(/)

The StackOverflowError in Java

Last updated: January 8, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Reviewed by: Predrag Marić (https://www.baeldung.com/editor/predrag-author)

Core Java (https://www.baeldung.com/category/java/core-java)

Exception (https://www.baeldung.com/tag/exception)

Java Stack Trace (https://www.baeldung.com/tag/java-stack-trace)

StackOverflowError can be annoying for Java developers, as it's one of the most common runtime errors we can encounter.

In this article, we'll see how this error can occur by looking at a variety of code examples as well as how we can deal with it.

2. Stack Frames and How *StackOverflowError* Occurs

Let's start with the basics. **When a method is called, a new stack frame gets created on the call stack (/cs/call-stack).** This stack frame holds parameters of the invoked method, its local variables and the return address of the method i.e. the point from which the method execution should continue after the invoked method has returned.

The creation of stack frames will continue until it reaches the end of method invocations found inside nested methods.

During this process, if JVM encounters a situation where there is no space for a new stack frame to be created, it will throw a *StackOverflowError*.

The most common cause for the JVM to encounter this situation is **unterminated/infinite recursion** – the Javadoc description for *StackOverflowError* mentions that the error is thrown as a result of too deep recursion in a particular code snippet.

However, recursion is not the only cause for this error. It can also happen in a situation where an application keeps **calling methods from within methods until the stack is exhausted**. This is a rare case since no developer would intentionally follow bad coding practices. Another rare cause is **having a vast number of local variables inside a method**.

The *StackOverflowError* can also be thrown when an application is designed to have **cyclic relationships between classes**. In this situation, the constructors of each other are getting called repetitively which causes this error to be thrown. This can also be considered as a form of recursion.

Another interesting scenario that causes this error is if a **class is being instantiated within the same class as an instance variable of that class**. This will cause the constructor of the same class to be called again and again (recursively) which eventually results in a *StackOverflowError.*

In the next section, we'll look at some code examples that demonstrate these scenarios.

3. *StackOverflowError* in Action

In the example shown below, a *StackOverflowError* will be thrown due to unintended recursion, where the developer has forgotten to specify a termination condition for the recursive behavior:

```
public class UnintendedInfiniteRecursion {
    public int calculateFactorial(int number) {
        return number * calculateFactorial(number - 1);
    }
}
```

Here, the error is thrown on all occasions for any value passed into the method:

```
public class UnintendedInfiniteRecursionManualTest {
    @Test(expected = StackOverflowError.class)
    public void givenPositiveIntNoOne_whenCalFact_thenThrowsException() {
        int numToCalcFactorial= 1;
        UnintendedInfiniteRecursion uir
          = new UnintendedInfiniteRecursion();

        uir.calculateFactorial(numToCalcFactorial);
    }

    @Test(expected = StackOverflowError.class)
    public void givenPositiveIntGtOne_whenCalcFact_thenThrowsException() {
        int numToCalcFactorial= 2;
        UnintendedInfiniteRecursion uir
          = new UnintendedInfiniteRecursion();

        uir.calculateFactorial(numToCalcFactorial);
    }

    @Test(expected = StackOverflowError.class)
    public void givenNegativeInt_whenCalcFact_thenThrowsException() {
        int numToCalcFactorial= -1;
        UnintendedInfiniteRecursion uir
          = new UnintendedInfiniteRecursion();

        uir.calculateFactorial(numToCalcFactorial);
    }
}
```

However, in the next example a termination condition is specified but is never being met if a value of *-1* is passed to the *calculateFactorial()* method, which causes unterminated/infinite recursion:

```
public class InfiniteRecursionWithTerminationCondition {
    public int calculateFactorial(int number) {
       return number == 1 ? 1 : number * calculateFactorial(number - 1);
    }
}
```

This set of tests demonstrates this scenario:

```
public class InfiniteRecursionWithTerminationConditionManualTest {
    @Test
    public void givenPositiveIntNoOne_whenCalcFact_thenCorrectlyCalc() {
        int numToCalcFactorial = 1;
        InfiniteRecursionWithTerminationCondition irtc
          = new InfiniteRecursionWithTerminationCondition();

        assertEquals(1, irtc.calculateFactorial(numToCalcFactorial));
    }

    @Test
    public void givenPositiveIntGtOne_whenCalcFact_thenCorrectlyCalc() {
        int numToCalcFactorial = 5;
        InfiniteRecursionWithTerminationCondition irtc
          = new InfiniteRecursionWithTerminationCondition();

        assertEquals(120, irtc.calculateFactorial(numToCalcFactorial));
    }

    @Test(expected = StackOverflowError.class)
    public void givenNegativeInt_whenCalcFact_thenThrowsException() {
        int numToCalcFactorial = -1;
        InfiniteRecursionWithTerminationCondition irtc
          = new InfiniteRecursionWithTerminationCondition();

        irtc.calculateFactorial(numToCalcFactorial);
    }
}
```

In this particular case, the error could have been completely avoided if the termination condition was simply put as:

```
public class RecursionWithCorrectTerminationCondition {
    public int calculateFactorial(int number) {
        return number <= 1 ? 1 : number * calculateFactorial(number - 1);
    }
}
```

Here's the test that shows this scenario in practice:

```
public class RecursionWithCorrectTerminationConditionManualTest {
    @Test
    public void givenNegativeInt_whenCalcFact_thenCorrectlyCalc() {
        int numToCalcFactorial = -1;
        RecursionWithCorrectTerminationCondition rctc
          = new RecursionWithCorrectTerminationCondition();

        assertEquals(1, rctc.calculateFactorial(numToCalcFactorial));
    }
}
```

Now let's look at a scenario where the *StackOverflowError* happens as a result of cyclic relationships between classes. Let's consider *ClassOne* and *ClassTwo*, which instantiate each other inside their constructors causing a cyclic relationship:

```
public class ClassOne {
    private int oneValue;
    private ClassTwo clsTwoInstance = null;

    public ClassOne() {
        oneValue = 0;
        clsTwoInstance = new ClassTwo();
    }

    public ClassOne(int oneValue, ClassTwo clsTwoInstance) {
        this.oneValue = oneValue;
        this.clsTwoInstance = clsTwoInstance;
    }
}
```

```
public class ClassTwo {
    private int twoValue;
    private ClassOne clsOneInstance = null;

    public ClassTwo() {
        twoValue = 10;
        clsOneInstance = new ClassOne();
    }

    public ClassTwo(int twoValue, ClassOne clsOneInstance) {
        this.twoValue = twoValue;
        this.clsOneInstance = clsOneInstance;
    }
}
```

Now let's say that we try to instantiate *ClassOne* as seen in this test:

```
public class CyclicDependencyManualTest {
    @Test(expected = StackOverflowError.class)
    public void whenInstanciatingClassOne_thenThrowsException() {
        ClassOne obj = new ClassOne();
    }
}
```

This ends up with a *StackOverflowError* since the constructor of *ClassOne* is instantiating *ClassTwo*, and the constructor of *ClassTwo* again is instantiating *ClassOne*. And this repeatedly happens until it overflows the stack.

Next, we will look at what happens when a class is being instantiated within the same class as an instance variable of that class.

As seen in the next example, *AccountHolder* instantiates itself as an instance variable *jointAccountHolder*.

```
public class AccountHolder {
    private String firstName;
    private String lastName;

    AccountHolder jointAccountHolder = new AccountHolder();
}
```

When the *AccountHolder* class is instantiated, a *StackOverflowError* is thrown due to the recursive calling of the constructor as seen in this test:

```
public class AccountHolderManualTest {
    @Test(expected = StackOverflowError.class)
    public void whenInstanciatingAccountHolder_thenThrowsException() {
        AccountHolder holder = new AccountHolder();
    }
}
```

4. Dealing With *StackOverflowError*

The best thing to do when a *StackOverflowError* is encountered is to inspect the stack trace cautiously to identify the repeating pattern of line numbers. This will enable us to locate the code that has problematic recursion.

(/)

Let's examine a few stack traces caused by the code examples we saw earlier.

This stack trace is produced by *InfiniteRecursionWithTerminationConditionManualTest* if we omit the *expected* exception declaration:

```
java.lang.StackOverflowError

 at c.b.s.InfiniteRecursionWithTerminationCondition
  .calculateFactorial(InfiniteRecursionWithTerminationCondition.java:5)
 at c.b.s.InfiniteRecursionWithTerminationCondition
  .calculateFactorial(InfiniteRecursionWithTerminationCondition.java:5)
 at c.b.s.InfiniteRecursionWithTerminationCondition
  .calculateFactorial(InfiniteRecursionWithTerminationCondition.java:5)
 at c.b.s.InfiniteRecursionWithTerminationCondition
  .calculateFactorial(InfiniteRecursionWithTerminationCondition.java:5)
```

Here, line number 5 can be seen repeating. This is where the recursive call is being done. Now it's just a matter of examining the code to see if the recursion is done in a correct manner.

Here is the stack trace we get by executing *CyclicDependancyManualTest* (again, without *expected* exception):

```
java.lang.StackOverflowError
  at c.b.s.ClassTwo.<init>(ClassTwo.java:9)
  at c.b.s.ClassOne.<init>(ClassOne.java:9)
  at c.b.s.ClassTwo.<init>(ClassTwo.java:9)
  at c.b.s.ClassOne.<init>(ClassOne.java:9)
```

This stack trace shows the line numbers that cause the problem in the two classes that are in a cyclic relationship. Line number 9 of *ClassTwo* and line number 9 of the *ClassOne* point to the location inside the constructor where it tries to instantiate the other class.

Once the code is being thoroughly inspected and if none of the following (or any other code logic error) is the cause of the error:

- Incorrectly implemented recursion (i.e. with no termination condition)
- Cyclic dependency between classes
- Instantiating a class within the same class as an instance variable of that class

It would be a good idea to try and increase the stack size. Depending on the JVM installed, the default stack size could vary.

The *-Xss* flag can be used to increase the size of the stack, either from the project's configuration or the command line.

5. Conclusion

In this article, we took a closer look at the *StackOverflowError* including how Java code can cause it and how we can diagnose and fix it.

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member (/members/)**, start learning and coding on the project.

BAELDUNG ALL ACCESS (/COURSES/ALL-ACCESS)

BAELDUNG ALL TEAM ACCESS (/COURSES/ALL-ACCESS-TEAM)

LOGIN COURSE PLATFORM (HTTPS://WWW.BAELDUNG.COM/MEMBERS/ACCOUNT)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)

LEARN SPRING BOOT SERIES (/SPRING-BOOT)

SPRING TUTORIAL (/SPRING-TUTORIAL)

GET STARTED WITH JAVA (/GET-STARTED-WITH-JAVA-SERIES)

ALL ABOUT STRING IN JAVA (/JAVA-STRING)

SECURITY WITH SPRING (/SECURITY-SPRING)

JAVA COLLECTIONS (/JAVA-COLLECTIONS)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE FULL ARCHIVE (/FULL_ARCHIVE)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS/)

PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)

EBOOKS (/LIBRARY/)

FAQ (/LIBRARY/FAQ)


BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)

PRIVACY MANAGER