

## 2.1: Overloading Methods

**Method Overloading** is a feature in Java that allows a class to have **more than one methods having same name**, but with **different signatures** (Each method must have different number of parameters or parameters having different types and orders).

### **Advantage:**

- ✓ Method Overloading increases the readability of the program.
- ✓ Provides the flexibility to use similar method with different parameters.

### **Three ways to overload a method**

In order to overload a method, the argument lists of the methods must differ in either of these:

#### **1. Number of parameters. (Different number of parameters in argument list)**

For example: This is a valid case of overloading

```
add(int, int)
add(int, int, int)
```

#### **2. Data type of parameters. (Difference in data type of parameters)**

For example:

```
add(int, int)
add(int, float)
```

#### **3. Sequence of Data type of parameters.**

For example:

```
add(int, float)
add(float, int)
```

### **Rules for Method Overloading:**

1. First and important rule to overload a method in java is to change method signature.
2. Return type of method is never part of method signature, so only changing the return type of method does not amount to method overloading.

### **Example: To find the Minimum of given numbers:**

```
public class OverloadingCalculation1
{
    public static void main(String[] args)
    {
        int a = 11;
        int b = 6;
        int c = 3;
        double x = 7.3;
        double y = 9.4;
```

```

int result1 = minFunction(a, b, c);
double result2 = minFunction(x, y);
double result3 = minFunction(a, x);
System.out.println("Minimum("+a+", "+b+", "+c+") = " + result1);
System.out.println("Minimum("+x+", "+y+") = " + result2);
System.out.println("Minimum("+a+", "+x+") = " + result3);
}

public static int minFunction(int n1, int n2, int n3)
{
    int min;
    int temp = n1 < n2 ? n1 : n2;
    min = n3 < temp ? n3 : temp;
    return min;
}

public static double minFunction(double n1, double n2)
{
    double min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}

public static double minFunction(int n1, double n2)
{
    double min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;

    return min;
}
}

```

**This would produce the following result:**

Minimum(11,6,3) = 3

Minimum(7.3,9.4) = 7.3

Minimum(11,7.3) = 7.3

**Note:-**

Method overloading is not possible by changing the return type of the method because of ambiguity that may arise while calling the method with same parameter list with different return type.

**Example:**

```

class Add
{
    static int sum(int a, int b)
    {
        return a+b;
    }
    static float sum(int a, int b)
    {
        return a+b;
    }
    public static void main(String arg[])
    {
        System.out.println(sum(10,20));
        System.out.println(sum(15,25));
    }
}

```

**Output:**

Compile by: javac TestOverloading3.java

```

Add.java:7: error: method sum(int,int) is already defined in class Add
    static float sum(int a, int b)
           ^
1 error

```

**Method Overloading and Type Promotion**

**Type Promotion:** When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

**Type Promotion in Method Overloading:**

One type is promoted to another implicitly if no matching data type is found.

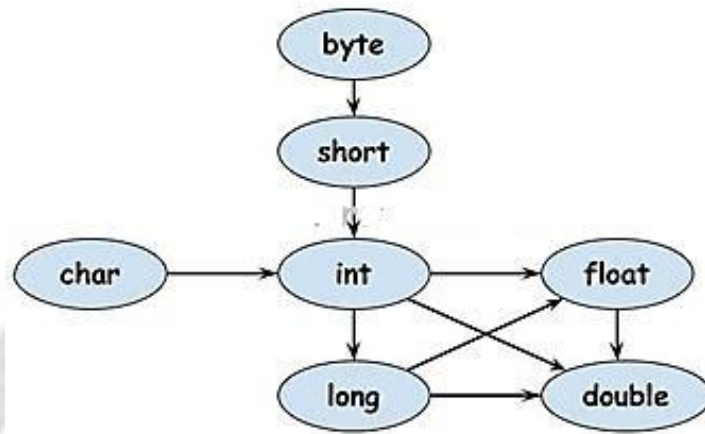
**Type Promotion Table:**

The data type on the left side can be promoted to the any of the data type present in the right side of it.

```

byte → short → int → long → double
short → int → long → float → double
int → long → float → double
float → double
long → float → double
char → int → long → float → double

```



### Example: Method Overloading with Type Promotion:

```

class Overloading
{
    void sum(int a, float b)
    {
        System.out.println(a+b);
    }
    void sum(int a, int b, int c)
    {
        System.out.println(a+b+c);
    }
    public static void main(String args[])
    {
        OverloadingCalculation1 obj=new OverloadingCalculation1();
        obj.sum(20,20); //now second int literal will be promoted to float
        obj.sum(100,'A'); //Character literal will be promoted to float
        obj.sum(20,20,20);
    }
}
  
```

### OUTPUT:

```

40.0
165.0
60
  
```

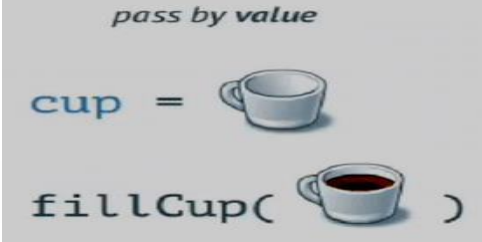
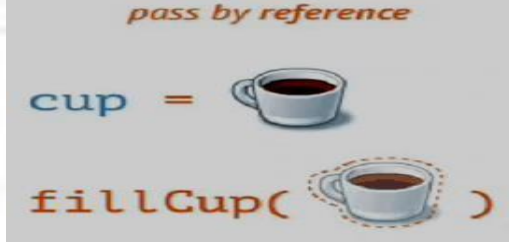
## 2.2: Objects as Parameters

Java is strictly pass-by-value. But the scenario may change when the parameter passed is of primitive type or reference type.

- If we pass a primitive type to a method, then it is called **pass-by-value** or call-by-value.
- If we pass an object to a method, then it is called **pass-by-reference** or call-by-reference.

**Object as a parameter** is a way to establish communication between two or more objects of the same class or different class as well.

**Pass-by-value vs. Pass-by-reference:**

<b>Pass-by-value (Value as parameter)</b>	<b>Pass-by-reference (Object as parameter)</b>
Only values are passed to the function parameters. So any modifications done in the formal parameter will not affect the value of actual parameter	Reference to the object is passed. So any modifications done through the object will affect the actual object.
Caller and Callee method will have two independent variables with same value.	Caller and Callee methods use the same reference for the object.
Callee method will not have any access to the actual parameter	Callee method will have the direct reference to the actual object
Requires more memory	Requires less memory
	
<pre>class CallByVal {     void Increment(int count)     {         count=count+10;     } } public class CallByValueDemo {     public static void main(String arg[])     {         CallByVal ob1=new CallByVal();         int count=100;         System.out.println("Value of Count before             method call = "+count);     } }</pre>	<pre>class CallByRef {     int count=0;     CallByRef(int c)     {         count=c;     }     static void Increment(CallByRef obj) {         obj.count=obj.count+10;     }     public static void main(String arg[]) {         CallByRef ob1=new CallByRef(10);         System.out.println("Value of Count (Object 1) before             method call = "+ob1.count);         Increment(ob1);         System.out.println("Value of Count (Object 1) after             method call = "+ob1.count);     } }</pre>
<pre>ob1.Increment(count); System.out.println("Value of Count after     method call = "+count); } }</pre> <p><b>OUTPUT:</b></p> <p>Value of Count before method call = 100 Value of Count after method call = 100</p>	<pre>method call = "+ob1.count); Increment(ob1); System.out.println("Value of Count (Object 1) after     method call = "+ob1.count); } }</pre> <p><b>OUTPUT:</b></p> <p>Value of Count (Object 1) before method call = 10 Value of Count (Object 1) after method call = 20</p>

**Returning Objects:**

In Java, a method can return any type of data. Return type may any primitive data type or class type (i.e. object). As a method takes objects as parameters, it can also return objects as return value.

**Example:**

```
class Add
{
    int num1,num2,sum;
    static Add calculateSum(Add a1,Add a2)
```

{

```
Add a3=new Add();  
a3.num1=a1.num1+a1.num2;  
a3.num2=a2.num1+a2.num2;  
a3.sum=a3.num1+a3.num2;  
return a3;
```

}

```
public static void main(String arg[])
```

{

```
Add ob1=new Add();  
ob1.num1=10;  
ob1.num2=15;  
Add ob2=new Add();  
ob2.num1=100;  
ob2.num2=150;  
Add ob3=calculateSum(ob1,ob2);  
System.out.println("Object 1 -> Sum = "+ob1.sum);  
System.out.println("Object 2 -> Sum = "+ob2.sum);  
System.out.println("Object 3 -> Sum = "+ob3.sum);
```

}

}

**OUTPUT:**

Object 1 -> Sum = 0

Object 2 -> Sum = 0

Object 3 -> Sum = 275