Variable Argument

In past releases (prior to Java 1.5), a method that took an arbitrary number of values required you to create

1. Either overload the method.
2. Or can take an array or Collection and pass the no of argument wrapped in array or Collection like List, Set or Map.

But the problem with this is to if we are overloading the method and we don't know about how many arguments we have to handle how many method will be created in the code i.e. the code will become clumsy or if he has not created sufficient method then again the codes need to be modified and complied so it's become repetitive task which is not a good programming practice and requires more maintenance .Now we can go for array also but why not we give this task to Java for creating an array and store the element in to that array to internally handle and let make programmer free of this.

varargs or variable arguments makes it possible for us to call one method with variable number of argument; means define only one method and call that method with zero or more than zero argument.

Syntax:

type… variable_Name.

Ellipses stands for variable argument java treats variable argument as an array of same data type. 3 dots is used to denote variable argument in a method and if there are more than one parameter, varargs arguments must be last, as better listed below

Important points about varargs in java

1. We can have only one varargs in the method.
2. Only the last argument of a method can be varargs.
3. According to java documentation, we should not overload a varargs method. We will see why it's not a good idea.

```
public class CalculateSum
    {
        int calculateSum(int... nums)
         {
        int sum = 0;
        for (int num : nums) {
                sum = sum + num;
                }
        return sum;
        }
```

```
public static void main(String[] args)
 {
CalculateSum ex = new CalculateSum ();

int sum = ex.calculateSum(1, 2);
System.out.println(sum);

sum = ex.calculateSum(2, 4, 5, 7);
System.out.println(sum);

int[] arr = { 1, 3, 5, 7, 9 };
sum = ex.calculateSum(arr);
System.out.println(sum);

} //main

} //class
```

Method Overloading in Varargs

Overloading allows different methods to have same name, but different signatures where signature can differ by number of input parameters or type of input parameters or both. We can overload a method that takes a variable-length argument by following ways:

Case 1:

```
public class VarargsMethodOverloadingDemo1
{
   public static void main(String[] args)
          {
      m();
          }
   static void m(float... x)    //varargs method with float datatype
          {
           System.out.println("float varargs");
           }
   static void m(int... x)   //varargs method with int datatype
          {
          System.out.println("int varargs");
          }
   static void m(double... x)   //varargs method with double datatype
            {
          System.out.println("double varargs");
          }
}
```

Output : int varargs

Case 2 – Methods with Varargs along with other parameters In this case, Java uses both the number of arguments and the type of the arguments to determine which method to call.

```
class  VarargsMethodOverloadingDemo1
{
      static void m(int ... a)
           {
      System.out.print("m(int ...):   Number of args: " + a.length +  " Contents: ");

        for(int x : a)
          System.out.print(x + " ");

      System.out.println();
           }

     static void m(boolean...  a)
          {
      System.out.print("m(boolean ...)  : Number of args: " + a.length + " Contents: ");

        for(boolean x : a)
            System.out.print(x + " ");

      System.out.println();
           }

     static void m(String msg, int ... a)
            {
      System.out.print("m(String, int ...): " +  msg + a.length +  " Contents: ");

        for(int x : a)
          System.out.print(x + " ");

      System.out.println();
   }

   public static void main(String args[])
   {
      m(1, 2, 3);
      m("Testing: ", 10, 20);
      m(true, false, false);
   }
}
```

Exact Parameter match vs Widening

```
class ExactMatchAndWidening  {
   static void go(short x) { System.out.print("short "); }
   static void go(int x) { System.out.print("int "); }
   static void go(long x) { System.out.print("long "); }
   static void go(double x) { System.out.print("double "); }

   public static void main(String [] args) {
      char c = 'a';
      byte b = 5;
      short s = 5;
      long l = 5;
      float f = 5.0f;
      go(c);       //go(int) --- widening
      go(b);       //go(short) -- -widening
      go(s);       //go(short) --- exact parameter match
      go(l);       //go(long)  --- exact parameter match
      go(f);       //go(double) --- widening
   }
}
```

In every case, when an exact match isn't found, the JVM uses the method with the smallest argument that is wider than the parameter.

char produces a slightly different effect, since if it do not find an exact **char** match,*it is promoted to* **int**.
char can not be promoted
byte and **short** whenever it is, it can be promoted **long**, **float** and **double** if **int** does not exist.

Widening vs varargs

```
class  WideningVsVarargs
  {
   static void go(long x, long y) { System.out.println("long,long");}
   static void go(byte... x) { System.out.println("byte... "); }

   public static void main(String[] args) {
      byte b = 5;
      go(b,b);     // which go() will be invoked?
   }
}
```

Widening will get preference over Variable arguments.

Method definition binding will be done as follows by the compiler
1. First try for parameter data type exact match
2. if exact match of parameters not available , then compiler will try for widening .
3.if widening is not applicable, then compiler will try for variable argument match. Variable argument will get the least priority