**MethodOverLoadig :**

1. **What is method signature?**

Ans : Methodname + arguments considered as Method Signature. (Note : access modifier and return types will not considered as method signature)

Ex : public void m1(int i)

In above example m1(int i) is considered as method signature.

1. What is method overloading?

Ans : **Method Overloading** is a feature that allows a class to have more than one **method** having the same name, if their argument lists are different. It is similar to constructor **overloading in Java**, that allows a class to have more than one constructor having different argument lists.

**Points to Note:**

1. **Method overloading** is an example of [Static Polymorphism](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/)

2. Static Polymorphism is also known as compile time binding or early binding.  
3. [Static binding](https://beginnersbook.com/2013/04/java-static-dynamic-binding/) happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

**Three ways to overload a method**

1. Number of parameters.  
For example: This is a valid case of overloading

add(int, int)

add(int, int, int)

2. 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)

**Invalid case of method overloading:**  
When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

int add(int, int)

float add(int, int)

## Method Overloading and 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.

**What it has to do with method overloading?**  
Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.  
Lets take an example to see what I am talking here:

class Demo{

void disp(int a, double b){

System.out.println("Method A");

}

void disp(int a, double b, double c){

System.out.println("Method B");

}

public static void main(String args[]){

Demo obj = new Demo();

/\* I am passing float value as a second argument but

\* it got promoted to the type double, because there

\* wasn't any method having arg list as (int, float)

\*/

obj.disp(100, 20.67f);

}

}

Output:

Method A

As you can see that I have passed the float value while calling the disp() method but it got promoted to the double type as there wasn’t any method with argument list as (int, float)

But this type promotion doesn’t always happen, lets see another example:

class Demo{

void disp(int a, double b){

System.out.println("Method A");

}

void disp(int a, double b, double c){

System.out.println("Method B");

}

void disp(int a, float b){

System.out.println("Method C");

}

public static void main(String args[]){

Demo obj = new Demo();

/\* This time promotion won't happen as there is

\* a method with arg list as (int, float)

\*/

obj.disp(100, 20.67f);

}

}

Output:

Method C

As you see that this time type promotion didn’t happen because there was a method with matching argument type.  
**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

short → int → long

int → long → float → double

float → double

long → float → double

**Different cases of Overloading :**

**package** com.methodOverLoading;

**public** **class** DifferentCasesOfOverLoading {

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

DifferentCasesOfOverLoading obj= **new** DifferentCasesOfOverLoading();

Test t = obj.**new** Test();

//Case 1 - Automatic Type promotion in overloading

//If exact argument type is not found, compiler will do automatic type promotion and then will check whether matched methos is avialable or not.

//If matched method is available it will be considered and if matched methods is not available then compiler promots argument to the next level.

//This process will be continued until all possible promotions. Still if the matched methods is not available then compiler will raise compile time error

t.m1(10);

t.m1('A');

//Type Promotion :-

//char --> int --> long --> float --> double

//byte --> short --> int --> long --> float --> double

//Case 2

Test1 t1 = obj.**new** Test1();

t1.m1(**new** Object());//Object version

t1.m1("JAK");//String version

t1.m1(**null**);//String version (While resolving overloaded methods, compiler will always give the precedence for child type argument when compared with parent type argument]

//As String is child of Object[parent] class in above context, so String version will execute

Test2 t2 = obj.**new** Test2();

t2.m1("JAK");//String version

t2.m1(**new** StringBuffer("JAK"));//StringBuffer version

t2.m1(**null**);//will get compile time error as "reference to m1() is ambiguous"

//Reason for above compile error is - As String and StringBuffer both are at same level [Parent class of both String and StringBuffer is Object] which extends Object class.

//Hence compiler can't decide in this situation, hence it will throw the compile time error.

Test3 t3 = obj.**new** Test3();

t3.m1(10, 10.5f);

t3.m1(10.5f, 10);

t3.m1(10, 10); // ambiguous error, [event though first argument match with int argument of "m1(int,float)" method and there is chance of second argument promotes to float - compiler will throw ambiguous error,

//because we considered arguments from left to right, if few countries may treat them right to left. So problem will arise.

//As java platform independent and not a country/region dependent, to make unique compiler will throw the error.

//Note: if code doesn't have m1(float,int) overloaded method the above problem will not arise.

t3.m1(10.5f, 10.5f);//compile time error

Test4 t4 = obj.**new** Test4();

t4.m1();//var-arg method - As var-arg method will match for zero or more arguments

t4.m1(10,20);//var-arg method

t4.m1(10);//Single argument method [ Because general method concept came in 1.0 version of java, var-arg concept came in 1.5 version of java, to make support/compatability preference always go to old versions only]

//In general Var-arg method will get least priority.i.e. if no other method matched then only var-arg method will get chance.

}

**class** Test{

**public** **void** m1(**int** i) {

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

}

**public** **void** m1(**float** f) {

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

}

**public** **void** m1(**double** d) {

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

}

}

**class** Test1{

**public** **void** m1(String str) {

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

}

**public** **void** m1(Object obj) {

System.***out***.println("object version");

}

}

**class** Test2{

**public** **void** m1(String str) {

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

}

**public** **void** m1(StringBuffer obj) {

System.***out***.println("StringBuffer version");

}

}

**class** Test3{

**public** **void** m1(**int** i, **float** f) {

System.***out***.println("int-float version");

}

**public** **void** m1(**float** f, **int** i) {

System.***out***.println("float-int version");

}

}

**class** Test4{

**public** **void** m1(**int** i) {

System.***out***.println("single argument method");

}

**public** **void** m1(**int**... i) {

System.***out***.println("var-arg method");

}

}

}