**Overloading in Java**

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 both. Overloading is related to compile-time (or static) polymorphism.

**public** **class** OverloadingExample {

// 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[]) {

OverloadingExample s = **new** OverloadingExample();

System.***out***.println(s.sum(10, 20));

System.***out***.println(s.sum(10, 20, 30));

System.***out***.println(s.sum(10.5, 20.5));

}

}

**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);

}

}

**Question Arises:  
Q. What if the exact prototype does not match with arguments.**  
Ans.  
Priority wise, compiler take these steps:

1. Type Conversion but to higher type(in terms of range) in same family.
2. Type conversion to 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).

Let’s take an example to clear the concept:-

**class** UseDemo {

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

**byte** a = 25;

Demo obj = **new** Demo();

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

// byte argument

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

obj.show(250); // int

obj.show('A');

/\*

\* Since char is not available, so the data type higher than char in terms of

\* range is int.

\*/

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

/\*

\* since float data type is not available and so it's higher data type, so at

\* this step their will be an error.

\*/

obj.show(7.5);

}

}

**What is the advantage?**  
We don’t have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2, … or sum2Int, sum3Int, … etc.

**Can we overload methods on return type?**  
We **cannot** overload by return type. This behavior is same in C++. Refer this for details

|  |
| --- |
| public class Main {      public int foo() { return 10; }        // compiler error: foo() is already defined      public char foo() { return 'a'; }        public static void main(String args[])      {      }  } |

However, Overloading methods on return type are possible in cases where the data type of the function being called is explicitly specified. Look at the examples below :

// Java program to demonstrate the working of method

// overloading in static methods

public class Main {

    public static int foo(int a) { return 10; }

    public static char foo(int a, int b) { return 'a'; }

    public static void main(String args[])

    {

        System.out.println(foo(1));

        System.out.println(foo(1, 2));

    }

}

// Java program to demonstrate working of method

// overloading in  methods

class A {

    public int foo(int a) { return 10; }

    public char foo(int a, int b) { return 'a'; }

}

public class Main {

    public static void main(String args[])

    {

        A a = new A();

        System.out.println(a.foo(1));

        System.out.println(a.foo(1, 2));

    }

}

**Can we overload static methods?**  
The answer is ‘**Yes**’. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program. Refer [this](https://www.geeksforgeeks.org/can-we-overload-or-override-static-methods-in-java/) for details.

**Can we overload methods that differ only by static keyword?**  
We **cannot** overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. Refer [this](https://www.geeksforgeeks.org/can-we-overload-or-override-static-methods-in-java/) for details.

**Can we overload main() in Java?**  
Like other static methods, we **can** [overload main() in Java](https://www.geeksforgeeks.org/gfact-48-overloading-main-in-java/). Refer overloading main() in Java for more details.

// A Java program with overloaded main()

import java.io.\*;

public class Test {

    // Normal main()

    public static void main(String[] args)

    {

        System.out.println("Hi Geek (from main)");

        Test.main("Geek");

    }

    // Overloaded main methods

    public static void main(String arg1)

    {

        System.out.println("Hi, " + arg1);

        Test.main("Dear Geek", "My Geek");

    }

    public static void main(String arg1, String arg2)

    {

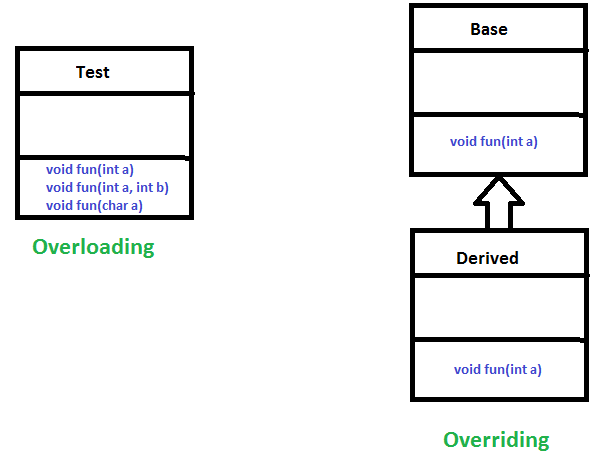
        System.out.println("Hi, " + arg1 + ", " + arg2);

    }

}

**Does Java support Operator Overloading?**  
Unlike C++, Java doesn’t allow user-defined overloaded operators. Internally Java overloads operators, for example, + is overloaded for concatenation.

**What is the difference between Overloading and**[Overriding](https://www.geeksforgeeks.org/overriding-in-java/)**?**

* Overloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.  
  [](http://media.geeksforgeeks.org/wp-content/uploads/OverridingVsOverloading.png)
* Overloading is an example of compiler time polymorphism and overriding is an example of run time polymorphism.

**Different ways of Method Overloading in Java**

Java can distinguish the methods with **different method signatures**. i.e. the methods can have the same name but with different parameters list (i.e. the number of the parameters, the order of the parameters, and data types of the parameters) within the same class. 

* Overloaded methods are differentiated based on the number and type of the parameters passed as an argument to the methods.
* You can not define more than one method with the same name, Order and the type of the arguments. It would be a compiler error.
* The compiler does not consider the return type while differentiating the overloaded method. But you cannot declare two methods with the same signature and different return type. It will throw a compile-time error.   
  If both methods have the same parameter types, but different return type, then it is not possible. [(Java SE 8 Edition, §8.4.2)](https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.2)

**Why do we need Method Overloading?**

If we need to do some kind of the operation with different ways i.e. for different inputs. In the example described below, we are doing the addition operation for different inputs. It is hard to find many meaningful names for a single action. 

**Different ways of doing overloading methods**

Method overloading can be done by changing:

* The number of parameters in two methods.
* The data types of the parameters of methods.
* The Order of the parameters of methods.

**Method 1: By changing the number of parameters.**

**class** Addition {

// adding two integer values.

**public** **int** add(**int** a, **int** b) {

**int** sum = a + b;

**return** sum;

}

// adding three integer values.

**public** **int** add(**int** a, **int** b, **int** c) {

**int** sum = a + b + c;

**return** sum;

}

}

**public** **class** GFG {

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

Addition ob = **new** Addition();

**int** sum1 = ob.add(1, 2);

System.***out***.println("sum of the two integer value :" + sum1);

**int** sum2 = ob.add(1, 2, 3);

System.***out***.println("sum of the three integer value :" + sum2);

}

}

**Method 2: By changing the Data types of the parameters**

**class** Addition {

// adding three integer values.

**public** **int** add(**int** a, **int** b, **int** c) {

**int** sum = a + b + c;

**return** sum;

}

// adding three double values.

**public** **double** add(**double** a, **double** b, **double** c) {

**double** sum = a + b + c;

**return** sum;

}

}

**public** **class** GFG {

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

Addition ob = **new** Addition();

**int** sum2 = ob.add(1, 2, 3);

System.***out***.println("sum of the three integer value :" + sum2);

**double** sum3 = ob.add(1.0, 2.0, 3.0);

System.***out***.println("sum of the three double value :" + sum3);

}

}

**Method 3: By changing the Order of the parameters**

**class** Geek {

**public** **void** geekIdentity(String name, **int** id) {

System.***out***.println("geekName :" + name + " " + "Id :" + id);

}

**public** **void** geekIdentity(**int** id, String name) {

System.***out***.println("Id :" + id + " " + "geekName :" + name);

}

}

**public** **class** GFG {

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

Geek geek = **new** Geek();

geek.geekIdentity("Mohit", 1);

geek.geekIdentity("shubham", 2);

}

}

**What happens when method signature is the** **same and the return type is different?**

The compiler will give an error as the return value alone is not sufficient for the compiler to figure out which function it has to call. More details: [https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.2.](http://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.2)  
Only if both methods have different parameter types (so, they have a different signature), then Method overloading is possible.

**class** Addition {

// adding two integer value.

**public** **int** add(**int** a, **int** b) { // Error Duplicate method add(int, int) in type Addition

**int** sum = a + b;

**return** sum;

}

// adding three integer value.

**public** **double** add(**int** a, **int** b) {// Error Duplicate method add(int, int) in type Addition

**double** sum = a + b + 0.0;

**return** sum;

}

}

**public** **class** GFG {

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

**try** {

Addition ob = **new** Addition();

**int** sum1 = ob.add(1, 2);

System.***out***.println("sum of the two integer value :" + sum1);

**int** sum2 = ob.add(1, 2);

System.***out***.println("sum of the three integer value :" + sum2);

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

**Method overloading and null error in Java**

In Java it is very common to overload methods. Below is an interesting Java program.

**public** **class** GFG {

// Overloaded methods

**public** **void** fun(Integer i) {

System.***out***.println("fun(Integer ) ");

}

**public** **void** fun(String name) {

System.***out***.println("fun(String ) ");

}

// Driver code

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

GFG mv = **new** GFG();

// This line causes error The method fun(Integer) is ambiguous for the type GFG

mv.fun(**null**);

}

}

The reason why we get compile time error in the above scenario is, here the method arguments Integer and String both are not primitive data types in Java. That means they accept null values. When we pass a null value to the method1 the compiler gets confused which method it has to select, as both are accepting the null.  
This compile time error wouldn’t happen unless we intentionally pass null value. For example see the below scenario which we follow generally while coding.

**public** **class** GFG {

// Overloaded methods

**public** **void** fun(Integer i) {

System.***out***.println("fun(Integer ) ");

}

**public** **void** fun(String name) {

System.***out***.println("fun(String ) ");

}

// Driver code

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

GFG mv = **new** GFG();

Integer arg = **null**;

// No compiler error

mv.fun(arg);

}

}

In the above scenario if the “arg” value is null due to the result of the expression, then the null value is passed to method1. Here we wouldn’t get compile time error because we are specifying that the argument is of type Integer, hence the compiler selects the method1(Integer i) and will execute the code inside that.

Note: This problem wouldn’t persist when the overriden method arguments are primitive data type. Because the compiler will select the most suitable method and executes it.