W

Avoiding Null Checks in Java 8

March 15, 2015

How to prevent the famous `NullPointerException` in Java? This is one of the key questions every Java beginner will ask sooner or later. But also intermediate and expert programmers get around this error every now and then. It's by far the most prevalent kind of error in Java and many other programming languages as well.

Tony Hoare, the inventor of the null reference apologized in 2009 and denotes this kind of errors as his **billion-dollar mistake**.

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

Anyways, we have to deal with it. So what can we do to prevent NullPointerExceptions at all? Well, the obvious answer is to add null checks all around the place. Since null checks are kinda cumbersome and painful many languages add special syntax for handling null checks via null coalescing operators - also known as *elvis operator* in languages like Groovy or Kotlin.

Unfortunately Java doesn't provide such a syntactic sugar. But luckily things get better in Java Version 8. This post describes a couple of techniques how to prevent writing needless null checks by utilizing new features of Java 8 like lambda expressions.

Improving Null Safety in Java 8

I've already shown in another post how we can utilize the `Optional` type of Java 8 to prevent null checks. Here's the example code from the original post.

Assuming we have a hierarchical class structure like this:

```
class Outer {
    Nested nested;
    Nested getNested() {
        return nested;
    }
}
class Nested {
    Inner inner;
    Inner getInner() {
        return inner;
    }
}
class Inner {
    String foo;
    String getFoo() {
        return foo;
    }
}
```

Resolving a deep nested path in this structure can be kinda awkward. We have to write a bunch of null checks to make sure not to raise a `NullPointerException`:

```
Outer outer = new Outer();
if (outer != null && outer.nested != null && outer.nested.inner != null) {
    System.out.println(outer.nested.inner.foo);
}
```

We can get rid of all those null checks by utilizing the Java 8 `Optional` type. The method `map` accepts a lambda expression of type `Function` and automatically wraps each function result into an `Optional`. That enables us to pipe multiple `map` operations in a row. Null checks are automatically handled under the hood.

```
Optional.of(new Outer())
    .map(Outer::getNested)
    .map(Nested::getInner)
    .map(Inner::getFoo)
    .ifPresent(System.out::println);
```

An alternative way to achieve the same behavior is by utilizing a supplier function to resolve the nested path:

```
Outer obj = new Outer();
resolve(() -> obj.getNested().getInner().getFoo());
    .ifPresent(System.out::println);
```

Calling `obj.getNested().getInner().getFoo()` might throw a `NullPointerException`. In this case the exception will be caught and the method returns `Optional.empty()`.

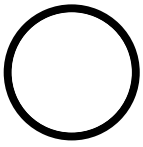
```
public static <T> Optional<T> resolve(Supplier<T> resolver) {
    try {
        T result = resolver.get();
        return Optional.ofNullable(result);
    }
    catch (NullPointerException e) {
        return Optional.empty();
    }
}
```

Please keep in mind that both solutions are probably not as performant as traditional null checks. In most cases that shouldn't be much of an issue.

As usual the above code samples are hosted on GitHub.

Happy coding!

> *UPDATE: I've updated the code samples thanks to a hint from Zukhramm on Reddit.*



Benjamin is Software Engineer, Full Stack Developer at Pondus, an excited runner and table foosball player. Get in touch on Twitter, Google+ and GitHub.

Read More