports Inheritance
  ↳ Object Oriented programming language

* Any language supports not the Inheritance
  ↳ Object based programming language.

⇒ Class of its members can be accessed upto class definition only and outside the class it can be accessed with the same class object reference

⇒ Inheritance principle is used for reusability mechanism of one class member in another class definition without redeclaring and to reduce the length of source code and duplicate piece of code.

Existing class is known as Parent Class or
Super Class or
Base Class.

Newely defined class is known as Child Class or
Derived Class or
Sub Class.

=> To maintain the inheritance relation between two class we have to use one keyword i.e. extends.

=> Once we created inheritance relation between two classes then one class member can be accessed in another class member definition directly.

=> Super class member can be accessed with the sub class reference but sub class newly defined members cannot be accessed with the super class reference because the member will be inherited from Super class to sub class but not from sub class definition to super class.

=> Between two classes if we want to maintain IS-A relationship then we can provide inheritance relation, but to maintain HAS-A relationship we should use one class as a member of another class definition.

* Types of Inheritance-

1) Simple Inheritance
2) Multi Level Inheritance
3) Hierorchial Inheritance
4) Multiple Inheritance
5) Hybrid Inheritance
6) Cyclic Inheritance.

#NOTE-

Object is the default super class for any type of class

* Limitations -

-> To restrict the inheritance relationship for any type of class we need to declare the class as the final declaration.

-> Sub class newly defined members cannot be accessed with the super class reference.

-> Whenever two classes are maintaining inheritance relationship then any property of super class will be inherited to sub class except private member of the class

<->

NOTE -

-> In case of blocks, when we are accessing super class static variable with only super class name then super class will be loaded and super class static block will be executed.

-> Whenever we are accessing super class static variable with sub class name, then only super class will be loaded and its static block will be executed.

-> Whenever we are accessing sub class static variable with sub class name then sub class should be loaded but before loading sub class first of all its super class should be loaded. if it is not loaded then after sub class will be loaded.

-> In the case of blocks and constructors with inheritance then before moving the priority to sub class 1st of all super class has to perform the operation either in class loading or object creation.

* Super-

Super keyword is used for accessing the super class inherited member in the sub class.

-> Super keyword is an instance final reference variable and it is used for accessing always the members from direct super class only.

-> Super keyword cannot be used in static content i.e either static block or static method but we can access the super class static variable with the super keyword

=> Super keyword can be used for three purposes-

1) To access the direct super class variables in sub class.

```
super.a;
super.b;
```

2) To access the direct super class method ~~to~~ in sub class

```
super.m1();
super.m2();
```

3) To invoke super class constructor from sub class constructor

```
super();        -> default
super(10);      -> 1 argumented
super(10,20);   -> 2 argumented.
```

NOTE-

In the case of inheritance relationship constructors will never inherited from super class to subclass because JVM is providing one default statement i.e. "super();" in each and every constructor definition to call the super class default constructor then if constructor are inherited from super class to subclass at that time JVM no need to provide default statement in every constructor.

=> So whenever a sub class object is created then from the sub class constructor, by default it is invoking super class default default constructor only and if super class have default constructor definition then sub class object will be created succesfully in case if super class constructor doesn't have default constructor and argument constructor are available then while creating sub class object it will give error message because without default constructor in super class the sub class constructor cannot be executed or else there is no chance JVM will provide the default constructor for super class since explicit constructors definitions are there.

=> To overcome this error while creating subclass object we should call the super class constructor explicitely from the sub class constructor definition by using "super" keyword according to the requirment

=> Inside the subclass constructor definition if we call the super class constructor explicitely by using "super" keyword then JVM will not provide the default statement "super();" but whenever we are calling super class constructors from sub class constructors using 'super' keyword it should be 1st statement only.

Q- By default private member will not be inherited from super class to sub class. Is it possible to initialize the private variable of super class while creating subclass object. If it is yes, write the code if no, specify the reason.

Ans 1-

```
class Inheritance {
        public static void main (String arg[]){
        Sub subref = new Sub();
        }
}
class Super{
        private int x;
        {
        x = 100;
        }
}
class Sub extends Super { }
```

Ans 2-

```
class Inheritance {
    public static void main (String arg[]){
        Sub ref = new Sub(10);
    }
}
class Super{
        private int x;
        Super(int val){
        x=val;
        }
}
class Sub extends Super{
        Sub(int x){
        Super (x);
        }
}
```

31/10/2017

* Packages-

Package is a namespace which is used as a folder specification and it is used for collection of multiple class files which are used for another project from one project.

case 1-

G:\JA-B11\OOPS\Examples ===> Hello.java → show()
===> Hello.java → display()

Duplicate file cannot be allowed in the same location of file system.

case 2-

Reusability of one project related file in to another project without redeclaration then we can specify the old project related files in a specified folder and we can reuse them.

Package will be used for placing the duplicate file in the same location.

Package is used for reusing one project related file to another project without redeclaration.

Package is used for specifying the scope of the class and its members by using access modifier concept.

Package is a folder specification. Insted of creating manually we need to create by logical description and whenever we need to use the class and its member then first of all JVM will perform class loading by using byte code file but not .java file

Package is a collection of .class file but not .java files which is allowed to collect automatically by creating a folder based on logical statement.

Package declaration statement-

Syntax-

package <packageName>;

⇒ It should be provided at the starting of the source code before declaring the class.
Only one package declaration is allowed in the source code.

* Without package declaration statement-

javac srcfileName.java → compilation
java .class fileName → execution
} normal syntax.

* With package declaration statement-

-d <directory> specify where to place generated class file.

javac -d <location> <srcfileName>.java → compilation

i.e- javac -d . Lab.java ↵ → here . represents current workspace location.

java < fully Qualified Class Name > → execution

i.e java p1.Lab ↵

⇒ Fully qualified class Name → Package Name. Classfile Name.
p1. Lab

Restriction-

Package name should not starts with java keyword.

⇒ It will give Security exception error.

* Exploring Compilation-

Case 1 - In current workspace directory

G:\JA-B11\OOPS\Examples

Relative Location → javac -d . Hello.java

Absolute Location → javac -d G:\JA-B11\OOPS\Examples Hello.java

Case 2 - In current workspace directory but in different folder
i.e "classes" folder

Relative location → javac -d classes Hello.java

Absolute Location → javac -d G:\JA-B11\OOPS\Examples\classes Hello.java

Case 3 - In parent folder of current workspace
i.e OOPS folder.

Relative Location - javac -d ../ Hello.java

Absolute Location - javac -d G:\JA-B11\OOPS Hello.java

Case 4 - In subfolder of parent folder
i.e bin folder.

Relative Location - javac -d ../bin Hello.java

Absolute Location - javac -d G:\JA-B11\OOPS\bin Hello.java

Case 5 - In parent to parent folder to current workspace
i.e JA-B11 folder

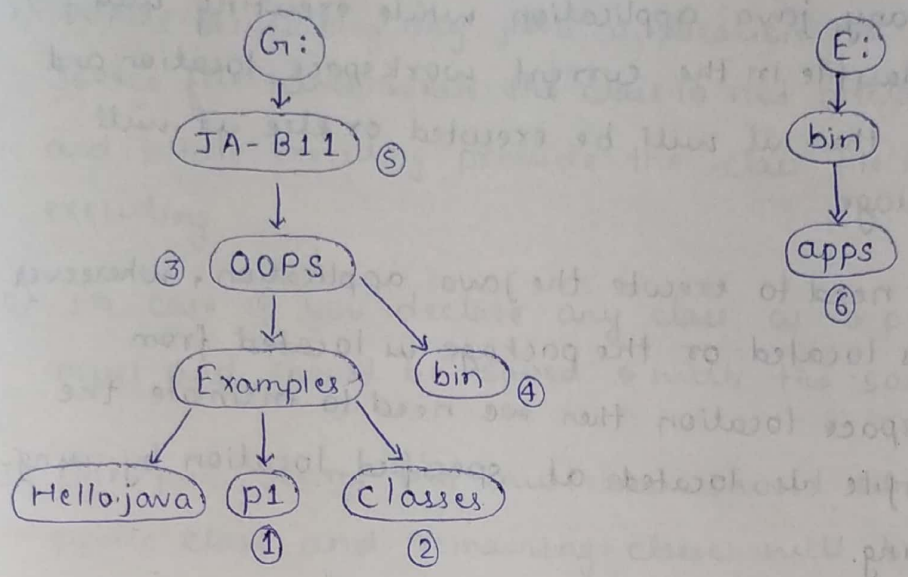
Relative Location - javac -d ../../ Hello.java

Absolute Location - javac -d G:\JA-B11\ Hello.java

Case 6 - In different drive
i.e E:\bin\apps

Absolute Location - javac -d E:\bin\apps Hello.java

=> Folder Structure-



ex-

```
package p1;

Class Test{
    public static void main (String [] args){
        System.out.println ("Main Started");

        A ref1 = new A();
        B ref2 = new B();
    }
}

Class A {
    {
     System.out.println ("A -> IB");
    }
    static {
    System.out.println ("A -> SB");
    }
}

Class B {
    {
    System.out.println ("B -> IB");
    }
    static {
    System.out.println ("B -> SB");
    }
}
```

The above code will create a package with name p1 in the current workspace with 2 class files A.class and B.class including the Test.class file also.

* Exploring ClassPath-

1) By default any java application while executing will search the classfile in the current workspace location and if it is there then it will be executed or else it will give error message.

2) Whenever we need to execute the java application, wherever the classfile is located or the package is located from current workspace location then we need to intimate the JVM that classfile is located at specified location by using class path setting.

=> We can provide class path setting in 3 ways:-

1) -cp option
2) temporary classpath setting.
3) permanent classpath setting.

1) -cp option-

Only one application should be executed by specifying the classpath location then while executing itself we need to provide -cp option.

```
java -cp <location of Classfile/package> .classfileName;
```

i.e

```
java -cp E:\Workspace\OOPS p1.Test
```

2) For the total command prompt whatever we are executing multiple source files, the class files/packages are located at the different location then insted of using -cp option for every application execution, we can set the temporary classpath upto the command prompt works.

```
setclasspath= %classpath% ; E:\Workspace\OOPS;
```

3) For total one project, each and every classfile or package is located in different location but not in current workspace then we can execute the application by setting permanent classpath (But its not a recomended option).

NOTE-

1) While developing any java application we can provide any source file name when the class is not declared with public and while executing provide the .class file name only for executing.

2) In case if you declare any class as a public then it must and should be saved with the same as class name.

3) One java source file must and should contain only one public class and remaining classes will become as non public classes.

4) There is no chance to provide multiple public classes in a single source file but we can declare multiple public classes with multiple source files that is
   -> Single source file contains Single public class
   -> Single package contains Multiple public classes

5) By default while executing any java application, JVM will search for class file or package in the current workspace location only. In case, if class file is located in different location then we need to intimate about the class file location to the JVM by setting the class path.

* Accessing one package class files in another - 01/11/2017

=> When multiple classes belongs to same package then we can access one class in another class directly either public class or non public class because all the multiple class files are available in same folder.

=> Whenever we need to access one package classes in different package then we cannot access directly since one folder's .class files cannot be accessed in another folder because JVM will search for the .class file within the same folder location. In case if we want to access one package information in different package then we need to follow some rules and regulations.

1) The class and its member must be public scope
2) The package should be available in class path.
3) We need to specify package information in different package package source file and can be done in 2 ways.
   - (a) Using fully qualified class Name
   - (b) Using import statement.

(a) Fully qualified class Name -

While specifying the package information of the class in different packages source file, we need to use fully qualified class name every time. If we specify fully qualified class name only once, its not enough. So its very complex to use the fully qualified class name.

eg:-

```
package com.jlc.p2;
class Lab {
    public static void main (String arg[]){
    com.jlc.p1.Hello hello = new com.jlc.p1.Hello();
    hello.show();
    com.jlc.p1.Hai hai = new com.jlc.p1.Hai();
    hai.display();
    }
}
```

(b) Import statement-

import is a keyword which is used for specifying the package information to access one package classes in another package. But we need to use import statement only once, that is after the package declaration statement in java source file.

Syntax-

① import <packageName>.<ClassName> → Single type import statement

In this import statement only specified class from specified package can be accessed

② import <packageName>.* → Multi type import statement

In this all the public classes from specified package can be accessed.

ex-

```
package com.jlc.p1;
public class Hello{
    public void show(){
        System.out.println("Hello -> show()");
    }
}
```

Class file Hello in package p1

```
package com.jlc.p1;
public class Hai{
    public void display(){
        System.out.println("Hai -> display()");
    }
}
```

Class file Hai in package p1

```
package com.jlc.p2;
import com.jlc.p1.Hello;
import com.jlc.p1.Hai;
class Lab{
    public static void main (String arg[]){
        Hello hello=new Hello();
        hello.show();
        Hai hai=new Hai();
        hai.display();
    }
}
```

Accessing class file Hello & Hai of package p1 in package p2.

* NOTE.

To access one package information into different package we are specifying with import statement then if we import or write import statement of one package in different package then all the classes of one package can not be loaded in different package until the member of the class is used.

ex.

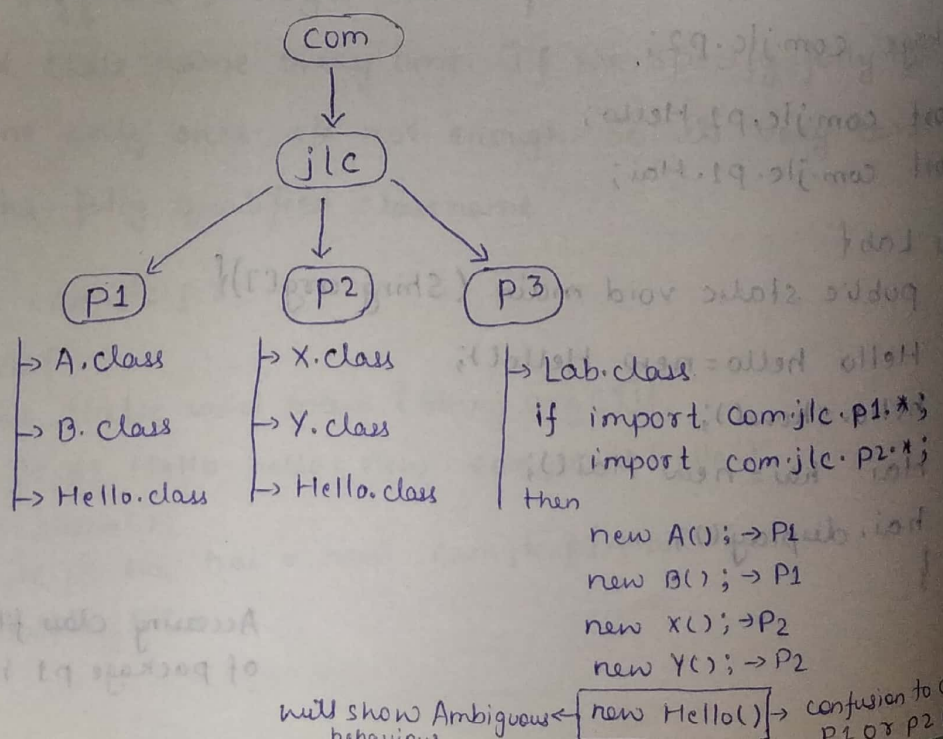
```
package com.jlc.p2;

import com.jlc.p1.Hello;
import com.jlc.p1.Hai;
Class Lab{
    public static void main(String arg[]){
        System.out.println("Main Started");
    }
}
```

O/P

Main Started.

* Limitations of import statement-

If 2 packages contains common class, if we access the common class directly then it will give ambiguous problem. Then to overcome that ambiguous problem we need to access in the following ways.



=> Ambiguous Solution- (Normal import)

The ambiguous error can be rectified with the help of fully qualified class name representation.

ex-

```
package com.jlc.p3;

import com.jlc.p2.*;
import com.jlc.p1.*;

public class Lab{
    public static void main (String arg[]){
        System.out.println ("Main Started");
        com.jlc.p1.Hello h1 = new com.jlc.p1.Hello();
        h1.show();
        com.jlc.p2.Hello h2 = new com.jlc.p2.Hello();
        h2.display();
    }
}
```

ex-

```
package com.jlc.p3;

import com.jlc.p1.Hello;
import com.jlc.p2.*;

Class Lab{
    public static void main (String arg[]){
        System.out.println ("Main Started");
        Hello h1 = new Hello();
        h1.show();
        com.jlc.p2.Hello h2 = new com.jlc.p2.Hello();
        h2.display();
    }
}
```

NOTE-

For the package we are using * in import statement, we should use fully qualified name for the members of class of that relevent package or else it will not be accepted (i.e error)

NOTE-

If we specify import statement to access one package info in different package then all the classes of import package cannot be loaded in different package because once we imported one package in different package there is no gurantee that we can access all the classes of specified package in different package. So, if all the classes are loaded from import package then it will be a waste of memory.

To reduce this wastage of memory, they have given option that after importing ~~and~~ any package which ever the class member we are using then that class will be loaded and other classes will not be loaded.

=> Static Import-

It is the newly added feature from java 5 onwards and it is used to access only public static members of the class of one package to another package.

* Import statement is classified into 2 types.

1) Normal import
2) Static import

1) Normal import-

It is used for accessing public class from one package to another package.

2) Static import-

It is provided for accessing the public static member of the classes from one package to another package directly.

=> Static import syntax is of 2 types-

1) Single type static import statement-

Syntax-

```
import static <packageName>.<className>.<static member>;
```

2) Multi type static import statement-

Syntax-

```
import static <packageName>.<className>.*;
```

ex-

```
package com.jlc.p1;
public class Hello{
  public static int A=90;
  public static void m1(){
    System.out.println("Hello->m1()");
  }
  static void m2(){
   System.out.println("Hello->m2()");
  }
}
```

class Hello in p1 package

```
package com.jlc.p1;
public class Hai{
  public int B = 80;
  public static void m3(){
    System.out.println("Hai->m3()");
  }
  static void m4(){
    System.out.println("Hai->m4()");
  }
}
```

class Hai in p1 package

```
package com.jlc.p2;
import com.jlc.p1.Hello.*;
import com.jlc.p1.Hai.*;
class Lab{
    public static void main (String arg[]){
      System.out.println(A);
//    System.out.println(B);      -> can't be called because its not static
      m1();
//    m2();                       -> can't be called because its not public
      m3();
//    m4();                       -> can't be called because its not public
    }
}
```

* NOTE-

1) Whenever we are specifying static import for any specified class to access the public static member directly then if common variables and common variables are not there then we can access directly.

In case if any common variable or common method is there then while accessing it will throw ambiguous error.

2) To over this ambiguous error we have the solution same as the normal import statement, that is we can refer common static variable and common static method with fully qualified class name or with single type static import statement to access the common variable and common method we can use normal import and we can access the member with its corresponding class names.

3) Whenever we are accessing one package information in another package by using import statement or static import statement then while accessing any member or any class, first of all it will verify in current class package. If it is there then it will access from current class and current package only or else the first priority it will go for imported package and imported class.

It is similar like whenever we have a local variable, class level variable and super class inherited variable with same name. First of all it will access from local scope then if it is not there in local scope then priority will go for class level variable, if class level variable is not there then it will go super class level inherited.

* JAR FILE-

It is defined as collection of package placed in a single folder by using zip algorithm.

one project files relate to another project

package → collection of class files

collection of package → jar file creation

=> To create jar file-

jar -cf nameofjarfile pkg1 pkg2 pkg3 .....

or jar -cvf nameofjar file pkg1 pkg2 pkg3 .....

=> To extract jar file-

jar -xf name of jar file

or jar -xvf name of jar file

* Can we execute out program without extracting jar file-

Yes, we can execute the program without extracting the jar files.

→ First we need to set the class path of the jar file.

set classpath=%classpath%;E:\Workspace\ClassAccess.jar

→ After setting the class we can run the java file with the help of fully qualified class name and java file name.

java jar.com.r1.Lab ↵

* Source code for jar file creation

=> Student.java package com.jlc.p1

```
package com.jlc.p1;

public class Student{
    static {
      System.out.println("Student SB in com.jlc.p1");
    }
    {
      System.out.println("Student IB in com.jlc.p1");
    }
}
```

=> Employee.java package org.edu.jlc

```
package org.edu.jlc;
public class Employee{
    static{
      System.out.println("Employee SB in org.edu.jlc");
    }
    {
      System.out.println("Employee IB in org.edu.jlc");
    }
}
```

=> Test.java package sri.io.q1

```
package sri.io.q1;
public class Test{
    static{
      System.out.println("Test SB in sri.io.q1");
    }
    {
      System.out.println("Test IB in sri.io.q1");
    }
}
```

=> Lab.java package jar.com.r1

```
package jar.com.r1;
import com.jlc.p1.*;
import org.edu.jlc.*;
import sri.io.q1.*;
public class Lab{
    public static void main (String args[]){
        System.out.println ("Main Started");
        Student st = new Student();
        Employee emp = new Employee();
        Test tst = new Test();
    }
}
```

-> Now create the jar file by collecting multiple package.

-> jar -cf ClassAccess.jar com org sri

jar -cvf ClassAccess.jar com org sri -> verbose.

-> Once the jar file is created then we can access all the multiple package for executing the application in two way:

① By extracting jar file (not recomended)

-> jar -xf Access.jar {i.e ClassAccess} -> extract

-> java -d . Lab.java -> recompilation because we donot included the jar folder while creating the .jar file

-> java jar.com.r1.Lab -> execution

② By setting the classpath to the jar file.

-> keep Access.jar file and Lab.java file at any location {i.e - E:\workspace\

Now set classpath

-> set classpath=%.classpath%.; E:\Workspace\Access.jar ↵

Now execute the file with fully qualified class name.

-> java jar.com.r1.Lab

* NOTE- 02/11/2017

javac -d . Lab.java

It will give error if we compile the jar file without setting the class path file location.

Once after setting the classpath the main file will be executed without any problem.

javac -d . Lab.java ↵

java jar.com.r1.Lab ↵

* Packages benifits-

1) Duplicate file can be placed within the same location

2) Reusability of one project files in another project without redeclaration.

3) Specify the scope of class and its class member.

* Access Modifiers-

These are used for specifying the scope of the class and its members.

=> There are 3 predefined keyword which are provided as the Access modifiers

(i) private

(ii) protected

(iii) public

=> According to scope Access modifier is classified in 4 type

1) Private scope

2) Default scope

3) Protected scop

4) Public scope

1) Private Scope-

-> It is also known as class level scope.

-> To provide any member of the class as the private scope we need to declare the member by using private modifier

-> Private scope visibility is upto the class definition, whatever we have declared the member of the class, even that cannot be accessed in sub class of the same package.

-> We cannot access the private member within the subclass of the same package and also it cannot be accessed in different package

-> Private scope can be provided only for member of class but not for class declaration.

2) Default Scope-

-> It is also known as package level scope.

-> To provide any member of the class as the Default scope, we need to declare the member without any modifiers.

-> Default scope member can be accessed anywhere (current class, sub-class, non sub-class) in same package

-> Default scope member cannot be accessed in different package

-> Default scope can be provided for member of the class as well as class declaration.

3) Protected Scope-

-> To provide any member of the class as protected scope, we need to declare the member by using protected modifiers

-> Protected scope member can be accessed anywhere (current class, sub-class, non sub-class) in same package.

-> Protected scope members can be accessed in different package for the sub class but not for non-sub class.

-> Protected scope can be provided for only members of the class but not for class declaration.

4) Public Scope-

-> It is also known as Global Scope.

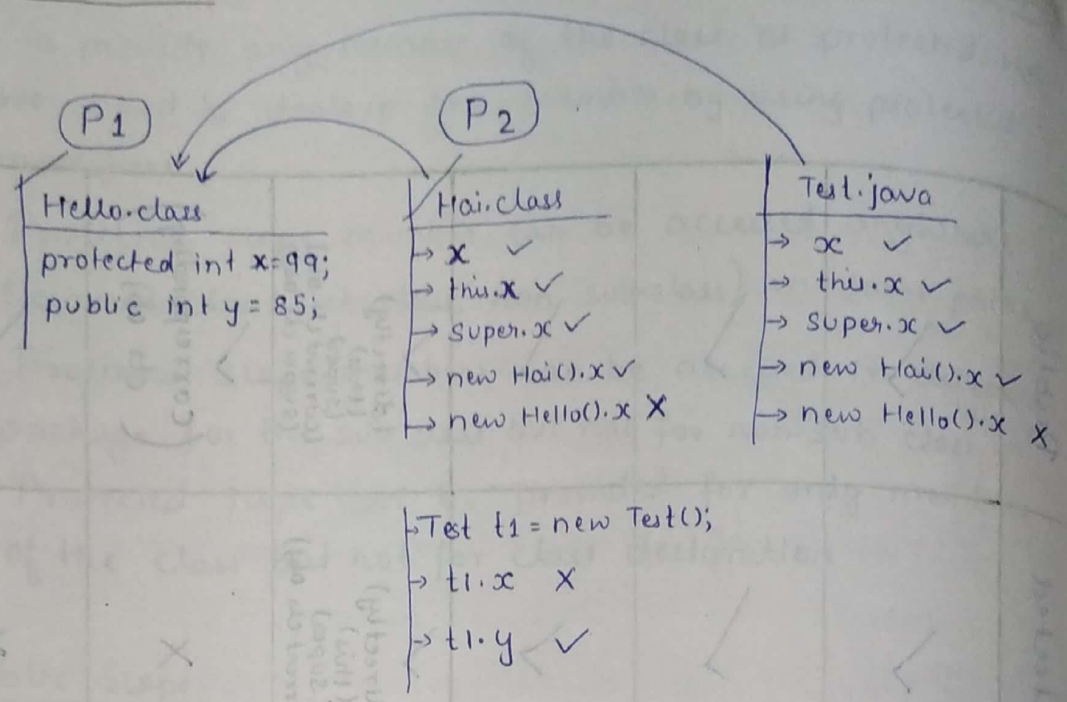
-> To provide any member of class as the public scope we need to declare the member by using public modifier.

-> Public scope member can be accessed any where (current class, sub class, non sub-class) in the same or different package.

* Access Modifier scope chart-

Package Scope	Class level Ways	Private	Default	Protected	Public
Same Package	Current class	✓ (direct) (this) (current cls obj)	✓	✓	✓
Same Package	Sub class	X	✓ (directly) (this) (super) (current cls obj) (super cls obj)	✓	✓
Same Package	Non Sub class	X	✓ (corresponding cls obj)	✓	✓
Different Pkg	Current Sub Class	X	X	✓ (directly) (this) (super) (current cls obj)	✓ (directly) (this) (super) (current cls obj) (super cls obj)
Different Pkg	Non-Sub class	X	X	X	✓ (Corresponding cls obj)
Different Pkg	Other Sub class	X	X	X	✓

* Other Sub Class-




```
package com.jlc.p2;
import com.jlc.p1.*;

class Test extends Hello {};
public class Hai extends Hello {
        void show(){
//      System.out.println(new Hello().x);
//      System.out.println(new Test().x);
        System.out.println(new Hello().y);
        System.out.println(new Test().y);
```

NOTE-

1) Protected scope member visibility is upto the current sub-class of different package and within the current sub class if you try to access with super class object then visibility is not there in the same way if you try to access the protected member with other sub-class object then visibility is not there but we can access the protected member only within the current sub class or within the other sub class definition.

2) But in the case of protected members, if public member is there then we can access with either super class object or other sub class object.

* Method Overriding- 03/11/2017

It is a process of defining the same method from super class to sub class with same signature but internal implementation should be different is known as Method Overriding Mechanism.

=> Method overriding mechanism is performed between two classes which was inheritance relation

=> Existed class cannot be done any modification. Newely defined class by inheriting the property of existed class is used for defining new member which are not there in existed class.

(i) Modifying the implementation of existed class member

(ii) Inheriting the property from existed class to newly defined class.

=> Rules while overriding any method from Super to Sub class-

1) Method signature should be same from super class to sub class in terms of

a) Method name as per case sensitivity should be same

b) Method parameter should be same

c) Method return type should be same.

2) In the case of Method return type, it should be same upto Jdk 1.4 by from jdk 1.5 onwards, if any super class method return type is reference type then while overriding in sub class we need to return same reference type as a return type or its sub type can be return which is known as Covariant type

3) If super class method is instance then we need to overide in sub class as instance method

4) If super class method is static then we need to override in sub class as static method.

5) If any super class method is declared as final method then that cannot be overridden in method.

6) If super class method is non final then we can override that method as the final method in sub class.

7) If any super class contains any access modifier scope then we need to follow some rules while overriding in sub class.

Super Class	Sub Class
private scope	private, default, protected or public scope In sub class, its not a overridden method since it will not be inherited to the ~~new~~ subclass so it is considered as new defined method in sub class
default scope	default, protected & public scope
protected scope	protected or public scope
public	public scope
native	non native
non native	native
synchronized	non-synchronized
non-synchronized	synchronized
strictfp	non strictfp
non strictfp	strictfp

* Dynamic Dispatch-

Process of assigning any sub class object to the super class object reference variable

```
A aref = new A();
aref = new B();

class A(){}
class B extends A(){}
```

* Polymorphism-

It is the process of one source behaviour differently at different scenario. To achive the principle logically we need to use three process.

1) Method Overloading
2) Method Overriding
3) Dynamic dispatch

=> Polymorphism is classified into two types-

a) Static Polymorphism (Early Binding)
b) Dynamic Polymorphism (Late Binding)

NOTE-

=> Static polymorphism can be achived with Methor overloading
Method overloading is done without Inheritance principle

=> Dynamic polymorphism can be achived with
→ Method overriding and
→ Dynamic dispatch

Method overriding is done with Inheritance principle only

a) Static Polymorphism-

→ It is also known as compile time polymorphism
→ We can achive static polymorphism with both instance and static method.
→ It is also known as Early Binding mechanism because java compiler will verify the method name first and then method parameter will be verified while calling the method. So, if matching method found then it will be binded for the method call or else compilation error will be thrown by the compiler.

Return type will not be considered.

NOTE-

Static polymorphism is known as early binding mechanism because before going to execution, 1st of all at the time of compilation itself it is matching/verifying the method or not with corresponding defination and parameter.

In case if method definition is not available then it will give compilation error only because compiler is rectifying that the method is not available and after that in case if method definition is available then it will search for method parameter and if parameter are matching or compatibility is not there then compiler is rectifying and throwing error. So, if any compilation problem occurs then it will not go for execution procedure since byte code file is not there so, it will be considered as early binding mechanism or method matching mechanism.

=> We can achive static polymorphism with both instance and static methods because both can be overloaded within the class definition and while calling both methods same verification will happen with compiler.

* Dynamic Polymorphism-

It is also called as Late Binding mechanism

Two methods are used in Dynamic Polymorphism.

1) Method Overriding

2) Dynamic dispatch.

ex-

```
class Test{
    public static void main (String arg[]){
        A aref = new B();    //-> dynamic dispatch
        aref.show();
    }
}

class A {
    void show(){
        System.out.println("A ==> show()");
    }
}

class B extends A {
    void show(){
        System.out.println("B ==> show()");
    }
}
```

inheritence relation and overridding mechanism.

NOTE-

1) In the case of dynamic dispatch if we assign subclass object into the super class reference variable then we can call the member only from super class and there is no chance to call the newly defined member from subclass with the super class reference variable.

2) Whenever we are calling any member with super class reference variable, that member signature must and should be available in super class. If it is not there then we cannot access it.

3) We can assign any sub class object into the super class reference variable directly but if you try to assign any super class object to sub class reference variable, it is not possible.

4) The process of assigning any sub class object into the super class reference variable is known as Dynamic dispatch or upcasting mechanism. but if you want to assign any super class object to sub class reference variable directly its not possible but we can assign by using down casting mechanism as follows.

```
class A{}
class B extends A{}
```

```
A aref= new B();        -> valid (dynamic dispatch/upcasting)
B bref= aref;           -> invalid
B bref = (B)aref;       -> valid (down casting)
```

5) In the case of dynamic polymorphism, when we assign any sub class object into the super class reference variable and we are invoking the methods either instance or static method. it will do following task.

(i) In the case of instance method always it will verifies based on the actual object syntax. So, according to different sub class objects it is calling from corresponding sub classes where method is overridden. So we can achive dynamic ~~dispatch~~ polymorphism with instance method even though if you are calling based on single reference variable which contains different sub class object and the implementation are also calling from different sub classes.

(ii) In the case of static method, always it will depends on the type of reference variable and it doesn't verify which actual object is present and here we are calling the static method with single reference variable which contains different sub class object for multiple times will be invoked from same class. So we cannot achive dynamic polymorphism in the case of static method.

6) In a class definition there is a mechanism of method overriding but not variable overriding. In any scenario if we have same variable declaration in super class and sub class with different content and if you try to access the variable with dynamic dispatch then it will always call from type of reference variable that is either from super class or sub class but not on the actual object syntax. So, this mechanism is known as "variable hiding".