Method Overloading

Defintion: When a class has more than one method with the same name but with different number of arguments or different types of arguments then we can say that the method is overloaded.

Compiler checks **method signature** for duplicate methods or for method overloading. method signature consist of three things, **1) Method Name   2) Number Of Arguments   3) Types of arguments.**

**Execution Process:**

1)Compiler first checks method name. if it is same it checks number of arguments. If the number of arguments are different then it will not check for the type of arguments.

2) If number of arguments are same then compiler checks types of arguments.

3) If types of arguments are also same, then compiler will give duplicate method error

4) If types of arguments are not same, then compiler will treat them as methods are overloaded.

Example:

public class MethodOverloading

{

**void methodOverloaded()**

**{**

**//No argument method**

**}**

**void methodOverloaded(int i)**

**{**

**//One argument is passed**

**}**

**void methodOverloaded(double d)**

**{**

**//One argument is passed but type of argument is different**

**}**

**void methodOverloaded(int i, double d)**

**{**

**//Two argument method**

**//Method signature of this method is methodOverloaded(int, double)**

**}**

**void methodOverloaded(double d, int i)**

**{**

**//It is also two argument method but type of arguments changes**

**//Method signature of this method is methodOverloaded(double, int)**

**}**

**void methodOverloaded(double d1, int i1)**

**{**

**//It has same method signature methodOverloaded(double, int) as of above method**

**//So, it is a Duplicate method, You will get compile time error here**

**}**

**void differentMethod()**

**{**

**//Different method**

**}**

}

**Some important cases of Method overloading:**

**Case 1:** Automatic promotion in Java

While resolving mehtod overloading if exact method match wont available then we wont get compile time error immediately.

First it promotes the argument to the next level and check whether matched method is available or not.

If matched method is available then it will be considered.

If matched method is not available then compiler promotes once again to the next level. This process is continued until all promotions.

Still if it is not matched then will get compile time error.

Possible promotions:

Byte🡪short🡪int🡪long🡪float🡪double

Char🡪int🡪long🡪float🡪double

public class AutomaticPromotion {

public void m1(int i) {

System.out.println("Int-args");

}

public void m1(float f) {

System.out.println("Float-args"); }

public static void main(String[] args) {

AutomaticPromotion AP=new AutomaticPromotion();

AP.m1(10);//Int-args

AP.m1(10.5f);//Float-args

AP.m1('a'**);//Int-args because char is promoted to int**

AP.m1(10l);**//Float-args because long is promoted to float**

//AP.m1(10.5); **//compile error because we cant promote double**

}

}

**Case 2:**

While resolving overloaded methods compiler will always gives precedence for child type argument when compared to parent type argument.

In the below example Object is parent for String class and null is valid value for both String and Object. So the compiler will give preference to child type argument. So for cp.m1(null) the output is String version.

**public** **class** ChildParent {

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

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

}

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

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

}

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

ChildParent cp=**new** ChildParent();

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

cp.m1("Narayana");//String version

cp.m1(**null**);//String version

}

}

**Case 3:** When there are two child objects at the same level then we will get compile time error.

In the below example String and StringBuffer are the child objects of Parent object Object. And null is valid value for both String and String Buffer. So for ccsl.m1(null); we will get compile error.

**public** **class** ChildClassSameLevel {

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

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

}

**public** **void** m1(StringBuffer s1) {

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

}

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

ChildClassSameLevel ccsl=**new** ChildClassSameLevel();

ccsl.m1("Narayana");

ccsl.m1(**new** StringBuffer("Narayana"));

//ccsl.m1(null);//compile error: The method m1(String) is ambiguous for the type ChildClassSameLevel

}

}

**Case 4:**

**package** methodOverloading;

**public** **class** Case4 {

**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");

}

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

Case4 c4=**new** Case4();

c4.m1(10, 10.5f);

c4.m1(10.5f, 10);

//c4.m1(10, 10); //The method m1(int, float) is ambiguous for the type Case4

//c4.m1(10.5f, 10.5f); //The method m1(float, int) in the type Case4 is not applicable for the arguments (float, float)

}

}

For c4.m1(10,10) this is matched for both methods m1(int I,float f) and m1(float f, int i) as we can promote int to float. So we have two methods with same signature. We get compile error saying m1(int, float) is ambiguous

For c4.m1(10.5f,10.5f) we cant convert the second argument 10.5f (float) value to int. so we get compile time error saying no method m1(float, int)

**Case 5:** In general var arg method will get least priority. If no other method matched var-arg method will get a chance. It is same as default case inside switch.

Var-arg method i.e. we can pass n number of arguments.

**package** methodOverloading;

**public** **class** VarargCase5 {

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

System.***out***.println("General method");

}

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

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

}

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

VarargCase5 vc5=**new** VarargCase5();

vc5.m1(); //var-arg method

vc5.m1(10,20);//var-arg method

vc5.m1(10); //General method

}

}

**Case 6: overloading loading using Reference type**

In overloading method resolution always takes care by compiler based on reference type.

For //3 reference type is parent so the output we get is Animal version.

**public** **class** ReferenceCase5 {

**public** **void** m1(Animal a) {

System.***out***.println("Animal Version");

}

**public** **void** m1(Monkey m) {

System.***out***.println("Monkey Version");

}

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

ReferenceCase5 c6=**new** ReferenceCase5();

//1

Animal a=**new** Animal();

c6.m1(a); //Animal Version

//2

Monkey m=**new** Monkey();

c6.m1(m); //Monkey Version

//3

Animal a1=**new** Monkey();

c6.m1(a1); //Animal Version

}

}

**Rules of overloading:**

Rule 1: Overloaded methods may have same return types or different return types. It does not effect method overloading

Rule 2: Overloaded methods may have same access modifiers or different access modifiers. It also does not effect method overloading

Rule 3: Overloaded methods may be static or non-static. This also does not effect method overloading.

1)Overloaded methods may have same return types or different return types. It does not effect method overloading.

Example:

Public class methodoverloaidngreturntypeexample

{

void m1()

{

Syso(“void return type”);

}

int m1(int i)

{

return i;

Syso(“int return type”);

}

int m1(double d)

{

//Same return type as of above method

return 0;

Syso(“int return type”);

}

void methodOverloaded(double d)

    {

        //Duplicate method because it has same method signature as of above method

    }

}

2)Overloaded methods may have same access modifiers or different access modifiers. It also does not effect method overloading.

public class MethodOverloading

{

private void methodOverloaded()

{

**//No argument, private method**

}

private int methodOverloaded(int i)

{

**//One argument private method**

return i;

}

protected int methodOverloaded(double d)

{

**//Protected Method**

return 0;

}

public void methodOverloaded(int i, double d)

{

**//Public Method**

}

}

Rule 3:Overloaded methods may be static or non-static. This also does not effect method overloading.

public class MethodOverloading

{

private static void methodOverloaded()

{

**//No argument, private static method**

}

private int methodOverloaded(int i)

{

**//One argument private non-static method**

return i;

}

static int methodOverloaded(double d)

{

**//static Method**

return 0;

}

public void methodOverloaded(int i, double d)

{

**//Public non-static Method**

}

}