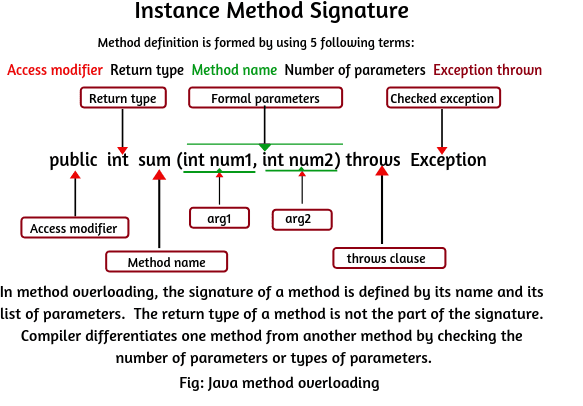
**Method Overloading in Java**

When a class has more than one method having the same name but with different parameter lists, this feature is called **method overloading in Java**.

Java compiler differentiates overloaded methods by their signatures. The signature of a method is defined by its name and list of parameters.

The return type of a method is not part of the method signature. It does not play any role in resolving methods overloaded.



## Method overloading rules in Java

The method name must be the same.

2. Parameters must be different i.e each overloaded method must take a unique list of parameter types. The parameters can be changed in one of the following three ways.

### a.Data types of Parameters

### b. Number of Parameters

### c. Sequence of Data type of Parameters

## Features of Method overloading

There are the following features of method overloading in java which you should have to keep in mind.

1. The call to overloaded method is bonded at compile time.

2. The concept of method overloading is also known as compile-time polymorphism in java.

3. Method overloading is generally done in the same class. But it can also be done in the subclass. You will have to make a relationship between the parent class and child class by using extends keyword for it.

4. Method overloading in Java cannot be done by changing the return type of the method because there may occur ambiguity. But the overloaded methods can change the return type.

5. The private methods can be overloaded in Java

6. The final methods can be overloaded in Java.

7. The main method can also be overloaded in Java.

8. Both static and instance methods can be overloaded in Java. Method overloading is possible when two or more static methods with the same name, but the difference in the list of parameters.

## When to use Method overloading in Java

Method overloading is a powerful feature in Java but should be used as per needs. It should be used when you actually need multiple methods with different parameters but methods do the same thing.

If multiple methods perform different tasks, don’t use the method overloading concept. Method overloading is done in java due to the following needs:

* Method overloading is done to reuse the same method name.
* It is done to make the program logically more readable and understandable.
* It is used to achieve the compile-time polymorphism in Java.

### Method overloading by changing number of arguments

public class Addition

{

// Method to calculate the sum of two numbers.

// Declare an instance method sum with two parameters a and b having data type int.

void sum(int a, int b)

{

int s = a + b;

System.out.println("Sum of two numbers: " +s);

}

// Method to calculate sum of three numbers.

// Declare an instance method sum with three parameters a,b, and c having data type int.

// Here, the method sum() is overloaded having the same method name.

void sum(int a, int b, int c)

{

int t = a + b + c;

System.out.println("Sum of three numbers: " +t);

}

public static void main(String[] args)

{

Addition a = new Addition();

a.sum(10, 20); // It will call sum() method to calculate sum of two arguments.

a.sum(50, 100, 200); // It will call sum() method to calculate sum of three arguments.

}

}

### ****2. Method overloading by changing data type of parameters****

public class Subtraction

{

// Declare an instance method sub with two parameters x and y having data type int.

int sub( int x, int y)

{

int a = x - y;

System.out.println("Subtraction of two numbers: " +a);

return a;

}

// Declare an instance method sub with two parameters x and y having data type double.

double sub(double x, double y)

{

double b = x - y;

System.out.println("Subtraction of two numbers: " +b);

return b;

}

public static void main(String[] args)

{

Subtraction s = new Subtraction();

s.sub(6, 5); // It will call the method sub() to calculate the subtraction of two int type arguments.

s.sub(20.8, 10.9); // It will call the method sub() to calculate the subtraction of two double type arguments.

}

}

### ****3. Method overloading by changing sequence of data type of parameters****

public class Multiplication

{

void multiply(int a, double b)

{

double m1 = a \* b;

System.out.println("Multiplication of two numbers: " +m1);

}

void multiply(double a, int b)

{

double m2 = a \* b;

System.out.println("Multiplication of two numbers: " +m2);

}

public static void main(String[] args)

{

Multiplication m = new Multiplication();

m.multiply(10, 20.5); // It will call method multiply() to calculate the multiplication of two arguments int and double.

m.multiply(10.5, 30); // It will call method multiply() to calculate the multiplication of two arguments double and int.

}

}

## Java Method overloading done in Subclass

public class SuperClass

{

void msg(int x, int y)

{

System.out.println("Hello Java");

}

}

public class Subclass extends SuperClass

{

void msg(double x, double y)

{

System.out.println("Welcome you in Java programming");

}

}

public class Test

{

public static void main(String[] args)

{

Subclass sc = new Subclass();

sc.msg(10, 20);

sc.msg(2.5, 3);

}

}

**In Java, Method overloading cannot be done when the return type, method name, and argument list are the same because there may occur ambiguity**

public class ReturnTypeEx1

{

public int m1(int a, int b) // Duplicate method error.

{

int x = a + b;

return x;

}

public int m1(int c, int d) // Duplicate method error.

{

int y = c \* d;

return y;

}

}

public class ReturntypeTest

{

public static void main(String[] args)

{

ReturnTypeEx1 obj = new ReturnTypeEx1();

int sum = obj.m1(20, 30);

System.out.println(sum);

int multiply = obj.m1(20,30);

System.out.println(multiply);

}

}

Example 2

public class ReturnTypeEx1

{

public int m1(int a, int b) // Return type is int.

{

int x = a + b;

return x;

}

public double m1(int c, int d) // Return type is double.

{

int y = c \* d;

return y;

}

}

public class ReturntypeTest

{

public static void main(String[] args)

{

ReturnTypeEx1 obj = new ReturnTypeEx1();

int sum = obj.m1(20, 30);

System.out.println(sum);

int multiply = obj.m1(20,30);

System.out.println(multiply);

}

}

**//WAP explicit type casting**

public class ExplicitTest

{

double d = 100.04;

void conversion()

{

// explicit type casting

long l = (long)d;

int i = (int)l;

System.out.println("Double value "+d); // fractional part lost.

System.out.println("Long value "+l); // fractional part lost.

System.out.println("Int value "+i);

}

public static void main(String[] args)

{

ExplicitTest obj = new ExplicitTest();

obj.conversion();

}

}

**//implicit and Excplicit**

**public class ExplicitTest**

**{**

**public static void main(String[] args)**

**{**

**byte b;**

**int i = 257;**

**double d = 323.142;**

**System.out.println("Conversion of int to byte.");**

**b = (byte) i;**

**System.out.println("i = " + i + " b = " + b);**

**System.out.println("nConversion of double to int.");**

**i = (int) d;**

**System.out.println("d = " + d + " b = " + i);**

**System.out.println("nConversion of double to byte.");**

**b = (byte) d;**

**System.out.println("d = " + d + " b = " + b);**

**}**

**}**

# Class Casting in Java