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.

```
public class Test
{
  public static void m1(int... x)
  {
    System.out.println("var-arg method");
  }
  public static void main(String[] args)
  {
    m1();
    m1(10);
    m1(10,20,30);
  }
}
```

Output

```
var-arg method
var-arg method
var-arg method
```

Internally var-arg parameter will be converted into array.

```
public class Test
{
  public static void sum(int... x)
  {
    int total=0;
    for(int x1 : x)
    {
      total=total+x1;
```

```
      }
      System.out.println("The Sum:"+ total);
   }
   public static void main(String[] args)
   {
      sum();
      sum(10);
      sum(10,20,30);
   }
}
```

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 ClasssCastException. This problem is called Heap Pollution.
In our code, if there is any chance of heap pollution then compiler will generate warnings.

```
import java.util.*;
public class Test
{
   public static void main(String[] args)
   {
      List<String> l1= Arrays.asList("A","B");
      List<String> l2= Arrays.asList("C","D");
      m1(l1,l2);
   }
   public static void m1(List<String>... l)//argument will become List<String>[]
   {
      Object[] a = l;// we can assign List[] to Object[]
      a[0]=Arrays.asList(10,20);
      String name=(String)l[0].get(0);//String type pointing to Integer type
      System.out.println(name);
   }
}
```

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.

```java
import java.util.*;
public class Test
{
   public static void main(String[] args)
   {
      List<String> l1= Arrays.asList("A","B");
      List<String> l2= Arrays.asList("C","D");
      m1(l1,l2);
   }
   @SafeVarargs
   public static void m1(List<String>... l)
   {
      for(List<String> l1: l)
      {
         System.out.println(l1);
      }
   }
}
```

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.
Unitl 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.

```java
import java.util.*;
public class Test
{
    @SafeVarargs //valid
    public Test(List<String>... l)
    {
    }
    @SafeVarargs //valid
    public static void m1(List<String>... l)
    {

    }
    @SafeVarargs //valid
    public final void m2(List<String>... l)
    {

    }
    @SafeVarargs //valid in Java 9 but not in Java 8
    private void m3(List<String>... l)
    {
    }
}
```

```
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 ?