1 Why do NullPointerExceptions happen?

As you as you start playing with pointers to create dynamic data structures you will soon receive a `NullPointerException`. Although we will learn more about exceptions in the near future, a short explanation may be helpful now.

Java throws a `NullPointerException` when it tries to follow a pointer and it cannot. This happens if the pointer is `null`, i.e. it is not pointing anywhere. If it is not pointing anywhere, it cannot be followed to any object.

Let's see a small example. Imagine that you have two Strings in your code:

```
String str1 = "This is a test string.";
String str2 = null;
println "First string's length: " + str1.length();
println "Second string's length: " + str2.length();
```

This code will compile without problems: it is syntactically correct. However, when you execute the code you will get a `NullPointerException`:

```
First string's length: 22
Exception in thread "main" java.lang.NullPointerException
     at MyClass.main(MyClass.java:15)
```

This is because Java is trying to follow the pointer of `str2` to execute the method `length()`, but it is findind the pointer is not pointing anywhere (Figure 1).

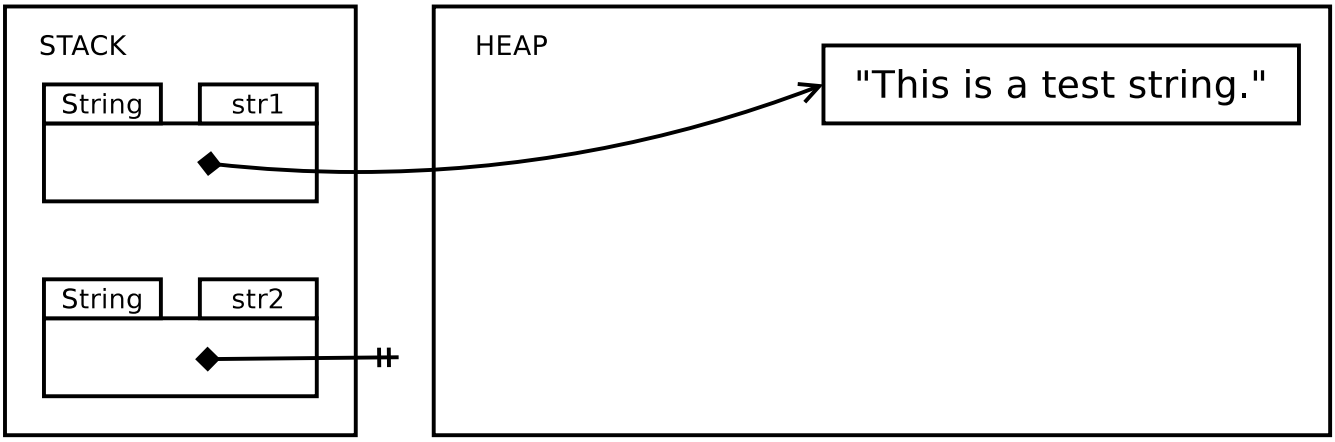

Figure 1: If Java tries to follow a null pointer it throws a NullPointerException. In the example, this happens with String str2.

1.1 Is this a problem only for methods?

We have said that member fields should be private, so at first sight it may seem that a `NullPointerException` can only be the result of calling a method on a null pointer, as in the example above. Admittedly, this is the most common source of `NullPointerException`, but not the only one.

Private member fields cannot be accessed from outside the class, but they can be accessed in the class. You must keep in mind that the code of a class is the same code for all objects of that class, so it may be accessing member fields of more than one object.

To illustrate the point, let's look at a possible implementation of method `equals` of class `String`:

```
/**
 * Possible (non-real) implementation of String.equals().
 * Assumes class String contains an array of chars
 * called charArray.
 *
 * The method is used like "str1.equals(str2)".
```

```
 */
public boolean equals(String other) {
   if (this.length() != other.length()) {
      return false;
   }
   for (int i = 0; i < charArray.length; i++) {
      if (this.charArray[i] != other.charArray[i]) {
         return false;
      }
   }
   return true;
}
```

This method seems quite clear. First, the lengths of the string where the method is called ("this", e.g. str1) and the other string (parameter "other", e.g. str2) are compared in size. If they have different lengths they are obviously different, so `false` is returned. If they have the same length, they are compared character by character; as soon as a character is different, `false` is returned. If the loop reached the end, that means that all characters are equal, and `true` is returned.

Simple and clear, and it works... or not. What happens if I compare with a string where `charArray` is `null`? I will get a verb+NullPointerException+ when I try to access `other.charArray[i]`. This is a private field and it is accessed from inside the class' code... from another object.

1.2 What can I do about it?

In simple programs, like the ones we have done so far, it is easy to keep track of all pointers to make sure they are not `null`. Real programs, on the other hand, have hundreds of classes containing thousands of pointers. No human being can keep track of hundreds or thousands of pieces of data: that is what computers where invented for[1].

But there is something that programmers *can* do, and it checking whether a pointer is null before following it. The method above would be safer if written as follows:

```
/**
 * Possible (non-real) implementation of String.equals().
 * Assumes class String contains an array of chars
 * called charArray.
 *
 * The method is used like "str1.equals(str2)".
 */
public boolean equals(String other) {
   if (other == null) {
      return false;
   }
   if (this.charArray == null || other.charArray == null) {
      return false;
   }
   if (this.length() != other.length()) {
      return false;
   }
   for (int i = 0; i < charArray.length; i++) {
      if (this.charArray[i] != other.charArray[i]) {
         return false;
      }
   }
   return true;
}
```

[1] Andrew Booth, founder of the Department of Computer Science at Birkbeck, invented one of the first computers to relieve his fellow crystallographers of the tyranny of mathematics, referring to how they had to perform heavy calculations involving a lot of numbers by hand.

As you can see, the first thing this method does is checking that the argument is not null ("this" string can never be null, otherwise the method would not be in execution); if it is, it cannot be equal to "this" string so `false` is returned. The second thing is checking that the arrays are not null; if any of they are, there is no comparison possible and `false` is returned. From them on, assured that all pointers are non-null, the method continues normally as we have seen above.

1.3 Conclusion

Null pointers are not pointing anywhere. Trying to access a field or a method they are pointing to will result in a `NullPointerException`. We must be careful to prevent this from happening in our code.

Sometimes it is impossible to make sure whether a pointer is null or not, e.g. when the program is very complex and has hundreds of classes, or when we reuse code from another programmer. In those cases, we must protect ourselves by checking that pointers are not `null` before following them.