Varargs in Java

Last modified: June 7, 2020

by baeldung

Java +

Core Java

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE

1. Introduction

Varargs were introduced in *Java 5* and provide a short-hand for methods that support an arbitrary number of parameters of one type.

In this article, we'll see how we can use this core Java feature.

2. Before *Varargs*

Before Java 5, whenever we wanted to pass an arbitrary number of arguments, we had to pass all arguments in an array or implement N methods (one for each additional parameter):

```
public String format() { ... }

public String format(String value) { ... }

public String format(String val1, String val2) { ... }
```

3. Use of *Varargs*

Varargs help us avoid writing boilerplate code by introducing the new syntax that can handle an arbitrary number of parameters automatically – using an array under the hood.

We can define them using a standard type declaration, followed by an ellipsis:

```
public String formatWithVarArgs(String... values) {
    // ...
}
```

And now, we can call our method with an arbitrary number of arguments, like:

```
formatWithVarArgs();

formatWithVarArgs("a", "b", "c", "d");
```

As mentioned earlier, ***varargs* are arrays so we need to work with them just like we'd work with a normal array.**

4. Rules

Varargs are straightforward to use. But there're a few rules we have to keep in mind:

- Each method can only have one *varargs* parameter
- The *varargs* argument must be the last parameter

5. Heap Pollution

Using *varargs* can lead to so-called Heap Pollution. To better understand the heap pollution, consider this *varargs* method:

```
static String firstOfFirst(List<String>... strings) {
```

```
    List<Integer> ints = Collections.singletonList(42);
    Object[] objects = strings;
    objects[0] = ints; // Heap pollution

    return strings[0].get(0); // ClassCastException
}
```

If we call this strange method in a test:

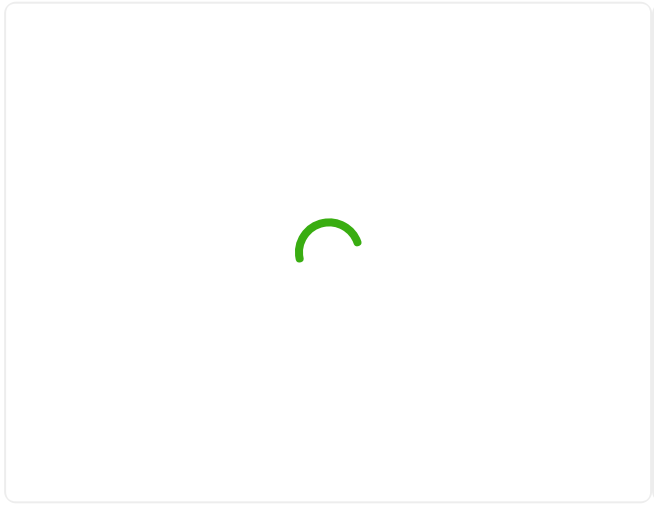
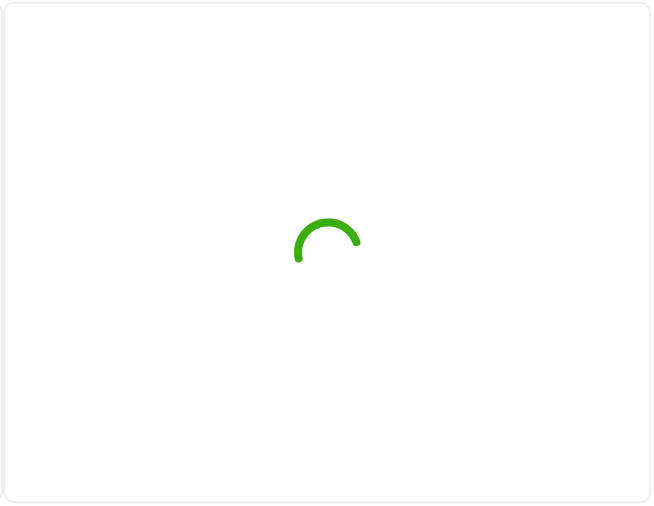
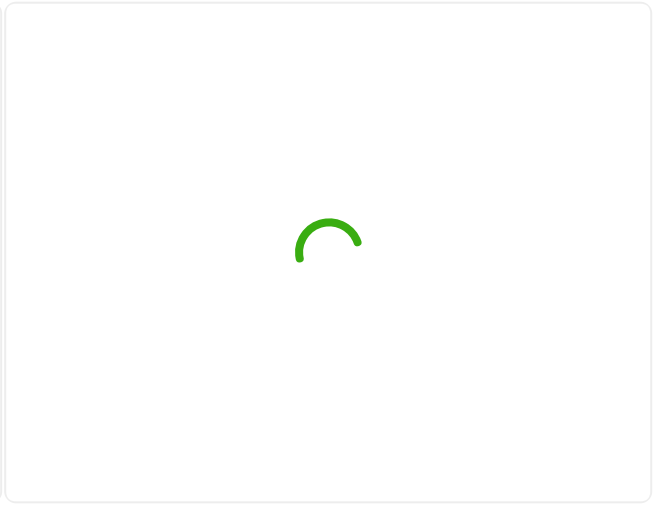
```
String one = firstOfFirst(Arrays.asList("one", "two"), Collections.emptyList());

assertEquals("one", one);
```

We would get a *ClassCastException* even though we didn't even use any explicit type casts here:

```
java.lang.ClassCastException: class java.lang.Integer cannot be cast to class java.lang.String
```

5.1. Safe Usage

Every time we use *varargs*, the Java compiler creates an array to hold the given parameters. In this case, the compiler creates an array with generic type components to hold the arguments.

When we use *varargs* with generic types, as there's a potential risk of a fatal runtime exception, the Java compiler warns us about a possible unsafe *varargs* usage:

```
warning: [varargs] Possible heap pollution from parameterized vararg type T
```

The *varargs* usage is safe if and only if:

- We don't store anything in the implicitly created array. In this example, we did store a *List<Integer>* in that array
- We don't let a reference to the generated array escape the method (more on this later)

If we are sure that the method itself does safely use the varargs, we can use *@SafeVarargs* to suppress the warning.

Put simply, the *varargs* usage is safe if we use them to transfer a variable number of arguments from the caller to the method and nothing more!

5.2. Escaping *Varargs* Reference

Let's consider another unsafe usage of *varargs*:

```
static <T> T[] toArray(T... arguments) {
    return arguments;
}
```

At first, it might seem that the *toArray* method is completely harmless. **However, because it let the varargs array escape to the caller, it violates the second rule of safe *varargs***.

To see how this method can be dangerous, let's use it in another method:

```
static <T> T[] returnAsIs(T a, T b) {
    return toArray(a, b);
}
```

Then if we call this method:

```
String[] args = returnAsIs("One", "Two");
```

We would, again, get a *ClassCastException*. Here's what happens when we call the *returnAsIs* method:

- To pass *a* and *b* to the *toArray* method, Java needs to create an array
- Since the *Object[]* can hold items of any type, the compiler creates one
- The *toArray* method returns the given *Object[]* to the caller
- Since the call site expects a *String[]*, the compiler tries to cast the *Object[]* to the expected *String[]*, hence the *ClassCastException*

For a more detailed discussion on heap pollution, it's highly recommended to read item 32 of Effective Java by Joshua Bloch.

6. Conclusion

Varargs can make a lot of boilerplate go away in Java.

And, thanks to their implicit *autoboxing* to and from *Array,* they play a role in future-proofing our code.

As always, all code examples from this article can are available in our GitHub repository.

>> CHECK OUT THE COURSE

1 COMMENT

Oldest

JoeHx 2 years ago

I remember when I first discovered varargs I used them EVERYWHERE. Ahh, such innocence.

Comments are closed on this article!