UNIT-8

CLASS, OBJECT, METHOD PART-2

8.1 Method overloading
8.2 Actual and Formal Arguments
8.3 Passing Arrays to Method, Array of Object
8.4 Call by Value and Call by Reference,
8.5 Returning object, Object as Parameter

8.1 Method overloading: -

- Method Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters, or a mixture of both.
- Method overloading is also known as Compile-time Polymorphism, Static Polymorphism, or Early binding in Java. In Method overloading compared to parent argument, child argument will get the highest priority.

EXAMPLE: -

```
// Java program to demonstrate working of method
// overloading in Java
public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y) { return (x + y); }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
            return (x + y + z);
    }
```

```
    // Overloaded sum(). This sum takes two double
    // parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Output :

```
30
60
31.0
```

Different Ways of Method Overloading in Java

1. Changing the Number of Parameters.
2. Changing Data Types of the Arguments.
3. Changing the Order of the Parameters of Methods

1. Changing the Number of Parameters: -

Method overloading can be achieved by changing the number of parameters while passing to different methods.

```
// Java Program to Illustrate Method Overloading
// By Changing the Number of Parameters
// Importing required classes
import java.io.*;
// Class 1
// Helper class
class Product {

// Method 1
// Multiplying two integer values
public int multiply(int a, int b)
{
int prod = a * b;
return prod;
}

// Method 2
// Multiplying three integer values
public int multiply(int a, int b, int c)
{
int prod = a * b * c;
return prod;
}
}

// Class 2
// Main class
class GFG {
// Main driver method
public static void main(String[] args)
{

// Creating object of above class inside main()
```

```
// method
Product ob = new Product();

// Calling method to Multiply 2 numbers
int prod1 = ob.multiply(1, 2);

// Printing Product of 2 numbers
System.out.println(
"Product of the two integer value :" + prod1);

// Calling method to multiply 3 numbers
int prod2 = ob.multiply(1, 2, 3);

// Printing product of 3 numbers
System.out.println(
"Product of the three integer value :" + prod2);
}
}
```

Output:

```
Product of the two integer value :2
Product of the three integer value :6
```

2. Changing Data Types of the Arguments

In many cases, methods can be considered Overloaded if they have the same name but have different parameter types, methods are considered to be overloaded.

```
// Java Program to Illustrate Method Overloading
// By Changing Data Types of the Parameters

// Importing required classes
import java.io.*;

// Class 1
// Helper class
```

```
class Product {

	// Multiplying three integer values
	public int Prod(int a, int b, int c)
	{

		int prod1 = a * b * c;
		return prod1;
	}

	// Multiplying three double values.
	public double Prod(double a, double b, double c)
	{

		double prod2 = a * b * c;
		return prod2;
	}
}

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

		Product obj = new Product();

		int prod1 = obj.Prod(1, 2, 3);
		System.out.println("Product of the three integer value :" + prod1);
		double prod2 = obj.Prod(1.0, 2.0, 3.0);
		System.out.println("Product of the three double value :" + prod2);
	}
}
```

Output:

```
Product of the three integer value :6
Product of the three double value :6.0
```

3. Changing the Order of the Parameters of Methods

Method overloading can also be implemented by rearranging the parameters of two or more overloaded methods. For example, if the parameters of method 1 are (String name, int roll_no) and the other method is (int roll_no, String name) but both have the same name, then these 2 methods are considered to be overloaded with different sequences of parameters.

```
// Java Program to Illustrate Method Overloading
// By changing the Order of the Parameters
// Importing required classes
import java.io.*;
// Class 1
// Helper class
class Student {

	// Method 1
	public void StudentId(String name, int roll_no)
	{
		System.out.println("Name :" + name + " " + "Roll-No :" + roll_no);
	}
	// Method 2
	public void StudentId(int roll_no, String name)
	{
		// Again printing name and id of person
		System.out.println("Roll-No :" + roll_no + " " + "Name :" + name);
	}
}
// Class 2
// Main class
class GFG {
	public static void main(String[] args)
	{

		// Creating object of above class
		Student obj = new Student();

		// Passing name and id
```

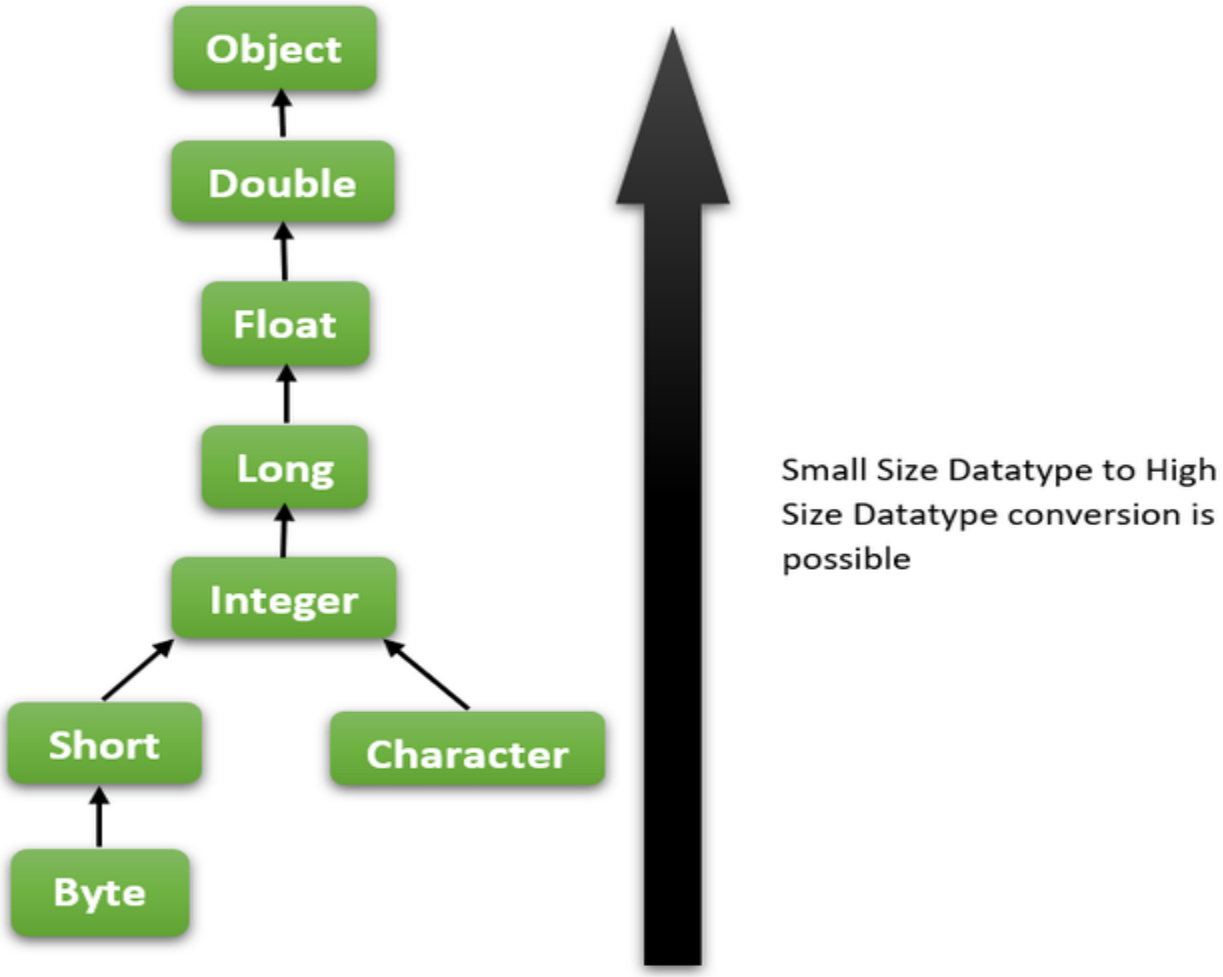
```
                // Note: Reversing order
                obj.StudentId("Spyd3r", 1);
                obj.StudentId(2, "Kamlesh");
        }
}
```

Output:

Name :Spyd3r Roll-No :1
Roll-No :2 Name :Kamlesh

CASE 1:-What happen if the exact prototype does not match with arguments?

Priority-wise, the compiler takes these steps:
Type Conversion but to higher type(in terms of range) in the same family.
Type conversion to the next higher family(suppose if there is no long data type available for an int data type, then it will search for the float data type).



```
class Demo {
	public void show(int x)
	{
		System.out.println("In int" + x);
	}
	public void show(String s)
	{
		System.out.println("In String" + s);
	}
	public void show(byte b)
	{
		System.out.println("In byte" + b);
	}
}
class UseDemo {
	public static void main(String[] args)
	{
		byte a = 25;
		Demo obj = new Demo();

		// it will go to
		// byte argument
		obj.show(a);

		// String
		obj.show("hello");

		// Int
		obj.show(250);

		// Since char is
		// not available, so the datatype
		// higher than char in terms of
		// range is int.
		obj.show('A');

		// String
```

```
                obj.show("A");

                // since float datatype
                // is not available and so it's higher
                // datatype, so at this step their
                // will be an error.
                obj.show(7.5);
        }
}
```

CASE 2-What happen when argument is ambiguous for different class?

```
import java.util.*;
class Test
{
        //      method-1
        public void name(String s)
        {
                System.out.println("String class ");
        }
        //      method-2
        public void name(StringBuffer s)
        {

                System.out.println("StringBuffer class ");
        }
}
class Run
{
   public static void main (String args[])
        {
                Test t=new Test();
                t.name("LJ");//method-1 calling
                t.name(new StringBuffer("ABC"));//method-2 calling
                t.name(null);//Compile error
        }
}
```

Output:-

```
D:\>javac ob.java
ov.java:25: error: reference to name is ambiguous
                t.name(null);
                 ^
  both method name(String) in Test and method name(StringBuffer) in Test match
1 error
```

CASE 3-What happen when argument is ambiguous for different Parent child class?

Example:-

```
class Argument {
//method-1
public void m1(Object o) {
     System.out.println("Object version");
   }
//method-2
public void m1(String s) {
     System.out.println("String version");
   }
public static void main(String args[]) {
   Argument A = new Argument();
   A.m1(new Object());//method-1 calling
        A.m1("ABC");//method-2 calling
        A.m1(null);//method-2 calling
  }
}
```

Output: -

```
Object version
String version
String version
```

CASE 4-What happen when we have two types of arguments general method or variable length arguments?

Example:-

```
import java.util.*;
class Test
```

```
{
	//	method-1
	public void add(int i)
	{
		System.out.println("General method");
	}
	//	method-2
	public void add(int... i)
	{

		System.out.println("Variable length argments method");
	}
}
class Run
{
   public static void main (String args[])
	{
		Test t=new Test();
		t.add(2);//method-1 calling
		t.add();//method-2 calling
		t.add(2,2);//method-2 calling
		t.add(2,2,2);//method-2 calling
	}
}
```

Output: -

```
General method
Variable length argments method
Variable length argments method
Variable length argments method
```

Example:-

```
import java.util.*;
class Test
{
	//	method-1
	public void add(int i,float f)
	{
```

```
                System.out.println("int-float version");
        }
        //      method-2
        public void add(float f,int i)
        {

                System.out.println("float-int version");
        }
}
class Run
{
   public static void main (String args[])
        {
                Test t=new Test();
                t.add(10,10.5f);//method-1 calling
                t.add(10.5f,10);//method-2 calling
                t.add(10,10);//Compile error
                t.add(10.5f,10.5f);//compile error

        }
}
error: reference to add is ambiguous
                t.add(10,10);//Compile error
                ^
  both method add(int,float) in Test and method add(float,int) in Test match
ov.java:25: error: no suitable method found for add(float,float)
                t.add(10.5f,10.5f);//compile error
                ^
   method Test.add(int,float) is not applicable
     (argument mismatch; possible lossy conversion from float to int)
   method Test.add(float,int) is not applicable
     (argument mismatch; possible lossy conversion from float to int)
2 errors
```

CASE 5- It is possible to overload main method?

```
class Overloading_Case5{

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

```
System.out.println("Main method String[] args");
main(10);
}
public static void main(int a){
        System.out.println("Main method int a");
}
}
```

OUTPUT:-

Main method String[] args

Main method int a

Advantages of Method Overloading

1. Method overloading improves the Readability and reusability of the program.
2. Method overloading reduces the complexity of the program.
3. Using method overloading, programmers can perform a task efficiently and effectively.
4. Using method overloading, it is possible to access methods performing related functions with slightly different arguments and types.

8.2 Actual and Formal Arguments: -

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the "caller function" and B is called the "called function or callee function". Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

Types of parameters:

1. **Formal Parameter:** A variable and its type as they appear in the prototype of the function or method.

Syntax:

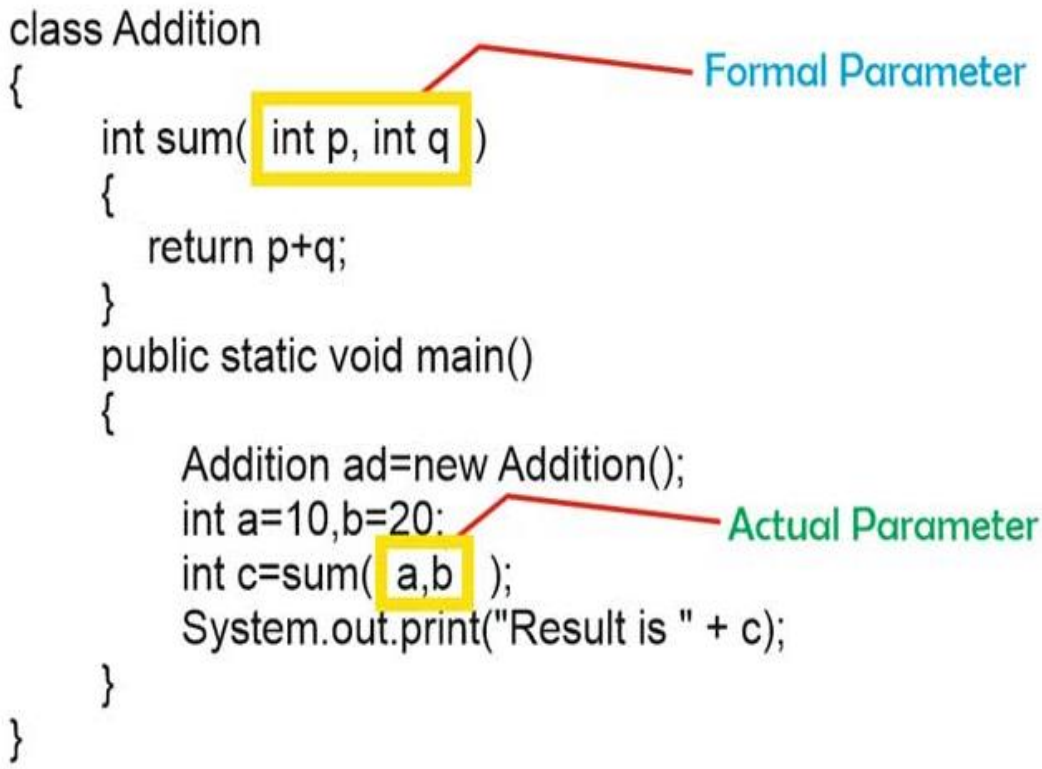
function_name(datatype variable_name)

2. **Actual Parameter:** The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

Syntax:

func_name(variable name(s));

Example: -



Actual parameters are situated in caller method and formal parameters are written in called function. That is if we write a method called "sum()" with two parameter called "p" and "q". These parameters are known as formal parameters. Now we call the "sum()" function from another function and passes two parameters called "a" and "b". then these parameters known as actual parameters.

8.3 Passing Arrays to Method: -

You can pass arrays to a method just like normal variables. When we pass an array to a method as an argument, actually the address of the array in the memory is passed (reference). Therefore, any changes to this array in the method will affect the array.

Passing an array to a function is an easy-to-understand task in java. Let function GFG() be called from another function GFGNews(). Here, GFGNews is called the "Caller Function" and GFG is called the "Called Function OR Callee Function". The arguments/parameters which GFGNews passes to GFG are called "Actual Parameters" and the parameters in GFG are called "Formal Parameters". The array to pass can be a one-dimensional(1D) array or a multi-dimensional array such as a 2D or 3D array. The Syntax to Pass an Array as a Parameter is as follows:

Caller Function:

```
called_function_name(array_name);
```

The code for the called function depends on the dimensions of the array.

The number of square brackets in the function prototype is equal to the dimensions of the array, i.e. [n] for 1D arrays, [n][n] for 2D arrays, [n][n][n] for 3D arrays, and so on.

Called Function:

```
// for 1D array
returnType functionName(datatype[] arrayName) {
    //statements
}
```

OR

```
// for 1D array
returnType functionName(datatype arrayName[]) {
```

```
    //statements
}
```

Similarly, for 2D Arrays, the Syntax would be:

```
// for 2D array
returnType functionName(datatype[][] arrayName) {
    //statements
}
OR
// for 2D array
returnType functionName(datatype arrayName[][]) {
    //statements
}
```

Example:-

```
import java.io.*;

class GFG {
        void function1(int[] array) {
                System.out.println("The first element is: " + array[0]);
        }
        void function2(int[][] array) {
                System.out.println("The first element is: " + array[0][0]);
        }

        public static void main(String[] args) {

                // creating instance of class
                GFG obj = new GFG();

                // creating a 1D and a 2D array
                int[] oneDimensionalArray = { 1, 2, 3, 4, 5 };
```

```
            int[][] twoDimensionalArray = { { 10, 20, 30 },{ 40, 50, 60 },{ 70, 80,
90 } };

            // passing the 1D array to function 1
            obj.function1(oneDimensionalArray);

            // passing the 2D array to function 2
            obj.function2(twoDimensionalArray);
        }
}
```

Output: -

```
The first element is: 1
The first element is: 10
```

Example: -

```
import java.util.Scanner;
class array {
  public int max(int [] array) {
    int max = 0;

    for(int i=0; i<array.length; i++ ) {
      if(array[i]>max) {
        max = array[i];
      }
    }
    return max;
  }

  public int min(int [] array) {
    int min = array[0];

    for(int i = 0; i<array.length; i++ ) {
      if(array[i]<min) {
        min = array[i];
      }
    }
```

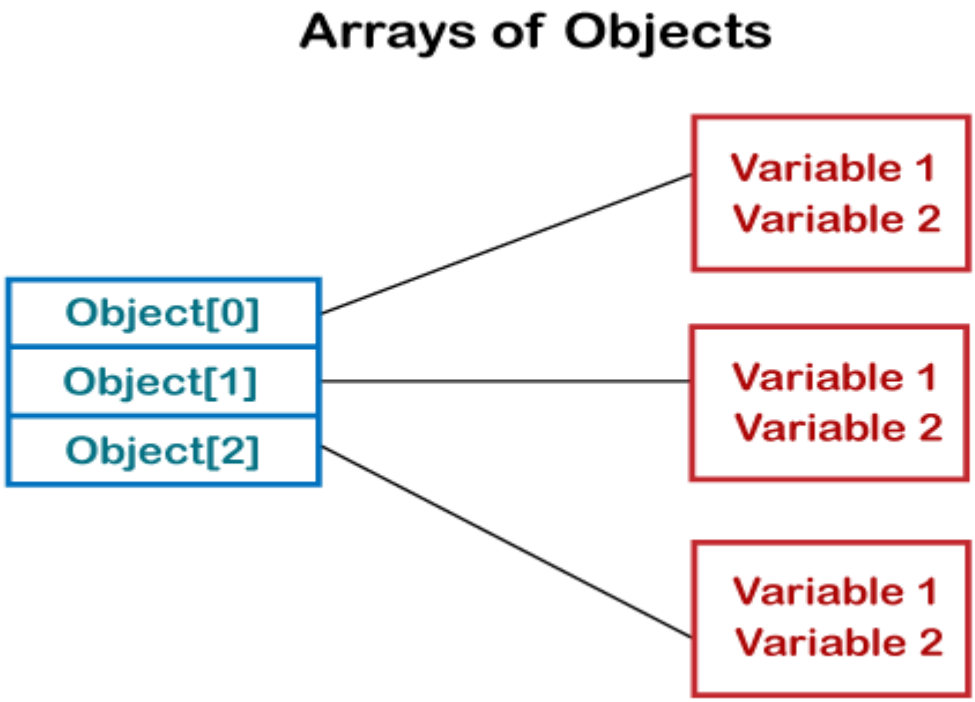
```
    return min;
  }
}
class Run{
  public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the array range");
    int size = sc.nextInt();
    int[] arr = new int[size];
    System.out.println("Enter the elements of the array ::");

    for(int i=0; i<size; i++) {
      arr[i] = sc.nextInt();
    }
    array m = new array();
    System.out.println("Maximum value in the array is::"+m.max(arr));
    System.out.println("Minimum value in the array is::"+m.min(arr));
  }
}
```

Output: -

```
Enter the array range
5
Enter the elements of the array ::
1
2
3
4
5
Maximum value in the array is::5
Minimum value in the array is::1
```

Array of Object: -

- We can create an array of object in java.
- Similar to primitive data type array we can also create and use arrays of derived data types
- We know that an array is a collection of the same data type that dynamically creates objects and can have elements of primitive types. Java allows us to store objects in an array.
- In Java, the class is also a user-defined data type. An array that conations class type elements are known as an array of objects. It stores the reference variable of the object.



Creating an Array of Objects

Before creating an array of objects, we must create an instance of the class by using the new keyword. We can use any of the following statements to create an array of objects.

Syntax:

ClassName obj[]=new ClassName[array_length];

//declare and instantiate an array of objects

Or

ClassName[] objArray;
Or

ClassName objeArray[];

Suppose, we have created a class named Employee. We want to keep records of 20 employees of a company having three departments. In this case, we will not create 20 separate variables. Instead of this, we will create an array of objects, as follows.

Employee department1[20];
Employee department2[20];
Employee department3[20];
Example: -

```
// Java program to demonstrate initializing
// an array of objects using a method

class GFG {

    public static void main(String args[])
    {

        // Declaring an array of student
        Student[] arr;

        // Allocating memory for 2 objects
        // of type student
        arr = new Student[2];

        // Creating actual student objects
        arr[0] = new Student();
        arr[1] = new Student();

        // Assigning data to student objects
```

```
                arr[0].setData(1701289270, "Satyabrata");
                arr[1].setData(1701289219, "Omm Prasad");
                // Displaying the student data
                System.out.println(
                        "Student data in student arr 0: ");
                arr[0].display();

                System.out.println(
                        "Student data in student arr 1: ");
                arr[1].display();
        }
}

// Creating a Student class with
// id and name as a attributes
class Student {

        public int id;
        public String name;

        // Method to set the data to
        // student objects
        public void setData(int id_, String name_)
        {
                id = id_;
                name = name_;
        }

        // display() method to display
        // the student data
        public void display()
        {
                System.out.println("Student id is: " + id + " "
                                        + "and Student name is: "
                                        + name);
                System.out.println();
        }
```

```
}
```

Output

```
Student data in student arr 0:
Student id is: 1701289270 and Student name is: Satyabrata

Student data in student arr 1:
Student id is: 1701289219 and Student name is: Omm Prasad
```

Example: -

```
import java.util.Scanner;
class Point
{
	int x,y;
	void setData()
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter x:");
		x = sc.nextInt();
		System.out.println("Enter y:");
		y = sc.nextInt();

	}
	void printData()
	{
		System.out.println("x = "+x);
		System.out.println("y = "+y);
	}
}
class TestProg
{
	public static void main(String args[])
	{

		Point p[]=new Point[3];
		for(int i=0;i<3;i++)
		{
			p[i]=new Point();
```

```
                p[i].setData();
            }
            for(int i=0;i<3;i++)
            {

                p[i].printData();
            }

        }
}
```

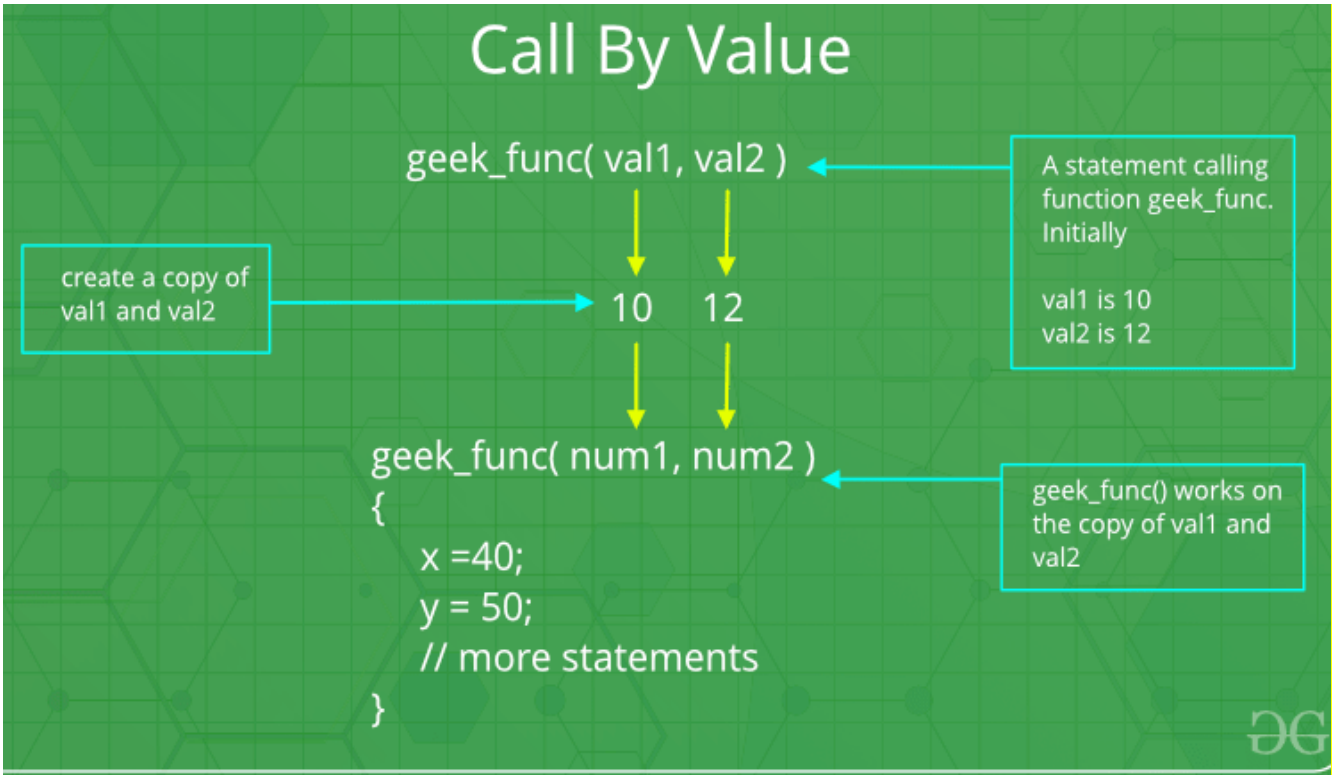
Output: -

Enter x:
3
Enter y:
4
Enter x:
1
Enter y:
2
Enter x:
7
Enter y:
8
x = 3
y = 4
x = 1
y = 2
x = 7
y = 8

8.4 Call by Value and Call by Reference: -

1. **Pass By Value:** Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as call by value.

Java in fact is strictly call by value.



Example: -

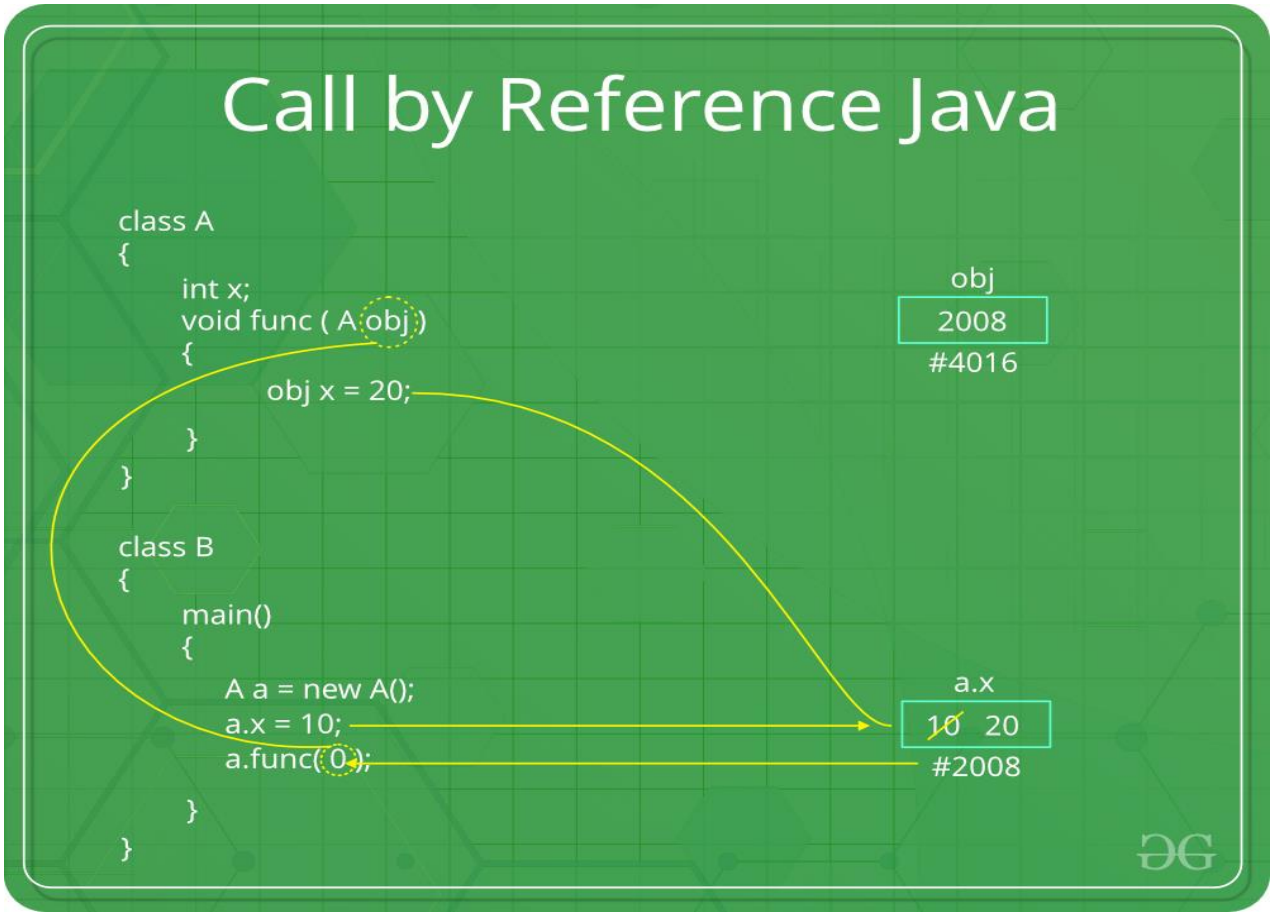
```
// Java program to illustrate
// Call by Value

// Callee
class CallByValue {

        // Function to change the value
        // of the parameters
```

```
	public void example(int x, int y)
	{
		x++;
		y++;
	}
}


// Caller
public class Main {
	public static void main(String[] args)
	{

		int a = 10;
		int b = 20;

		// Instance of class is created
		CallByValue object = new CallByValue();

		System.out.println("Value of a: " + a+ " & b: " + b);

		// Passing variables in the class function
		object.example(a, b);

		// Displaying values after
		// calling the function
		System.out.println("Value of a: "+ a + " & b: " + b);
	}
}
```

Output

```
Value of a: 10 & b: 20
Value of a: 10 & b: 20
```

2. **Call by reference(aliasing):** Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as call by reference. This method is efficient in both time and space.



Example: -

```
// Java program to illustrate
// Call by Reference

// Callee
class CallByReference {
    int a=10, b=20;

    // Changing the values of class variables
    void ChangeValue(CallByReference obj)
```

```
    {
        obj.a += 10;
        obj.b += 20;
    }
}

// Caller
class Main {

    public static void main(String[] args)
    {

        // Instance of class is created
        // and value is assigned using constructor
        CallByReference object = new CallByReference();

        System.out.println("Value of a: " + object.a+ " & b: " + object.b);

        // Changing values in class function
        object.ChangeValue(object);

        // Displaying values
        // after calling the function
        System.out.println("Value of a: " + object.a+ " & b: " + object.b);
    }
}
```

Output: -

Value of a: 10 & b: 20
Value of a: 20 & b: 40

8.5 Returning object, Object as Parameter: -

Although Java is strictly passed by value, the precise effect differs between whether a primitive type or a reference type is passed. When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference.
Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
This effectively means that objects act as if they are passed to methods by use of call-by-reference.
Changes to the object inside the method do reflect the object used as an argument.
In Java, a method can return any type of data, including class types that you can create.

Example: -

```
class Test
{
   int a;

   void set(int i)
   {
      a = i;
   }

   Test incrByTen(Test obj)
   {
      Test temp = new Test();
               temp.a = obj.a + 10;
      return temp;

   }
```

```
}

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

      Test obj1 = new Test();
      obj1.set(2);
      Test obj2;

      obj2 = obj1.incrByTen(obj1);

      System.out.println("obj1.a : " + obj1.a);
      System.out.println("obj2.a : " + obj2.a);

      obj2 = obj2.incrByTen(obj2);

      System.out.println("obj2.a after second increase : " + obj2.a);

   }
}
```

Output: -

```
obj1.a : 2
obj2.a : 12
obj2.a after second increase : 22
```

Example: -

```
import java.util.*;

import java.lang.String;


//Object as an argument

class Point

 {

        int x,y;
```

```
void set(int a,int b)
{
        x=a;
        y=b; }
void get()
{
        System.out.println(x);
        System.out.println(y);
}
void copy(Point p1)//object passed as an argument
{
        x=p1.x;
        y=p1.y;
}
}
class Main
{
        public static void main(String args[])
        {
                Point p1=new Point();
                p1.set(10,20);
                p1.get();
                Point p2=new Point();
                p2.copy(p1);
                p2.get();
```

```
        }
 }
```

Example:-

```
// object as an argument average of marks of two test
 class Test
 {
 double c_marks,java_marks;
 String Name,test;
 void set(String n,String t,double x,double y)
 {
 Name=n;
 test=t;
 c_marks=x;
 java_marks=y;
 }
 void get()
 {
 System.out.println("Name="+Name);
 System.out.println("Test="+test);
 System.out.println("c_marks= "+c_marks);
 System.out.println("java_marks ="+java_marks);
 System.out.println();
 }
 void average(Test t1, Test t2)
 {
```

```
c_marks=t1.c_marks+t2.c_marks;
java_marks=t1.java_marks+t2.java_marks;
double Total_Marks=(c_marks+java_marks);
double Avg_Marks=(c_marks+java_marks)/2;
System.out.println("Total_Marks of T1 and T2= "+Total_Marks);
System.out.println("Avg_Marks of T1 and T2= "+Avg_Marks);
}
}

class Main1
{
public static void main(String args[])
{
Test t1=new Test();
Test t2=new Test();
t1.set("Milan","T1",21.0,19.0);
t2.set("Milan","T2",23.0,21.0);
t1.get();
t2.get();
Test t3=new Test();
t3.average(t1,t2);
}
}
```