

Var Args in Java :

* It is a Variable Argument which is introduced from JDK 1.5V onwards.

* It is represented by exactly ... (3 dots)

* Internally, It is an array which can accept 0 to n number of arguments of same type OR different type.

* Var args we can accept as a method/constructor parameter only.

* Var args must be only one and last parameter.

Example :

public void accept(int ...x, int ...y) //Invalid

{

}

public void accept(int ...x, int y) //Invalid

{

}

public void accept(int ...x, float ...y) //Invalid

{

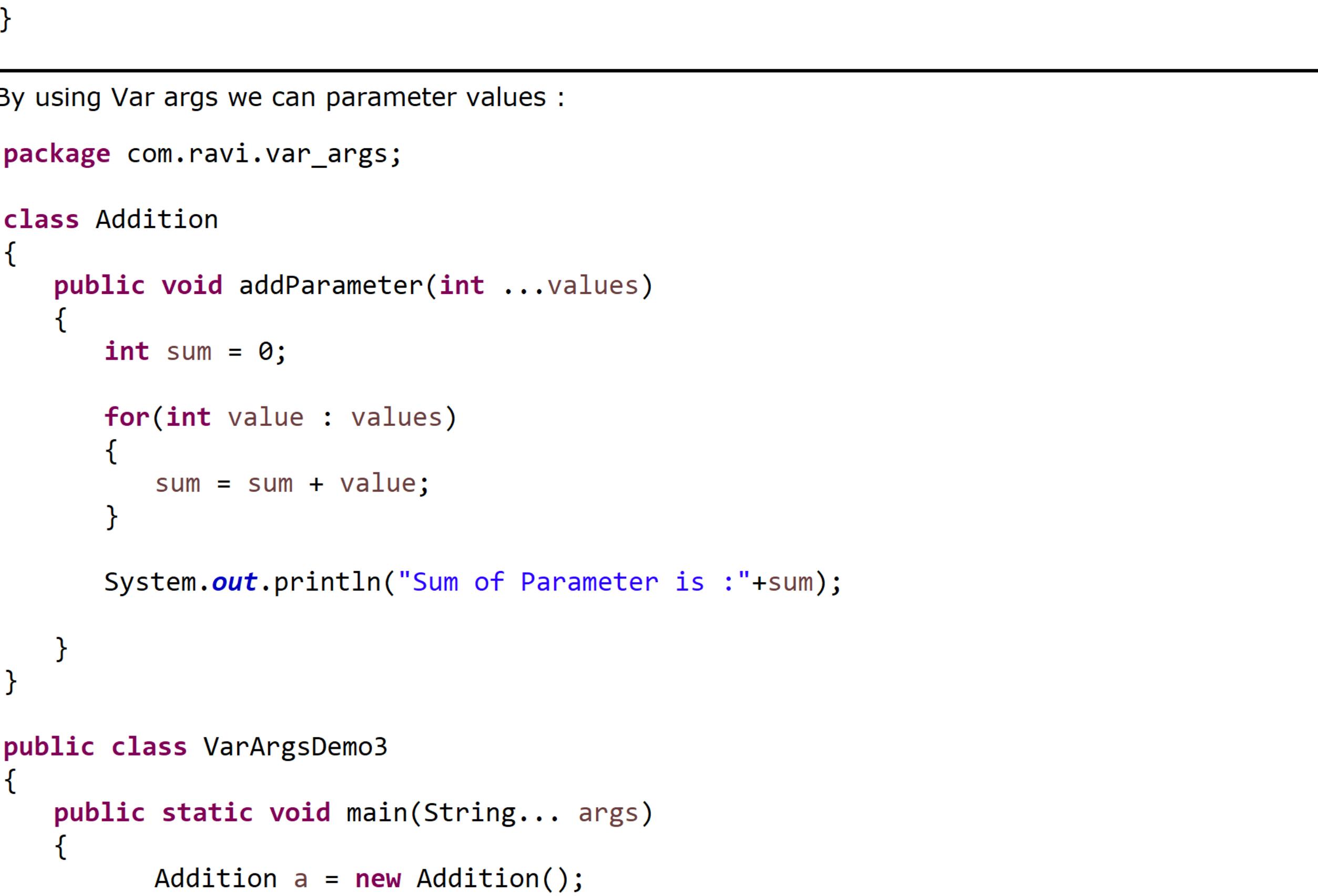
}

public void accept(int x, int ...y) //Valid

{

}

* In comparison to array, Var args will directly take the appropriate values.



WAP to show that var args can accept 0 to n number of parameters.

```
package com.ravi.var_args;

class Test
{
    public void accept(int ...x)
    {
        System.out.println("Executed");
    }
}

public class VarArgsDemo1
{
    public static void main(String[] args)
    {
        Test t1 = new Test();
        t1.accept();
        t1.accept(5);
        t1.accept(5,10);
        t1.accept(5,10,15);
        t1.accept(10,20,30,40);
    }
}
```

WAP to show that var args can accept heterogeneous types of data.

```
package com.ravi.var_args;

class Hetro
{
    public void acceptHetro(Object ...x)
    {
        for(Object y : x)
        {
            System.out.println(y);
        }
    }
}

public class VarArgsDemo2
{
    public static void main(String[] args)
    {
        Hetro h = new Hetro();
        h.acceptHetro(12,78.90, 'A', false, "NIT", new String("Java"));
    }
}
```

By using Var args we can parameter values :

package com.ravi.var_args;

class Addition
{
 public void addParameter(int ...values)
 {
 int sum = 0;

 for(int value : values)
 {
 sum = sum + value;
 }

 System.out.println("Sum of Parameter is :" +sum);
 }
}

public class VarArgsDemo3
{
 public static void main(String... args)
 {
 Addition a = new Addition();
 a.addParameter(10,20,30,40);
 a.addParameter(1000,2000,3000);
 }
}

WAP that show var args must be only one and last parameter

package com.ravi.var_args;

class Sample
{
 /*
 public void accept(int ...x, int ...y) //Invalid
 {
 }

 public void accept(int ...x, float ...y) //Invalid
 {
 }
 */

 public void accept(int x, int ...y)
 {
 System.out.println("x value is :" +x);

 for(int z : y)
 {
 System.out.println(z);
 }
 }
}

public class VarArgsDemo4
{
 public static void main(String... args)
 {
 Sample s1 = new Sample();
 s1.accept(100, 200, 300, 400);
 }
}

Ambiguity issue while overloading a method :

* While method overloading, compiler has the choice to select the appropriate method based on the method parameter, If compiler has more than one method to select then compiler will provide the priority on the following basis but on the other hand If compiler is unable to select appropriate method then it will generate CE.

Rule 1 :

Most Specific data type rule :

double > float // [Here float is the most specific type]

float > long

long > int

int > char

int > short // [There is no relation between char and short]

short > byte

Rule 2 :

WAV [Widening Autoboxing Var Args]

While working with overloading, Compiler will give the priority in the following order

Widening -> Autoboxing -> Var Args

Rule 3 :

Nearest Data type OR Nearest class Rule :

Object Number Alpha

↓ ↓ ↓

String Integer Beta

/Programs :

class Test
{
 public void accept(double d)
 {
 System.out.println("double");
 }
 public void accept(float d)
 {
 System.out.println("float");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept(6);
 }
}

Note : Here float will be executed because float is the most nearest type/Specific type

class Test
{
 public void accept(int d)
 {
 System.out.println("int");
 }
 public void accept(char d)
 {
 System.out.println("char");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept('6');
 }
}

Here 6 is int type so int will be executed.

class Test
{
 public void accept(int ...d)
 {
 System.out.println("int");
 }
 public void accept(char ...d)
 {
 System.out.println("char");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept('6');
 }
}

char will be executed becoz char is more specific type.

class Test
{
 public void accept(short ...d)
 {
 System.out.println("short");
 }
 public void accept(byte ...d)
 {
 System.out.println("byte");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept('6');
 }
}

Here byte will be executed because byte is the specific type.

class Test
{
 public void accept(double ...d)
 {
 System.out.println("double");
 }
 public void accept(long ...d)
 {
 System.out.println("long");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept(18);
 }
}

Here value 18 is of type int so, we can't assign directly to byte and short, If we want, explicit type casting is reqd.

class Test
{
 public void accept(int ...d)
 {
 System.out.println("int");
 }
 public void accept(short ...d)
 {
 System.out.println("short");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept(18);
 }
}

Here 18 is of type int so, we can't assign directly to short, If we want, explicit type casting is reqd.

class Test
{
 public void accept(int ...d)
 {
 System.out.println("int");
 }
 public void accept(short ...d)
 {
 System.out.println("short");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept(18);
 }
}

Here value 18 is of type int so, we can't assign directly to short, If we want, explicit type casting is reqd.

class Test
{
 public void accept(int ...d)
 {
 System.out.println("int");
 }
 public void accept(short ...d)
 {
 System.out.println("short");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept(18);
 }
}

Here value 18 is of type int so, we can't assign directly to short, If we want, explicit type casting is reqd.

class Test
{
 public void accept(int ...d)
 {
 System.out.println("int");
 }
 public void accept(short ...d)
 {
 System.out.println("short");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept(18);
 }
}

Here value 18 is of type int so, we can't assign directly to short, If we want, explicit type casting is reqd.

class Test
{
 public void accept(int ...d)
 {
 System.out.println("int");
 }
 public void accept(short ...d)
 {
 System.out.println("short");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept(18);
 }
}

Here value 18 is of type int so, we can't assign directly to short, If we want, explicit type casting is reqd.

class Test
{
 public void accept(int ...d)
 {
 System.out.println("int");
 }
 public void accept(short ...d)
 {
 System.out.println("short");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept(18);
 }
}

Here value 18 is of type int so, we can't assign directly to short, If we want, explicit type casting is reqd.

class Test
{
 public void accept(int ...d)
 {
 System.out.println("int");
 }
 public void accept(short ...d)
 {
 System.out.println("short");
 }
}

public class AmbiguityIssue {

public static void main(String[] args)
 {
 Test t = new Test();
 t.accept(18);
 }
}

Here value 18 is of type int so, we