

# List of Concepts Involved:

- Object Creation
- Instance variable v local variable
- Methods with Memory Map (JVM area)
- Method Overloading

## Topic 1: Object Creation

- We can use the "new" operator to create an object.
- There is no "delete" operator in java because destruction of useless objects is the responsibility of the garbage collector.

JAVA Virtual Machine (JVM) is used for creating an object. JVM requires a design (also c/a Blueprint) in order to create an object.

Blueprint is created through a concept called "Class".

```

object
class Test
{
    String name;
    int a;
    public static void main(String[] args) {
        Test t = new Test();
    }
}

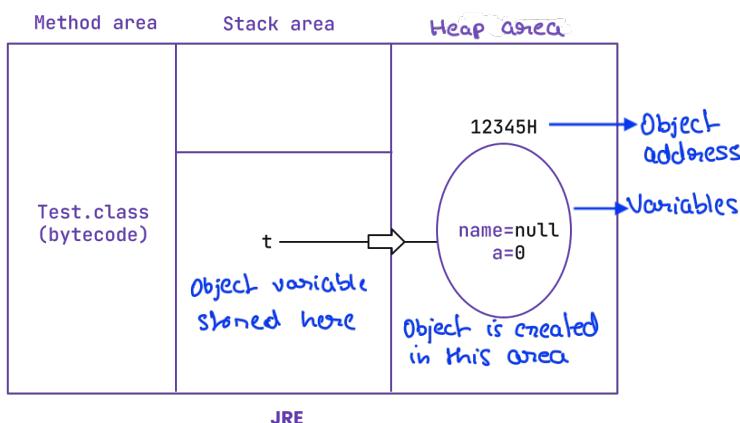
```

**Methods:** What object performs  
**Variables:** What object knows

Created Object requires a memory for storage and this created object must be allocated or deallocated. Ex:- In C++ this allocation & deallocation of object is done manually while in JAVA object allocation is done by "new" operator and deallocation is done automatically by the "Garbage Collector" which deallocates the object whenever it is not in use.

### Behind the scenes of Object Creation

- new operator is a keyword which is used to create the Object.
- When we say new operator , JVM would allocate memory on the heap area.
- JVM will load the supplied class name data into the method area.
- JVM will initialise the memory of instance variables.
- JVM will set the default value for instance variables based on the data type. Once the memory is set then the address of the object will be stored inside the reference variable.



#### Note:

`new` is an operator to create an object , if we know the class name at the beginning then we can create an object by using `new` operator.

So, when we run a code it goes to 2 steps  
 1<sup>st</sup> → compiled  
 2<sup>nd</sup> → Running code

When the code is compiled a `Test.class` file is created (which is imp for running a code). Now when we run the code the `Test.class` file (a bytecode) is loaded onto Method area, then the Object Variable (`t`) is stored inside the Stack area while the Object is created inside the Heap area an address is also allocated to the object.

Now, in order to use the object we have to link the Object variable (in stack) and Object (in heap) using :

`Test t = new Test();`

# Types of Variables

## Division 1:

Based on the type of value represented by a variable all variables are divided into 2 types. They are:

1. Primitive variables → Points to a primitive value (like no., character, point-value etc.)
2. Reference variables → Points to an Object

## Primitive variables:

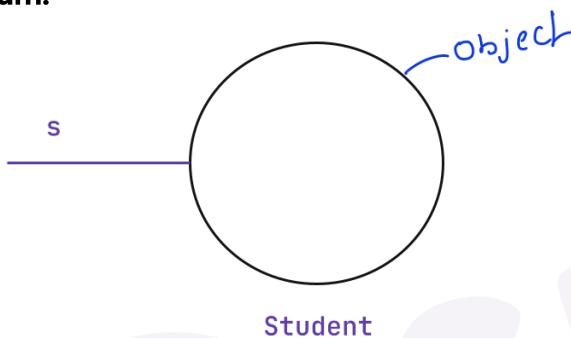
Primitive variables can be used to represent primitive values.

Example: int x=10;

Reference variables: Reference variables can be used to refer objects.

Example: Student s=new Student();

## Diagram:



## Division 2:

Based on the behavior and position of declaration all variables are divided into the following 3 types.

- ① Instance variables
- ② Static variables
- ③ Local variables

## Instance variables:

- If the value of a variable is varied from object to object such types of variables are called instance variables.
- For every object a separate copy of instance variables will be created.
- ★ Instance variables will be created at the time of object creation and destroyed at the time of object destruction hence the scope of instance variables is exactly the same as scope of objects.
- Instance variables will be stored on the heap as the part of the object.
- Instance variables should be declared within the class directly but outside of any method or block or constructor.
- Instance variables can be accessed directly from the Instance area. But cannot be accessed directly from a static area.
- But by using object reference we can access instance variables from static area

Q1 Write a code creating two objects Obj1 and Obj2, also explain How to access instance variable and where they are stored

Sol → Class Test

```
int a = 9; // Primitive Instance Variable  
String name = "Rahul"; // Primitive Instance Variable  
public static void main (String [] args)  
{  
    int num = 9; // Primitive Local Variable  
    Test Obj1 = new Test(); // Reference Local Variable  
    Test Obj2 = new Test(); // Reference Local Variable
```

Obj1.name = "Navin";

Sop (Obj1.a);

Sop (Obj1.name);

Sop (Obj2.a);

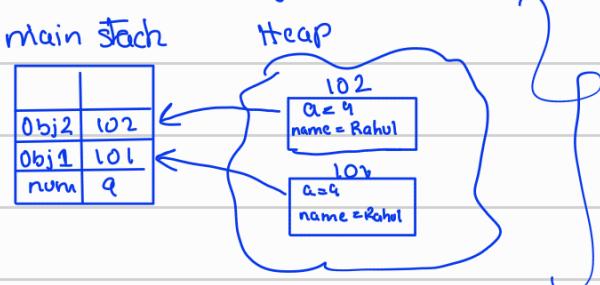
Sop (Obj2.name);

3

Output :-

2	navin
2	Rahul

Explanation:- So what happens is that stack memory stores all the local variables we stored inside the stack while the Instance variable are stored inside the heap along with the object like



As stack contains reference address of the object when

Obj1.name = Navin

line is executed only the value inside the obj2 is changed bcz

Every obj can have diff value for same instance variable.

### Example

```

class Test
{
    int i =10;
    public static void main(String[] args) {
        System.out.println(i); //CE: non static variable can't be referenced

        Test t = new Test();
        System.out.println(t.i); //valid
        t.m1();

    }
    public void m1()
    {
        System.out.println(i); //valid
    }
}

```

#When the execution is ended for the above object the the main stack will be deleted and the obj will be destroyed by Garbage Collector.

### Local variables:

- Sometimes to meet temporary requirements of the programmer we can declare variables inside a method or block or constructors such type of variables are called local variables or automatic variables or temporary variables or stack variables.
- Local variables will be stored inside the stack.
- The local variables will be created as part of the block execution in which it is declared and destroyed once that block execution completes. Hence the scope of the local variables is exactly the same as the scope of the block in which we declared.

→ Local variable must be initialised i.e int num=9;(V)  
in num;(X)

### Example

```

class Test
{
    public static void main(String[] args) {
        int i =0;
        for (int j =0;j<=3 ;j++ )
        {
            i = i+j;
        }
        System.out.println(j); //CE
    }
}

```

- The local variables will be stored on the stack.
- For the local variables JVM won't provide any default values, we should perform initialization explicitly before using that variable.

**Example**

```
class Test
{
    public static void main(String[] args) {
        int x;
        System.out.println("hello");//hello
    }
}
```

**Example**

```
class Test
{
    public static void main(String[] args) {
        int x;
        System.out.println(x);//CE
    }
}
```

**Example**

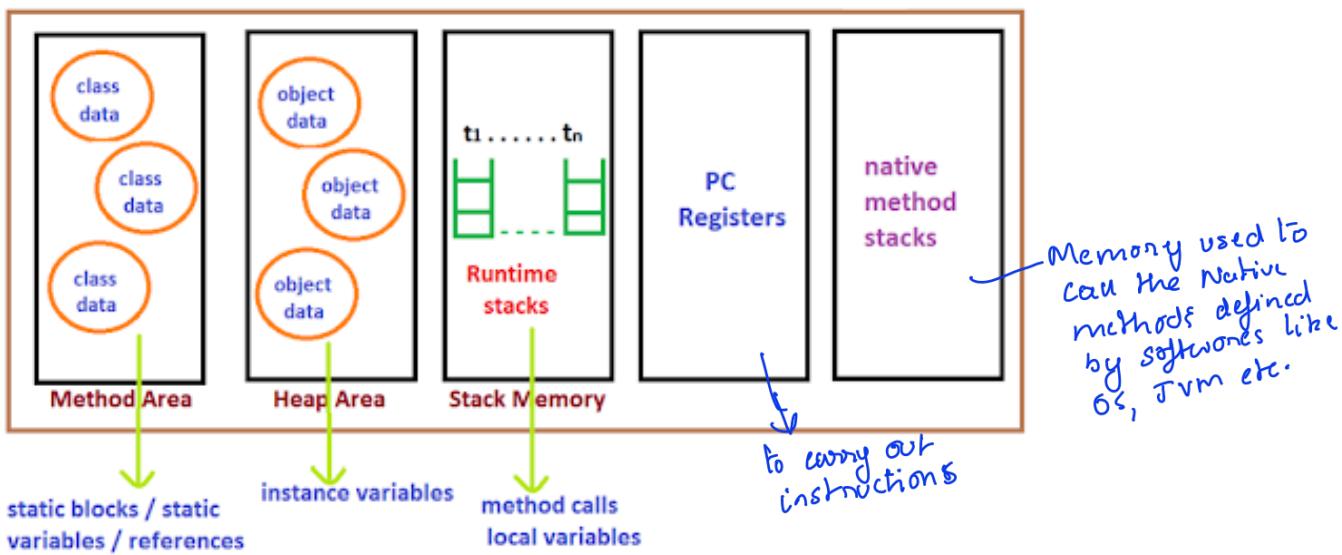
```
class Test
{
    public static void main(String[] args) {
        int x;
        if(args.length>0)
        {
            x=10;
        }
        System.out.println(x);//CE
    }
}
```

**Example**

```
class Test
{
    public static void main(String[] args) {
        int x;
        if(args.length>0)
        {
            x=10;
        }else{
            x = 20;
        }
        System.out.println(x);
    }
}
```

- It is never recommended to perform initialization for the local variables inside logical blocks because there is no guarantee of executing that block always at runtime.
- It is highly recommended to perform initialization for the local variables at the time of declaration at least with default values.
- The only applicable modifier for local variables is final. If we are using any other modifier we will get a compile time error.

## Various Memory areas present inside JVM :



- Class level binary data including static variables will be stored in method area.
- Objects and corresponding instance variables will be stored in the Heap area.
- For every method the JVM will create a Runtime stack, all method calls performed by that Thread and corresponding local variables will be stored in that stack. Every entry in stack is called Stack Frame or Action Record.
- The instruction which has to execute next will be stored in the corresponding PC Registers.
- Native method invocations will be stored in native method stacks.

## Method Overloading

- Two methods are said to be overloaded if and only if both have the same name but different argument types.
- In the 'C' language we can't take 2 methods with the same name and different types. If there is a change in argument type compulsory we should go for a new method name.

### Example :

`abs()` for int datatype  
`labs()` for long datatype  
`fabs()` for float datatype

- Lack of overloading in "C" increases complexity of the programming.
- But in java we can take multiple methods with the same name and different argument types.
  - `abs(int)` for int datatype
  - `abs(long)` for long datatype
  - `abs(float)` for float datatype
- Having the same name and different argument types is called method overloading. All these methods are considered as overloaded methods.
- Having overloading concept in java reduces complexity of the programming

### Example

```

class Test {
    public void m1(){
        System.out.println("zero arg method");
    }
    public void m1(int i){
        System.out.println("int arg method");
    }
    public void m1(double d){
        System.out.println("double arg method");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.m1();
        t.m1(10);
        t.m1(10.5);
    }
}

```

### Output

zero arg method

int arg method

double arg method

### Note:

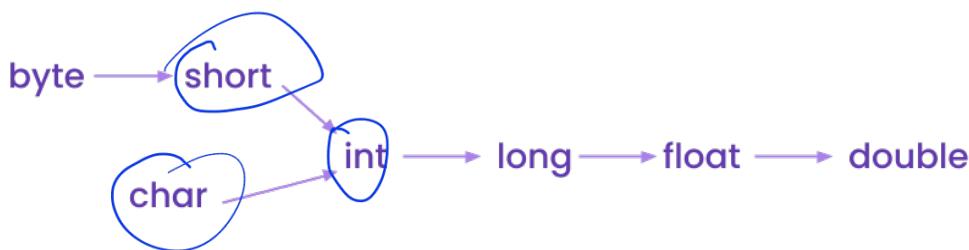
#### Conclusion :

In overloading, the compiler is responsible to perform method resolution(decision) based on the reference type(but not based on run time object). Hence overloading is also considered as compile time polymorphism(or) static polymorphism (or)early binding.

## Case 1: Automatic promotion in overloading.

- In overloading if the compiler is unable to find the method with exact match we won't get any compile time error immediately.
- First the compiler promotes the argument to the next level and checks whether the matched method is available or not if it is available then that method will be considered if it is not available then the compiler promotes the argument once again to the next level.
- This process will be continued until all possible promotions still if the matched method is not available then we will get a compile time error. This process is called automatic promotion in overloading.

The following are various possible automatic promotions in overloading.



**Example**

```

class Test {
    public void m1(int i){
        System.out.println("int arg method");
    }
    public void m1(float d){
        System.out.println("float arg method");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.m1('a');//int arg method
        t.m1(10l);//float arg method
        t.m1(10.5);//CE
    }
}

```

**Example**
**CASE-2**

```

class Test {
    public void m1(String s){
        System.out.println("String arg method");
    }
    public void m1(Object d){
        System.out.println("Object arg method");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.m1("sachin");//String arg method
        t.m1(new Object());//Object arg method
        t.m1(null);//String arg method
    }
}

```

**Note :**

While resolving overloaded methods, exact matches will always get high priority.

While resolving overloaded methods, the child class will get more priority than the parent class.

## Example

### CASE-3

```
class Test {
    public void m1(String s){
        System.out.println("String arg method");
    }
    public void m1(StringBuffer d){
        System.out.println("StringBuffer arg method");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.m1("sachin");//String arg method
        t.m1(new StringBuffer("dhoni"));//StringBuffer arg method
        t.m1(null);//CE
    }
}
```

## Example

### CASE-4

```
class Test {
    public void m1(int i, float f){
        System.out.println("int, float arg method");
    }
    public void m1(float f, int i){
        System.out.println("float,int arg method");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.m1(10,10.5f);//int,float arg method
        t.m1(10.5f,10);//float,int arg method
        t.m1(10,10);//CE
        t.m1(10.5f,10.5f);//CE
    }
}
```