

JAVA 9

SafeVarargs Annotation Enhancements

This SafeVarargs Annotation was introduced in Java 7.

Prior to Java 9, we can use this annotation for final methods, static methods and constructors.

But from Java 9 onwards we can use for private methods also.

To understand the importance of this annotation, first we should aware var-arg methods and heap pollution problem.

What is var-arg method?

Until 1.4 version, we can't declared a method with variable number of arguments. If there is a change in no of arguments compulsory we have to define a new method. This approach increases length of the code and reduces readability.

But from 1.5 version onwards, we can declare a method with variable number of arguments, such type of methods are called var-arg methods.

```
1) public class Test
2) {
3)     public static void m1(int... x)
4)     {
5)         System.out.println("var-arg method");
6)     }
7)     public static void main(String[] args)
8)     {
9)         m1();
10)        m1(10);
11)        m1(10,20,30);
12)    }
13) }
```

Output

var-arg method

var-arg method

var-arg method

Internally var-arg parameter will be converted into array.

```
1) public class Test
2) {
3)     public static void sum(int... x)
4)     {
5)         int total=0;
6)         for(int x1 : x)
7)         {
8)             total=total+x1;
```

```

9)      }
10)     System.out.println("The Sum:" + total);
11)    }
12)    public static void main(String[] args)
13)    {
14)        sum();
15)        sum(10);
16)        sum(10,20,30);
17)    }
18) }

```

Output

The Sum:0
The Sum:10
The Sum:60

Var-arg method with Generic Type:

If we use var-arg methods with Generic Type then there may be a chance of Heap Pollution. At runtime if one type variable trying to point to another type value, then there may be a chance of ClassCastException. This problem is called Heap Pollution. In our code, if there is any chance of heap pollution then compiler will generate warnings.

```

1) import java.util.*;
2) public class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         List<String> l1= Arrays.asList("A","B");
7)         List<String> l2= Arrays.asList("C","D");
8)         m1(l1,l2);
9)     }
10)    public static void m1(List<String>... l)//argument will become List<String>[]
11)    {
12)        Object[] a = l;// we can assign List[] to Object[]
13)        a[0]=Arrays.asList(10,20);
14)        String name=(String)l[0].get(0);//String type pointing to Integer type
15)        System.out.println(name);
16)    }
17) }

```

Compilation:

javac Test.java

Note: Test.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

```
javac -Xlint:unchecked Test.java
```

warning: [unchecked] unchecked generic array creation for varargs parameter of type

```
List<String>[]
```

```
    m1(l1,l2);
```

^

warning: [unchecked] Possible heap pollution from parameterized vararg type List<String>

```
    public static void m1(List<String>... l)
```

^

2 warnings

Execution:

```
java Test
```

RE: java.lang.ClassCastException: java.base/java.lang.Integer cannot be cast to

java.base/java.lang.String

In the above program at runtime,String type variable name is trying to point to Integer type,which causes Heap Pollution and results ClassCastException.

```
String name = (String)l[0].get(0);
```

Need of @SafeVarargs Annotation:

Very few Var-arg Methods causes Heap Pollution, not all the var-arg methods. If we know that our method won't cause Heap Pollution, then we can suppress compiler warnings with @SafeVarargs annotation.

```
1) import java.util.*;
2) public class Test
3) {
4)     public static void main(String[] args)
5)     {
6)         List<String> l1= Arrays.asList("A","B");
7)         List<String> l2= Arrays.asList("C","D");
8)         m1(l1,l2);
9)     }
10)    @SafeVarargs
11)    public static void m1(List<String>... l)
12)    {
13)        for(List<String> l1: l)
14)        {
15)            System.out.println(l1);
16)        }
17)    }
18) }
```

Output:

[A, B]
[C, D]

In the program, inside m1() method we are not performing any reassignments. Hence there is no chance of Heap Pollution Problem. Hence we can suppress Compiler generated warnings with @SafeVarargs annotation.

Note: At compile time observe the difference with and without SafeVarargs Annotation.

Java 9 Enhancements to @SafeVarargs Annotation:

@SafeVarargs Annotation introduced in Java 7.

Until Java 8, this annotation is applicable only for static methods, final methods and constructors. But from Java 9 onwards, we can also use for private instance methods also.

```
1) import java.util.*;
2) public class Test
3) {
4)     @SafeVarargs //valid
5)     public Test(List<String>... l)
6)     {
7)     }
8)     @SafeVarargs //valid
9)     public static void m1(List<String>... l)
10)    {
11)    }
12)    }
13)    @SafeVarargs //valid
14)    public final void m2(List<String>... l)
15)    {
16)
17)    }
18)    @SafeVarargs //valid in Java 9 but not in Java 8
19)    private void m3(List<String>... l)
20)    {
21)    }
22) }
```

javac -source 1.8 Test.java

error: Invalid SafeVarargs annotation. Instance method m3(List<String>...) is not final.

private void m3(List<String>... l)

^

javac -source 1.9 Test.java

We won't get any compile time error.

FAQs:

Q1. For which purpose we can use @SafeVarargs annotation?

Q2. What is Heap Pollution ?

TechieFolio.com