Var arg method:

It stands for variable arguments method.

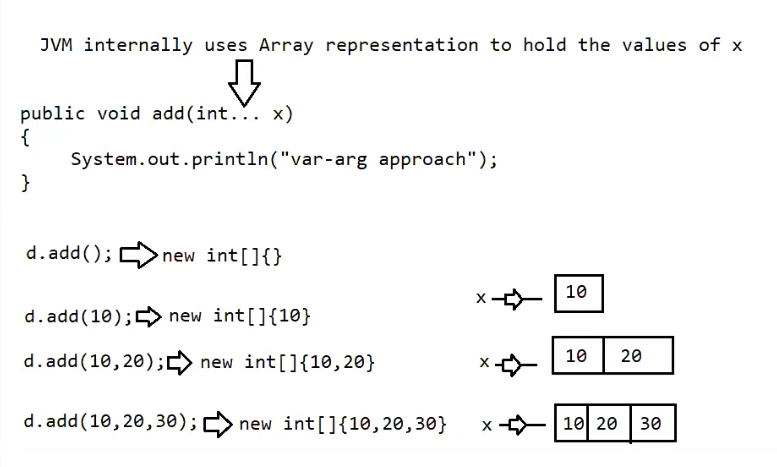
* Until JDK 4, we can’t declare a method with variable no. of arguments. If there is any change in the number of arguments, we have to declare a new method. This approach increases the length of the code and reduces readability.
* Before JDK 5, variable-length arguments could be handled in two ways.
  + - One uses an overloaded method (one for each).
    - Another puts the arguments into an array and then passes this array to the method. Both of them are potentially error-prone and require more code.

But in jdk 1.5, we can write single method which can handle variable number of arguments (but all of them should be of same type)

Syntax: methodOne (datatype… variableName)

…=> It stands for ellipse (… compiler understands it is variable arguments)

Eg: Var\_Args\_Eg1



Note: internally var-arg method will be converted to single dimensional array, so we can access var arg method arguments using index.

Eg: Var\_Args\_Eg2

Eg: Var\_Args\_Eg3

Case1: valid signatures

1. public void methodOne(int… x)
2. public void methodOne(int…x)
3. public void methodOne(int …x)

Case2:

We can mix normal argument with var argument

public void methodOne (int x, int… x)

public void methodOne (String s, int… x)

Case3:

While mixing var argument with normal argument var argument should always be last

public void methodOne (int… x, int y) // invalid

case 4:

In an argument list there should only be one var argument

public void methodOne (int… x, int…x) // invalid

case 5:

we can overload var arg method, but var arg method will get call only if none of the matches are found.

(just like default statement of switch case )

Eg: Var\_Arg\_Overload

Case 6:

public void methodOne (int… x) => internally it is be replace by int[] x , so it can be replaced by int[] x

Eg: Internal\_replacement\_In\_Var\_Args

Case 7:

public void methodOne (int… x)

public void methodOne (int[] x)

// we can’t have 2 methods with same signature, it leads to compile time error.

Eg: Var\_Arg\_Method\_Signature

single dimensional array vs var arg method:

1. wherever single dimensional array is present we can replace it with var args

Eg: public static void main (String[] args ) => String… args

1. wherever var arg is present we cannot replace it with single dimensional array

Eg: public void methodOne (String… args ) => String[] args // invalid

// only in some cases are valid we can do the second case in main method, but it is not possible in normal method containing var args .

Note:

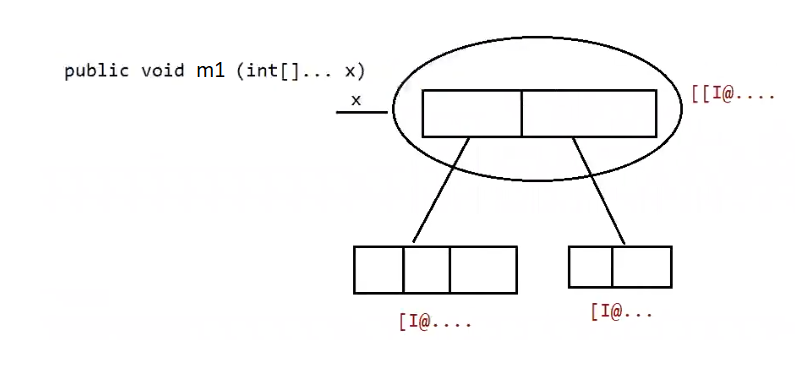
m1(int… x) => we can make a call to this method by passing group of int values ( as parameters ) and x will become 1d array (int[] x) .

m1(int[] x) => we can call this method by passing 1d array only

Note :

Eg: One\_Dimensional\_Array\_Var\_Args

Eg: Two\_Dimensional\_Array\_Var\_Args



m1(int…x)

=>we can call this method by passing group of int values, so it becomes 1-D Array

m 1(int[]… x)

* we can call this method by passing group of 1D int[], so it becomes 2-D Array

Wrapper class:

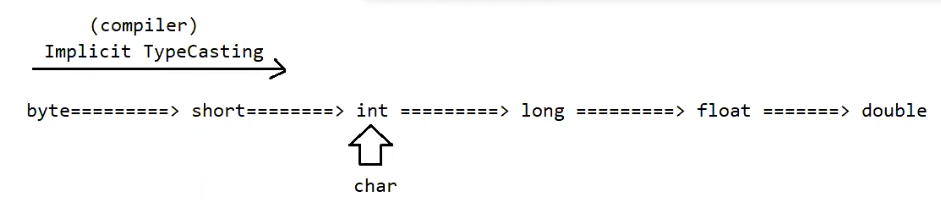
1. Autoboxing
2. Widening ( implicit type casting done by compiler, applicable for both primitive and wrapper)
3. Var - Args

Case1:

Widening vs autoboxing

Eg: AutoBoxing\_Vs\_Widening

Primitive ====🡺 do type casting ==🡺 found ===🡺 long (binding happens by compiler)



Eg: Widening\_Vs\_Var\_Args

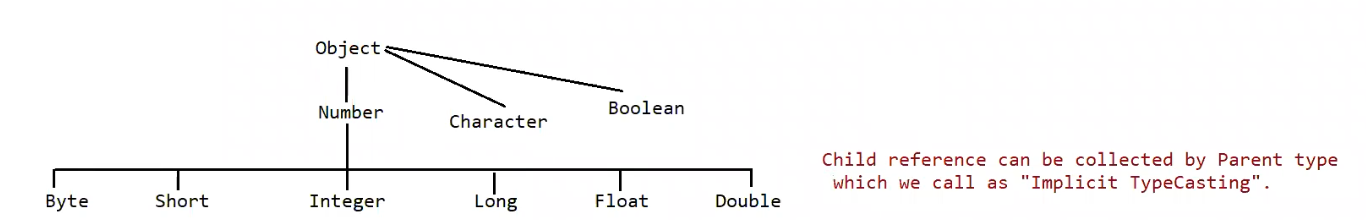
Primitive ====🡺 do type casting ==🡺 found ===🡺 long ( binding happens by compiler)

Eg: AutoBoxing\_Vs\_Var\_Args

// go through the code

Eg: AutoBoxing\_Vs\_UnBoxing

Note: widening followed by autoboxing is not allowed in java, but autoboxing followed by widening / typecasting is allowed. If it is not overloaded methods



Eg: Conversion\_AutoBoxing

// o/p: Number, since it is the first immediate parent.

Which of the following declarations are valid?

1. int i=10;//valid

2. Integer I=10; //AutoBoxing(valueof())

3. int i=10L; // invalid(long===> int)

4. Long I = 10L; //AutoBoxing(valueOf())

5. Long I = 10; //AutoBoxing=> Integer====> Number, Object, so invalid

6. long I = 10; // valid(int===> long) // widening possible for primitives not possible for Wrapper classes

7. Object o=10; //AutoBoxing==> Integer ===> Number, Object so valid parent reference child object

8. double d=10; // valid(int===> double)

9. Double d=10; // AutoBoxing=> Integer====> Number, Object, so invalid

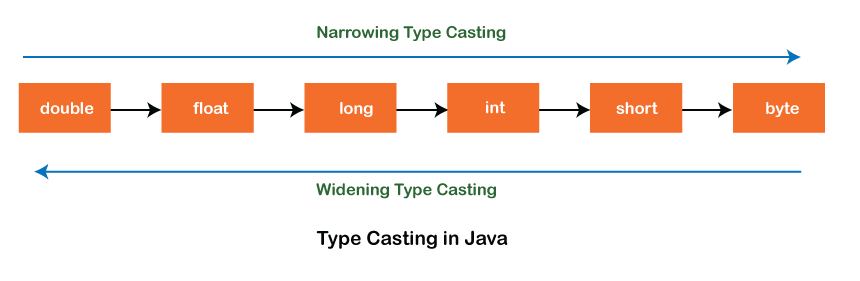
10. Number n=10; //AutoBoxing==> Integer ===> Number, Object so valid

Only same type of data can be stored in AutoBoxing

Note:

In general var-arg method will get the least priority, i.e if no other method is matched only then var-arg method will get a chance.

It is exactly same as default inside switch



While resolving the overloaded methods the compiler will give the precedence in the following order.

1. Widening
2. AutoBoxing

Eg: Var\_Args\_Eg4

Here variable argument in method go is the first argument, so it throws compile time error. variable argument should be the last one.

Eg: Var\_Args\_With\_Respect\_To\_Main\_Method

// go through the code

Eg: Var\_Args\_Eg5

// compiler will search for the exact match of a method that accepts two int arguments, it if it is found that method is loaded.

Variable arguments is always given the least priority.