
How to resolve a java.lang.NullPointerException

Tony Morris <http://tmorris.net/>

Copyright © 2006 Tony Morris

Abstract

Every person who is learning how to use the Java 2 Programming Language will encounter a `java.lang.NullPointerException` during the execution of their application at some point. While this error message may appear to be cryptic to the learner, it does have a very concise meaning and there is also a systematic approach towards resolving this problem. There are also perceived solutions that aren't actually solutions. That is, a learner might have written some fix that is problematic and actually hides the original problem, not fixes it.

Introduction

This article intends to document the systematic approach towards resolution of a `java.lang.NullPointerException`. It also intends to document approaches which are flawed. This document is not intended to answer the question of "why?", since that investigation is deferred as a reader exercise. That is, an explanation of why you are observing this behaviour is not given in this document, only the solution. This kind of information is best left to an appropriate tutorial such as Sun's "Object Basics and Simple Data Objects" [<http://java.sun.com/docs/books/tutorial/java/data/index.html>]. Like all exception types, a good understanding can be gained from the J2SE API Specification for `java.lang.NullPointerException` [<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/NullPointerException.html>].

A `NullPointerException` can only be caused by one of three things:

1. Dereferencing of a null object reference.
2. Explicitly throwing a `NullPointerException` using the 'throw' keyword.
3. An unforeseen internal JVM software defect (highly unlikely).

This list is complete despite some of the other reasons that some learners erroneously come up with. It is imperative that this is understood before moving on and that all myths regarding other causes are dispelled.

Of the three given causes, a reasonable estimate of the actual cause of problems in practice is given as:

1. Dereferencing of a null object reference - 95% of the time.
2. Explicitly throwing a `NullPointerException` using the 'throw' keyword - 4.99999999% of the time.
3. An unforeseen internal JVM software defect (highly unlikely) - 0.00000001% of the time.

It only makes sense now to put a lot of focus on the first given cause, a little on the second and none on the third.

Dereferencing a null object reference

The concept of an object reference can often confuse learners and it is this confusion that can sometimes lead to the misunderstandings when a `NullPointerException` is encountered. Without going into a detailed explanation, here are some examples:

```
// A null object reference called 'o'.
Object o = null;

// A null object reference called 's'.
String s = null;

// An object reference called 'i' referring to an object (not null).
Integer i = new Integer(8);

// Synchronizing on a null reference.
Object o = null;
synchronized(o){}

// Throwing a null reference.
Throwable t = null;
throw t;

// Now 'i' is a null object reference.
i = null;

// An object reference called 'd' referring to an object (not null).
// Each element of the array object is a null object reference.
Double[] d = new Double[10];

// A null object reference called 'array'.
Object[] array = null;
```

Class fields members that are object reference types are implicitly initialized to null at object creation time (when the 'new' operator is used) as are arrays of object reference types. An object reference type is any type that is not in the set {byte, short, int, long, float, double, char, boolean}. Note that all arrays and String are object reference types.

Given the many (infinite) number of object reference types, let's discover how they are dereferenced. An object reference can be dereferenced in only one of six possible ways:

1. Following the object reference name with a dot '.' and then calling on one of it's members or methods.

```
String s = null;
// following the reference name 's' with a dot is
char c = s.charAt(0);

// Accessing a static member, in this case the 'valueOf'
// method does NOT dereference s.
// Accessing static members through object reference
// is considered poor form.
// Use the class name instead, but be aware that
// does not throw a NullPointerException.
int j = s.valueOf(s);
```

2. Using a reference as the second argument to the foreach loop (applies to J2SE 1.5 only).

```
java.util.List<String> strings = null;
// throws a NullPointerException because the 'strings'
```

```
// is dereferenced implicitly.  
for(String s : strings){}
```

3. Unboxing a reference type to a primitive type (applies to J2SE 1.5 only).

```
Long x = null;  
// throws a NullPointerException because the 'x' :  
// is dereferenced implicitly because it is unboxed.  
long y = x;
```

4. Synchronizing on a reference to null.

```
String s = null;  
// throws a NullPointerException because the 's' :  
// is dereferenced implicitly.  
synchronized(s)  
{  
  
}
```

5. Throwing a reference to null.

```
Throwable t = null;  
// throws a NullPointerException because the 't' :  
// is dereferenced implicitly.  
throw t;
```

6. If the object reference is an array type only, following the object reference name with an open square bracket '[' to index the array.

```
char[] c = new char[42];  
// throws a NullPointerException because the 'c' :  
// is dereferenced by indexing the array.  
char x = c[0];
```

Here are some more examples of dereferencing a null object reference:

```
// A null object reference called 's'  
String s = null;  
// Dereferencing the object reference 's' by calling it's length.  
int len = s.length();
```

```
// A null object reference called 'c'.  
char[] c = null;  
// Dereferencing the object reference 'c' by attempting to index.  
char x = c[3];
```

```
// An object reference referring to an object called 'o'.  
// All elements of the array are initialized to null.  
Object[] o = new Object[10];
```

```
// Dereferencing a null object reference (the fourth element of  
// the array referred to by 'o').  
String s = o[4].toString();  
  
// A complete source file: Broken.java  
public class Broken  
{  
    // An object reference type, so implicitly initialized to null  
    private Integer i;  
  
    void m()  
    {  
        // 'i' has never been given a value so it is still  
        // an object reference to null.  
        int x = i.intValue();  
    }  
}
```

Explicitly throwing a NullPointerException using the 'throw' keyword

This less likely cause of a NullPointerException is less subtle than the aforementioned cause. A Java application may explicitly throw a NullPointerException using the 'throw' keyword - here are some examples:

```
// construct and throw a NullPointerException in one line.  
throw new NullPointerException();  
  
// construct a NullPointerException  
NullPointerException e = new NullPointerException  
    ("here is a message that you will see when it is thrown");  
// throw the NullPointerException  
throw e;
```

Summary of the cause of a NullPointerException

Given these explanations, the complete set of causes can be summarised as follows:

- Dereferencing of a null object reference using a dot '.'.
- Using a reference as the second argument to the foreach loop (applies to J2SE 1.5 only).
- Unboxing a reference type to a primitive type (applies to J2SE 1.5 only).
- Synchronizing on a reference to null using the 'synchronized' keyword.
- Throwing a reference to null using the 'throw' keyword.
- Dereferencing of a null object reference that is an array type using an open square bracket '['.
- Explicitly throwing a NullPointerException using the 'throw' keyword.
- Explicitly throwing a NullPointerException using the 'throw' keyword.
- An unforeseen internal JVM software defect (highly unlikely).

So when you are looking for the cause of the NullPointerException this 'checklist' can be exhausted.

What is the cause of my NullPointerException?

To actually diagnose the cause requires analysis of the stack trace. The stack trace is the output that you see, usually on the standard error stream (the console). Here is an example program that will throw a NullPointerException followed by the corresponding stack trace:

```
// Broken.java
public class Broken
{
    public static void main(String[] args)
    {
        method();
    }

    private static void method()
    {
        String s = null;
        int len = s.length();
    }
}
```

Here is the stack after execution:

```
Exception in thread "main" java.lang.NullPointerException
    at Broken.method(Broken.java:11)
    at Broken.main(Broken.java:5)
```

Analysis of the stack trace can be quite tricky, but it is still systematic.

1. Find the first line in the stack trace that refers to code that you wrote. In this case, Broken.java:11 is our code and is the first in the stack trace.
2. Go to this line of source code. Line 11 of Broken.java looks like this: `int len = s.length();`
3. Using the checklist of potential causes, find the cause:
 - a. Dereferencing of a null object reference using a dot '.' This is a possible cause, since we dereference 's' with a dot - it may be null.
 - b. Using a reference as the second argument to the foreach loop (applies to J2SE 1.5 only). This is not a possible cause because we do not use a foreach loop on line 11.
 - c. Unboxing a reference type to a primitive type (applies to J2SE 1.5 only). This is not a possible cause because we do not perform an operation on a reference type that results in unboxing to a primitive type on line 11, however, note that if the String.length() method returned a reference type, and not type int, it is certainly a possible cause.
 - d. Synchronizing on a reference to null using the synchronized keyword. This is not a possible cause because we never use the synchronized keyword.
 - e. Throwing a reference to null using the throw keyword. This is not a possible cause because we never use the throw keyword.
 - f. Dereferencing of a null object reference of an array type using an open square bracket '['. This is not a possible cause because we do not have an array type on line 11.

- g. Explicitly throwing a `NullPointerException` using the 'throw' keyword. This is not a possible cause because we never use the throw keyword.
 - h. An unforeseen internal JVM software defect (highly unlikely). This is not a possible cause since it is too unlikely to ever occur.
4. Since there is only one possible cause, it **MUST** be this cause - the object reference 's' is referring to null. This can be proven by preceding line 11 with the following code then running the application again:

```
if(s == null)
{
    System.err.println("Yes, s is null and I'm ab
    System.exit(1);
}
```

Only if the object reference called 's' is referring to null (which we know it is) will you see the above message.

Sometimes the stack trace will refer to code that you are calling and therefore, it is out of your control to change it. For example, some core APIs will throw a `NullPointerException` when you pass a null argument to a method. This can be resolved by reading and adhering to the contract (API documentation) for the method that you are calling on the line that was discovered at step 2 above. If this fails, the code that you are calling should be considered defective. In this case, consult the author(s) of the code for resolution.

How do I stop the `NullPointerException`?

This question is not very easy to answer since it is very context-dependant. It may be that you don't want to dereference the object reference after all; or it may be that you want to assign the String reference that you are dereferencing to a String with the value, "Tomorrow is Friday.". It is up to the context to decide the resolution, suffice to say that dereferencing a null object reference should always be avoided. A possible solution to the previous example may be:

```
// Broken.java
// Fixed now
public class Broken
{
    public static void main(String[] args)
    {
        method();
    }

    private static void method()
    {
        String s = "Hello to all";
        int len = s.length();
    }
}
```

If I declare to 'catch' the `NullPointerException` the stack trace goes away - is it fixed?

Certainly not - this will only cause more problems. Any source code that declares to catch `java.lang.NullPointerException` should be considered defective. An explanation of why this practice

is defective is left as a reader exercise. Some software vendors who have distributed defective APIs that erroneously throw a `NullPointerException` stipulate a workaround to their problem by declaring to catch `NullPointerException`. This is perhaps the only justified reason for performing this act, and it should be clearly documented what the intention is (to workaround an existing problem) for others who might maintain your code.