Avoiding != null statements

I use `object != null` a lot to avoid `NullPointerException` .

Is there a good alternative to this?

For example:

```
if (someobject != null) {
    someobject.doCalc();
}
```

This avoids a `NullPointerException` , when it is unknown if the object is `null` or not.

Note that the accepted answer may be out of date, see https://stackoverflow.com/a/2386013/12943 for a more recent approach.

java object nullpointerexception null

edited Apr 17 at 17:15

community wiki
28 revs, 23 users 12%
Goran Martinic

103 @Shervin Encouraging nulls makes the code less understandable and less reliable. – Tom Hawtin - tackline Aug 2 '10 at 9:17

38 The Elvis operators were proposed but it looks like it won't be in Java 7. Too bad, ?. ?: and ?[] are incredible time savers. – Scott Feb 25 '11 at 2:46

62 Not using null is superior to most other suggestions here. Throw exceptions, don't return or allow nulls. BTW - 'assert' keyword is useless, because it's disabled by default. Use an always-enabled failure mechanism – ianpojman Jun 8 '12 at 19:40

12 This is one reason why I am now using Scala. In Scala everything is not nullable. If you want to allow to pass or return "nothing", then you have to exlicitly use Option[T] instead of just T als argument or return type. – Thekwasti Feb 2 '14 at 20:33

7 @thSoft Indeed, Scala's awesome `Option` type is marred by the language's unwillingness to control or disallow `null`. I mentioned this on hackernews and got downvoted and told that "nobody uses null in Scala anyway". Guess what, I'm already finding nulls in coworker-written Scala. Yes, they are "doing it wrong" and must be educated about it, but the fact remains the language's type system should protect me from this and it doesn't :(– Andres F. May 19 '14 at 19:52

58 Answers

This to me sounds like a reasonably common problem that junior to intermediate developers tend to face at some point: they either don't know or don't trust the contracts they are participating in and defensively overcheck for nulls. Additionally, when writing their own code, they tend to rely on returning nulls to indicate something thus requiring the caller to check for nulls.

To put this another way, there are two instances where null checking comes up:

1. Where null is a valid response in terms of the contract; and
2. Where it isn't a valid response.

(2) is easy. Either use `assert` statements (assertions) or allow failure (for example, NullPointerException). Assertions are a highly-underused Java feature that was added in 1.4. The syntax is:

```
assert <condition>
```

or

```
assert <condition> : <object>
```

where `<condition>` is a boolean expression and `<object>` is an object whose `toString()` method's output will be included in the error.

An `assert` statement throws an `Error` (`AssertionError`) if the condition is not true. By default, Java ignores assertions. You can enable assertions by passing the option `-ea` to the JVM. You can enable and disable assertions for individual classes and packages. This means that you can validate code with the assertions while developing and testing, and disable them in a production environment, although my testing has shown next to no performance impact from assertions.

Not using assertions in this case is OK because the code will just fail, which is what will happen if you use assertions. The only difference is that with assertions it might happen sooner, in a more-meaningful way and possibly with extra information, which may help you to figure out why it happened if you weren't expecting it.

(1) is a little harder. If you have no control over the code you're calling then you're stuck. If null is a valid response, you have to check for it.

If it's code that you do control, however (and this is often the case), then it's a different story. Avoid using nulls as a response. With methods that return collections, it's easy: return empty collections (or arrays) instead of nulls pretty much all the time.

With non-collections it might be harder. Consider this as an example: if you have these interfaces:

```
public interface Action {
  void doSomething();
}

public interface Parser {
  Action findAction(String userInput);
}
```

where Parser takes raw user input and finds something to do, perhaps if you're implementing a command line interface for something. Now you might make the contract that it returns null if there's no appropriate action. That leads the null checking you're talking about.

An alternative solution is to never return null and instead use the Null Object pattern:

```
public class MyParser implements Parser {
  private static Action DO_NOTHING = new Action() {
    public void doSomething() { /* do nothing */ }
  };

  public Action findAction(String userInput) {
    // ...
    if ( /* we can't find any actions */ ) {
      return DO_NOTHING;
    }
  }
}
```

Compare:

```
Parser parser = ParserFactory.getParser();
if (parser == null) {
  // now what?
  // this would be an example of where null isn't (or shouldn't be) a valid
response
}
Action action = parser.findAction(someInput);
if (action == null) {
  // do nothing
} else {
  action.doSomething();
}
```

to

```
ParserFactory.getParser().findAction(someInput).doSomething();
```

which is a much better design because it leads to more concise code.

That said, perhaps it is entirely appropriate for the findAction() method to throw an Exception with a meaningful error message -- especially in this case where you are relying on user input. It would be much better for the findAction method to throw an Exception than for the calling method to blow up with a simple NullPointerException with no explanation.

```
try {
    ParserFactory.getParser().findAction(someInput).doSomething();
} catch(ActionNotFoundException anfe) {
    userConsole.err(anfe.getMessage());
}
```

Or if you think the try/catch mechanism is too ugly, rather than Do Nothing your default action should provide feedback to the user.

```
public Action findAction(final String userInput) {
    /* Code to return requested Action if found */
    return new Action() {
        public void doSomething() {
            userConsole.err("Action not found: " + userInput);
        }
    }
}
```

edited Jun 26 '15 at 22:32 community wiki

13 revs, 11 users 77%
cletus

586 I disagree with your statements for the DO_NOTHING action. If the find action method cannot find an action, then returning null is the right thing to do. You've "found" an action in your code which isn't really found, which violates the principle of the method, to find a useable action. – MetroidFan2002 Nov 7 '08 at 18:36

254 I agree that null is overused in Java, especially with lists. So many apis would be better if they return an empty list/array/collection instead of null. Many times, null is used where an exception should be thrown instead. An exception should be thrown if the parser can't parse. – Laplie Anderson Nov 14 '08 at 1:53

90 The latter example here is, IIRC, the Null Object design pattern. – Steven Evers Jul 23 '09 at 4:26

58 @Cshah (and MetroidFan2002). Simple, put that into the contract and then it is clear that a non-found returned action will do nothing. If this is important information to the caller, then provide a way to discover that it was a non-found action (i.e. provide a method to check if the result was the DO_NOTHING object). Alternatively, if action should normally be found then you still should not return null but instead throw an exception specifically indicating that condition - this still results in better code. If desired, provide a separate method returning boolean to check if action exists. – Kevin Brock Mar 5 '10 at 12:25

29 Concise does not equal quality code. I'm sorry you think so. Your code hides situations where an error would be advantageous. – gshauger May 4 '11 at 4:33

If you use (or planning to use) a Java IDE like JetBrains IntelliJ IDEA, Eclipse or Netbeans or a tool like findbugs then you can use annotations to solve this problem.

Basically, you've got `@Nullable` and `@NotNull`.

You can use in method and parameters, like this:

```
@NotNull public static String helloWorld() {
    return "Hello World";
}
```

or

```
@Nullable public static String helloWorld() {
    return "Hello World";
}
```

The second example won't compile (in IntelliJ IDEA).

When you use the first `helloWorld()` function in another piece of code:

```
public static void main(String[] args)
{
    String result = helloWorld();
    if(result != null) {
        System.out.println(result);
    }
}
```

Now the IntelliJ IDEA compiler will tell you that the check is useless, since the `helloWorld()` function won't return `null`, ever.

Using parameter

```
void someMethod(@NotNull someParameter) { }
```

if you write something like:

```
someMethod(null);
```

This won't compile.

Last example using `@Nullable`

```
@Nullable iWantToDestroyEverything() { return null; }
```

Doing this

```
iWantToDestroyEverything().something();
```

And you can be sure that this won't happen. :)

It's a nice way to let the compiler check something more than it usually does and to enforce your contracts to be stronger. Unfortunately, it's not supported by all the compilers.

In IntelliJ IDEA 10.5 and on, they added support for any other `@Nullable` `@NotNull` implementations.

See blog post *More flexible and configurable @Nullable/@NotNull annotations*.

edited Apr 17 at 17:12

community wiki
8 revs, 7 users 79%
Luca Molteni

111 `@NotNull`, `@Nullable` and other nullness annotations are part of JSR 305. You can also use them to detect potential problems with tools like FindBugs. – Jacek S May 8 '10 at 16:33

28 I find it strangely annoying that the `@NotNull` & `@Nullable` interfaces live in the package `com.sun.istack.internal`. (I guess I associate com.sun with warnings about using a proprietary API.) – Jonik Jun 30 '11 at 23:33

16 code portability goes null with jetbrains.I would think twice(square) before tying down to ide level.Like Jacek S said they are part of JSR any way which I thought was JSR303 by the way. – Java Ka Baby Sep 8 '11 at 9:39

6 I really don't think that using a custom compiler is a viable solution to this problem. – Shivan Dragon Aug 19 '12 at 8:24

57 The good thing about annotations, which `@NotNull` and `@Nullable` are is that they nicely degrade when the source code is built by a system that doesn't understand them. So, in effect, the argument that the code is not portable may be invalid - if you use a system that supports and understands these annotations, you get the added benefit of stricter error checking, otherwise you get less of it but your code should still build fine, and the quality of your running program is THE SAME, because these annotations were not enforced at runtime anyway. Besides, all compilers are custom ;-) – amn Mar 20 '13 at 9:49

If null-values are not allowed

If your method is called externally, start with something like this:

```
public void method(Object object) {
  if (object == null) {
    throw new IllegalArgumentException("...");
  }
```

Then, in the rest of that method, you'll know that `object` is not null.

If it is an internal method (not part of an API), just document that it cannot be null, and that's it.

Example:

```
public String getFirst3Chars(String text) {
  return text.subString(0, 3);
}
```

However, if your method just passes the value on, and the next method passes it on etc. it could get problematic. In that case you may want to check the argument as above.

If null is allowed

This really depends. If find that I often do something like this:

```
if (object == null) {
  // something
} else {
  // something else
}
```

So I branch, and do two completely different things. There is no ugly code snippet, because I really need to do two different things depending on the data. For example, should I work on the input, or should I calculate a good default value?

It's actually rare for me to use the idiom " `if (object != null && ...` ".

It may be easier to give you examples, if you show examples of where you typically use the idiom.

edited Jul 19 '16 at 9:02

community wiki
2 revs, 2 users 90%
myplacedk

89 What's the point in throwing IllegalArgumentException? I think NullPointerException would be clearer, and that would also be thrown if you don't do the null-check yourself. I'd either use assert or nothing at all. – Axel Aug 9 '11 at 16:47

21 It's unlikely that every other value than null is acceptable. You could have IllegalArgumentException, OutOfRageException etc etc. Sometimes this makes sense. Other times you end up creating a lot of exception classes that doesn't add any value, then you just use IllegalArgumentException. It doesn't makes sense to have one exception for null-input, and another one for everything else. – myplacedk Aug 15 '11 at 12:26

7 Yes, I agree to fail-fast-principle, but in the example given above, the value is not passed on, but is the object on which a method shall be called. So it fails equallys fast, and adding a null check just to throw an exception that would be thrown anyway at the same time and place doesn't seem to make debugging any easier. – Axel Aug 22 '11 at 11:20

8 A security hole? The JDK is full of code like this. If you don't want the user to see stacktraces, then just disable them.

No one implied that the behaviour is not documented. MySQL is written in C where dereferencing null pointers is undefined behaviour, nothing like throwing an exception. – fgb Apr 8 '13 at 0:39

7 `throw new IllegalArgumentException("object==null")` – Thorbjørn Ravn Andersen Apr 20 '13 at 23:35

Wow, I almost hate to add another answer when we have 57 different ways to recommend the `NullObject pattern`, but I think that some people interested in this question may like to know that there is a proposal on the table for Java 7 to add "null-safe handling"—a streamlined syntax for if-not-equal-null logic.

The example given by Alex Miller looks like this:

```
public String getPostcode(Person person) {
  return person?.getAddress()?.getPostcode();
}
```

The `?.` means only de-reference the left identifier if it is not null, otherwise evaluate the remainder of the expression as `null`. Some people, like Java Posse member Dick Wall and the voters at Devoxx really love this proposal, but there is opposition too, on the grounds that it will actually encourage more use of `null` as a sentinel value.

Update: An official proposal for a null-safe operator in Java 7 has been submitted under Project Coin. The syntax is a little different than the example above, but it's the same notion.

Update: The null-safe operator proposal didn't make it into Project Coin. So, you won't be seeing this syntax in Java 7.

edited Sep 20 '15 at 23:12

community wiki
5 revs, 3 users 82%
erickson

18 I think this is wrong. There should be a way to specify that a given variable is ALWAYS non-null. – Thorbjørn Ravn Andersen May 26 '09 at 11:54

5 Update: the proposal will not make Java7. See blogs.sun.com/darcy/entry/project_coin_final_five . – Boris Terzic Aug 29 '09 at 18:56

9 Interesting idea but the choice of syntax is absurd; I don't want a codebase full of question marks pinned into every joint. – Rob May 16 '12 at 11:51

7 This operator exists in Groovy, so those who want to use it still have that as an option. – Muhd Feb 15 '13 at 22:53

7 This is the most ingenious idea I've seen. It should be added to every sensible language of the C syntax. I'd rather "pin question marks" everywhere than scroll through screenful of lines or dodge "guard clauses" all day. – Victor Oct 1 '15 at 20:57

If undefined values are not permitted:

You might configure your IDE to warn you about potential null dereferencing. E.g. in Eclipse, see *Preferences > Java > Compiler > Errors/Warnings/Null analysis*.

If undefined values are permitted:

If you want to define a new API where undefined values make sense, use the Option Pattern (may be familiar from functional languages). It has the following advantages:

- It is stated explicitly in the API whether an input or output exists or not.
- The compiler forces you to handle the "undefined" case.
- Option is a monad, so there is no need for verbose null checking, just use map/foreach/getOrElse or a similar combinator to safely use the value (example).

Java 8 has a built-in `Optional` class (recommended); for earlier versions, there are library alternatives, for example Guava's `Optional` or FunctionalJava's `Option`. But like many functional-style patterns, using Option in Java (even 8) results in quite some boilerplate, which you can reduce using a less verbose JVM language, e.g. Scala or Xtend.

If you have to deal with an API which might return nulls, you can't do much in Java. Xtend and Groovy have the Elvis operator `?:` and the null-safe dereference operator `?.`, but note that this returns null in case of a null reference, so it just "defers" the proper handling of null.

edited Oct 24 '17 at 10:38

community wiki
10 revs, 3 users 87%
thSoft

20 Indeed, the Option pattern is awesome. Some Java equivalents exist. Guava contains a limited version of this called Optional which leaves out most of the functional stuff. In Haskell, this pattern is called Maybe. – Ben Hardy May 20 '11 at 18:31

7 An Optional class will be available in Java 8 – Pierre Henry Apr 25 '13 at 14:59

1 ...and it doesn't (yet) have map nor flatMap: download.java.net/jdk8/docs/api/java/util/Optional.html – thSoft Apr 25 '13 at 15:28

2 The Optional pattern doesn't solve anything; instead of one potentially null object, now you have two. – Boann Nov 4 '13 at 4:15

1 @Boann, if used with care, you solve all your NPEs problem. If not, then I guess there is a "usage" problem. – Louis F. Feb 12 '16 at 13:34

Only for this situation -

Not checking if a variable is null before invoking an equals method (a string compare example below):

```
if ( foo.equals("bar") ) {
 // ...
}
```

will result in a `NullPointerException` if `foo` doesn't exist.

You can avoid that if you compare your `String` s like this:

```
if ( "bar".equals(foo) ) {
 // ...
}
```

edited May 25 at 15:33

community wiki
3 revs, 3 users 68%
echox

33 I agree - only in that situation. I can't stand programmers that have taken this to the next unnecessary level and write if (null != myVar)... just looks ugly to me and serves no purpose! – Alex Worden May 23 '11 at 18:14

19 This is a particular example, probably the most used, of a general good practice: if you know it, always do `<object that you know that is not null>.equals(<object that might be null>);` . It works for other methods other than `equals` if you know the contract and those methods can handle `null` parameters. – Stef Sep 23 '11 at 1:20

9 This is the first example I have seen of Yoda Conditions that actually makes sense – Erin Drummond Sep 30 '13 at 19:37

5 NullPointerExceptions are thrown for a reason. They are thrown because an object is null where it shouldn't be. It is the programmers job to FIX this, not HIDE the problem. – Oliver Watkins May 12 '14 at 7:49

1 Try-Catch-DoNothing hides the problem, this is a valid practice to work around missing sugar in the language. – echox May 12 '14 at 11:13

With Java 8 comes the new `java.util.Optional` class that arguably solves some of the problem. One can at least say that it improves the readability of the code, and in the case of public APIs make the API's contract clearer to the client developer.

They work like that:

An optional object for a given type (`Fruit`) is created as the return type of a method. It can be empty or contain a `Fruit` object:

```
public static Optional<Fruit> find(String name, List<Fruit> fruits) {
   for (Fruit fruit : fruits) {
      if (fruit.getName().equals(name)) {
         return Optional.of(fruit);
      }
   }
   return Optional.empty();
}
```

Now look at this code where we search a list of `Fruit` (`fruits`) for a given Fruit instance:

```
Optional<Fruit> found = find("lemon", fruits);
if (found.isPresent()) {
   Fruit fruit = found.get();
   String name = fruit.getName();
}
```

You can use the `map()` operator to perform a computation on--or extract a value from--an optional object. `orElse()` lets you provide a fallback for missing values.

```
String nameOrNull = find("lemon", fruits)
    .map(f -> f.getName())
    .orElse("empty-name");
```

Of course, the check for null/empty value is still necessary, but at least the developer is conscious that the value might be empty and the risk of forgetting to check is limited.

In an API built from scratch using `Optional` whenever a return value might be empty, and returning a plain object only when it cannot be `null` (convention), the client code might abandon null checks on simple object return values...

Of course `Optional` could also be used as a method argument, perhaps a better way to indicate optional arguments than 5 or 10 overloading methods in some cases.

`Optional` offers other convenient methods, such as `orElse` that allow the use of a default value, and `ifPresent` that works with lambda expressions.

I invite you to read this article (my main source for writing this answer) in which the `NullPointerException` (and in general null pointer) problematic as well as the (partial) solution brought by `Optional` are well explained: *Java Optional Objects*.

edited Sep 29 '16 at 7:39

community wiki
4 revs, 4 users 72%
Pierre Henry

9 Google's guava has an optional implimention for Java 6+. – Bradley Gottfried May 27 '13 at 17:30

10 It's *very* important to emphasise that using Optional only with ifPresent() does *not* add much value above normal null checking. It's core value is that it is a monad that can be used in function chains of map/flapMap, which achieves results similar to the Elvis operator in Groovy mentioned elsewhere. Even without this usage, though, I find the orElse/orElseThrow syntax also very useful. – Cornel Masson Oct 9 '14 at 14:57

This blog has a good entry on Optional winterbe.com/posts/2015/03/15/avoid-null-checks-in-java – JohnC Dec 9 '15 at 0:05

I really never understood why people are so happy with this boilerplate code dzone.com/articles/java-8-elvis-operator – Mykhaylo Adamovych Feb 23 at 22:10

Why people have tendency to do this `if(optional.isPresent()){ optional.get(); }` instead of `optional.ifPresent(o -> { ...}) ` – Satyendra Kumar Mar 7 at 15:06

Depending on what kind of objects you are checking you may be able to use some of the classes in the apache commons such as: apache commons lang and apache commons collections

Example:

```
String foo;
...
if( StringUtils.isBlank( foo ) ) {
   ///do something
}
```

or (depending on what you need to check):

```
String foo;
...
if( StringUtils.isEmpty( foo ) ) {
   ///do something
}
```

The StringUtils class is only one of many; there are quite a few good classes in the commons that do null safe manipulation.

Here follows an example of how you can use null vallidation in JAVA when you include apache library(commons-lang-2.4.jar)

```
public DOCUMENT read(String xml, ValidationEventHandler validationEventHandler) {
    Validate.notNull(validationEventHandler,"ValidationHandler not Injected");
    return read(new StringReader(xml), true, validationEventHandler);
}
```

And if you are using Spring, Spring also has the same functionality in its package, see library(spring-2.4.6.jar)

Example on how to use this static classf from spring(org.springframework.util.Assert)

```
Assert.notNull(validationEventHandler,"ValidationHandler not Injected");
```

edited Feb 23 '16 at 12:14

community wiki
3 revs, 3 users 68%
javamonkey79

5 Also you can use the more generic version from Apache Commons, quite useful at the start of methods to check params I find. Validate.notNull(object, "object must not be null");
commons.apache.org/lang/apidocs/org/apache/commons/lang/… – monojohnny Jan 14 '10 at 13:57

@monojohnny does Validate use Assert statements into?. i ask that because Assert may be activate / deactivate on JVM and it's suggest do not use in production. – Kurapika Dec 9 '17 at 16:03

Don't think so - I believe it just throws a RuntimeException if validation fails – monojohnny Jan 6 at 14:45

- If you consider an object should not be null (or it is a bug) use an assert.
- If your method doesn't accept null params say it in the javadoc and use an assert.

You have to check for object != null only if you want to handle the case where the object may be null...

There is a proposal to add new annotations in Java7 to help with null / notnull params:
http://tech.puredanger.com/java7/#jsr308

edited Aug 14 '09 at 14:44

community wiki
2 revs, 2 users 100%
pgras

2 No, do not use assertions in production code. – Blauhirn Feb 26 '17 at 0:54

I'm a fan of "fail fast" code. Ask yourself - are you doing something useful in the case where the parameter is null? If you don't have a clear answer for what your code should do in that case... I.e. it should never be null in the first place, then ignore it and allow a NullPointerException to be thrown. The calling code will make just as much sense of an NPE as it would an IllegalArgumentException, but it'll be easier for the developer to debug and understand what went wrong if an NPE is thrown rather than your code attempting to execute some other unexpected contingency logic - which ultimately results in the application failing anyway.

edited Apr 23 '12 at 19:09

community wiki
2 revs
Alex Worden

1 better to use assertions, i.e Contract.notNull(abc, "abc must be non-null, did it fail to load during xyz?"); - this is a more compact way than doing an if (abc!=null) { throw new RuntimeException...} – ianpojman Oct 18 '12 at 3:35

Sometimes, you have methods that operate on its parameters that define a symmetric operation:

`a.f(b); <-> b.f(a);`

If you know b can never be null, you can just swap it. It is most useful for equals: Instead of `foo.equals("bar");` better do `"bar".equals(foo);` .

edited Dec 15 '15 at 19:02

community wiki
4 revs, 4 users 50%
Johannes Schaub - litb

2 But then you have to assume `equals` (could be any method) will handle null correctly. Really all this is doing is passing the responsibility to someone else (or another method). – Supericy Jun 13 '13 at 20:01

4 @Supericy Basically yes, but `equals` (or whatever method) has to check for `null` anyway. Or state explicitly that it does not. – Angelo Fuchs Jul 30 '13 at 7:26

The Google collections framework offers a good and elegant way to achieve the null check.

There is a method in a library class like this:

```
static <T> T checkNotNull(T e) {
   if (e == null) {
      throw new NullPointerException();
   }
   return e;
}
```

And the usage is (with `import static`):

```
...
void foo(int a, Person p) {
   if (checkNotNull(p).getAge() > a) {
      ...
   }
   else {
      ...
   }
}
...
```

Or in your example:

```
checkNotNull(someobject).doCalc();
```

edited Sep 19 '15 at 16:05

community wiki
3 revs, 3 users 83%
user2427

70 mmm, what is the difference? p.getAge() would throw the same NPE with less overhead and a clearer stack trace. What am I missing? – mysomic May 18 '09 at 23:26

15 It is better to throw an IllegalArgumentException("e == null") in your example as it clearly indicates that it is a programmer-intended exception (along with enough information to actually allow the maintainer to identify the problem). NullPointerExceptions should be reserved for the JVM, as it then clearly indicates that this was unintentional (and usually happens somewhere hard to identify) – Thorbjørn Ravn Andersen May 26 '09 at 11:56

6 This is now part of Google Guava. – Steven Benitez Feb 13 '11 at 20:55

29 Smells like over-engineering to me. Just let the JVM throw an NPE and don't clutter your code with this junk. – Alex Worden Apr 23 '12 at 19:11

5 I like it and open most methods and constructors with explicit checks on the arguments; if there is an error, methods always fail on the first few lines and I know the offending reference without finding something like `getThing().getItsThing().getOtherThing().wowEncapsulationIsBroken().setLol("hi");` – Cory Kendall Nov 22 '12 at 6:26

Rather than Null Object Pattern -- which has its uses -- you might consider situations where the null object is a bug.

When the exception is thrown, examine the stack trace and work through the bug.

answered Nov 7 '08 at 8:50

community wiki
Jim Nelson

16 Problem is that usually you loose context as the NullPointerException does not indicate WHICH variable was null, and you may have several "."-operations on the line. Using "if (foo == null) throw new RuntimeException("foo == null")" allows you to state explicitly WHAT was wrong, giving your stack trace much more value to those who have to fix it. – Thorbjørn Ravn Andersen Oct 31 '09 at 9:02

1 With Andersen - I would love Java's exception system to be able to include the name of a variable that's being worked upon, so that NullPointerExceptions would not only indicate the line the exception occurred on, but also the variable name. This should work just fine in unobfuscated software. – fwielstra Jan 18 '11 at 12:53

I haven't managed to get it working just yet, but this is intended to solve exactly that problem. – MatrixFrog Sep 8 '11 at 23:15

What is the problem? Just catch the NPE at the appropriate level where enough context is available, dump the context information and rethrow the exception... it's so easy with Java. – user1050755 Mar 7 '13 at 1:37

3 I had a professor that preached against method call chaining. His theory was that you should be wary of call chains that were longer than 2 methods. I don't know if that's a hard rule, but it definitely removes most of the problems with NPE stack traces. – RustyTheBoyRobot Apr 16 '13 at 17:12

Java 7 has a new `java.util.Objects` utility class on which there is a `requireNonNull()` method. All this does is throw a `NullPointerException` if its argument is null, but it cleans up the code a bit. Example:

```
Objects.requireNonNull(someObject);
someObject.doCalc();
```

The method is most useful for checking just before an assignment in a constructor, where each use of it can save three lines of code:

```
Parent(Child child) {
   if (child == null) {
      throw new NullPointerException("child");
   }
   this.child = child;
}
```

becomes

```
Parent(Child child) {
   this.child = Objects.requireNonNull(child, "child");
}
```

edited May 23 '17 at 12:18

community wiki
3 revs, 2 users 67%
Raedwald

7 Actually, your example constitutes code bloat: the first line is superfluous because the NPE would be thrown in the second line. ;-) – user1050755 Mar 7 '13 at 1:21

1 True. A better example would be if the second line were `doCalc(someObject)` . – Stuart Marks Mar 7 '13 at 5:23

Depends. If you are the author of doCalc(), I'd suggest putting the check into that method's body (if possible). And then you most likely will call someObject.someMethod() where again there is no need to check for null. :-) – user1050755 Mar 7 '13 at 7:07

Well, if you are not the author of `doCalc()`, and it doesn't immediately throw NPE when given null, you'd need to check for null and throw NPE yourself. That's what `Objects.requireNonNull()` is for. – Stuart Marks Mar 9 '13 at 6:43

7 It's not just code bloat. Better to check up front than halfway through a method that causes side effects or uses time/space. – Robert Grant Sep 26 '13 at 11:33

Null is not a 'problem'. It is an integral part of a complete modeling tool set. Software aims to model the complexity of the world and null bears its burden. **Null indicates 'No data' or 'Unknown'** in Java and the like. So it is appropriate to use nulls for these purposes. I don't prefer the 'Null object' pattern; I think it rise the 'who will guard the guardians' problem.
If you ask me what is the name of my girlfriend I'll tell you that I have no girlfriend. In the Java language I'll return null. An alternative would be to throw meaningful exception to indicate some problem that can't be (or don't want to be) solved right there and delegate it somewhere higher in the stack to retry or report data access error to the user.

1. **For an 'unknown question' give 'unknown answer'.** (Be null-safe where this is correct from business point of view) Checking arguments for null once inside a method before usage relieves multiple callers from checking them before a call.

```
public Photo getPhotoOfThePerson(Person person) {
    if (person == null)
        return null;
    // Grabbing some resources or intensive calculation
    // using person object anyhow.
}
```

Previous leads to normal logic flow to get no photo of a non-existent girlfriend from my photo library.

```
getPhotoOfThePerson(me.getGirlfriend())
```

And it fits with new coming Java API (looking forward)

```
getPhotoByName(me.getGirlfriend()?.getName())
```

While it is rather 'normal business flow' not to find photo stored into the DB for some person, I used to use pairs like below for some other cases

```
public static MyEnum parseMyEnum(String value); // throws
IllegalArgumentException
public static MyEnum parseMyEnumOrNull(String value);
```

And don't loathe to type `<alt> + <shift> + <j>` (generate javadoc in Eclipse) and write three additional words for you public API. This will be more than enough for all but those who don't read documentation.

```
/**
 * @return photo or null
 */
```

or

```
/**
 * @return photo, never null
 */
```

2. **This is rather theoretical case and in most cases you should prefer java null safe API (in case it will be released in another 10 years), but `NullPointerException` is subclass of an `Exception`.** Thus it is a form of `Throwable` that indicates conditions that a reasonable application might want to catch (javadoc)! To use the first most advantage of exceptions and separate error-handling code from 'regular' code (according to creators of Java) it is appropriate, as for me, to catch `NullPointerException`.

```
public Photo getGirlfriendPhoto() {
    try {
        return
appContext.getPhotoDataSource().getPhotoByName(me.getGirlfriend().getName());
    } catch (NullPointerException e) {
        return null;
    }
}
```

Questions could arise:

Q. What if `getPhotoDataSource()` returns null?
A. It is up to business logic. If I fail to find a photo album I'll show you no photos. What if appContext is not initialized? This method's business logic puts up with this. If the same logic should be more strict then throwing an exception it is part of the business logic and explicit check for null should be used

(case 3). The **new Java Null-safe API fits better here to specify selectively what implies and what does not imply to be initialized** to be fail-fast in case of programmer errors.

Q. Redundant code could be executed and unnecessary resources could be grabbed.
A. It could take place if `getPhotoByName()` would try to open a database connection, create `PreparedStatement` and use the person name as an SQL parameter at last. The approach *for an unknown question gives an unknown answer* (case 1) works here. Before grabbing resources the method should check parameters and return 'unknown' result if needed.

Q. This approach has a performance penalty due to the try closure opening.
A. Software should be easy to understand and modify firstly. Only after this, one could think about performance, and only if needed! and where needed! (source), and many others).

PS. This approach will be as reasonable to use as the *separate error-handling code from "regular" code* principle is reasonable to use in some place. Consider the next example:

```
public SomeValue calculateSomeValueUsingSophisticatedLogic(Predicate predicate) {
    try {
        Result1 result1 = performSomeCalculation(predicate);
        Result2 result2 = performSomeOtherCalculation(result1.getSomeProperty());
        Result3 result3 = performThirdCalculation(result2.getSomeProperty());
        Result4 result4 = performLastCalculation(result3.getSomeProperty());
        return result4.getSomeProperty();
    } catch (NullPointerException e) {
        return null;
    }
}

public SomeValue calculateSomeValueUsingSophisticatedLogic(Predicate predicate) {
    SomeValue result = null;
    if (predicate != null) {
        Result1 result1 = performSomeCalculation(predicate);
        if (result1 != null && result1.getSomeProperty() != null) {
            Result2 result2 =
performSomeOtherCalculation(result1.getSomeProperty());
            if (result2 != null && result2.getSomeProperty() != null) {
                Result3 result3 =
performThirdCalculation(result2.getSomeProperty());
                if (result3 != null && result3.getSomeProperty() != null) {
                    Result4 result4 =
performLastCalculation(result3.getSomeProperty());
                    if (result4 != null) {
                        result = result4.getSomeProperty();
                    }
                }
```

```
            }
        }
    }
    return result;
}
```

PPS. For those fast to downvote (and not so fast to read documentation) I would like to say that I've never caught a null-pointer exception (NPE) in my life. But this possibility was **intentionally designed** by the Java creators because NPE is a subclass of `Exception`. We have a precedent in Java history when `ThreadDeath` is an `Error` not because it is actually an application error, but solely because it was not intended to be caught! How much NPE fits to be an `Error` than `ThreadDeath`! But it is not.

3. **Check for 'No data' only if business logic implies it.**

```
public void updatePersonPhoneNumber(Long personId, String phoneNumber) {
    if (personId == null)
        return;
    DataSource dataSource = appContext.getStuffDataSource();
    Person person = dataSource.getPersonById(personId);
    if (person != null) {
        person.setPhoneNumber(phoneNumber);
        dataSource.updatePerson(person);
    } else {
        Person = new Person(personId);
        person.setPhoneNumber(phoneNumber);
        dataSource.insertPerson(person);
    }
}
```

and

```
public void updatePersonPhoneNumber(Long personId, String phoneNumber) {
    if (personId == null)
        return;
    DataSource dataSource = appContext.getStuffDataSource();
    Person person = dataSource.getPersonById(personId);
    if (person == null)
        throw new SomeReasonableUserException("What are you thinking about ???");
    person.setPhoneNumber(phoneNumber);
    dataSource.updatePerson(person);
}
```

If appContext or dataSource is not initialized unhandled runtime NullPointerException will kill current thread and will be processed by Thread.defaultUncaughtExceptionHandler (for you to define and use your favorite logger or other notification mechanizm). If not set, ThreadGroup#uncaughtException will

print stacktrace to system err. One should monitor application error log and open Jira issue for each unhandled exception which in fact is application error. Programmer should fix bug somewhere in initialization stuff.

edited Feb 23 at 22:03

community wiki
45 revs, 2 users 73%
Mykhaylo Adamovych

6 Catching `NullPointerException` and returning `null` is horrible to debug. You end up with NPE later on anyway, and it's really hard to figure out what was originally null. – artbristol Sep 14 '13 at 8:44

21 I'd downvote if I had the reputation. Not only is null not necessary, it's a hole in the type system. Assigning a Tree to a List is a type error because trees are not values of type List; by that same logic, assigning null should be a type error because null is not a value of type Object, or any useful type for that matter. Even the man that invented null considers it his "billion-dollar mistake". The notion of "a value that might be a value of type T OR nothing" is its own type, and should be represented as such (e.g. Maybe<T> or Optional<T>). – Doval Nov 20 '13 at 15:57

2 As of "Maybe<T> or Optional<T>" you still need to write code like `if (maybeNull.hasValue()) {...}` so what is the difference with `if (maybeNull != null)) {...}` ? – Mykhaylo Adamovych Apr 1 '14 at 11:06

1 As of "catching NullPointerException and returning null is horrible to debug. You end up with NPE later on anyway, and it's really hard to figure out what was originally null". I'm totally agree! In those cases you should write a dozen of 'if' statements or throw NPE if business logic imply data in-place, or use null-safe operator from new Java. But there are cases when I don't care about what exact step give me null. For example calculating some values for the user just before showing on the screen when you do expect data could be missing. – Mykhaylo Adamovych Apr 1 '14 at 11:32

2 @MykhayloAdamovych: The benefit of `Maybe<T>` or `Optional<T>` isn't in the case where your `T` might be null, but in the case where it should never be null. If you have a type that explicitly means "this value *might* be null -- use with caution", and you use and return such a type consistently, then whenever you see a plain old `T` in your code, you can assume it is never null. (Course, this would be a lot more useful if enforceable by the compiler.) – cHao Nov 7 '14 at 19:55

Ultimately, the only way to completely solve this problem is by using a different programming language:

- In Objective-C, you can do the equivalent of invoking a method on `nil`, and absolutely nothing will happen. This makes most null checks unnecessary, but it can make errors much harder to diagnose.
- In Nice, a Java-derived language, there are two versions of all types: a potentially-null version and a not-null version. You can only invoke methods on not-null types. Potentially-null types can be converted to not-null types through explicit checking for null. This makes it much easier to know where null checks are necessary and where they aren't.

edited Sep 19 '15 at 16:09

community wiki
2 revs, 2 users 75%
Michael Borgwardt

Woah... The most correct answer and its gets downvoted, where's the justice? In Java null is *always* a valid value. It's the corollary to Everything is an Object - Null is an Everything (of course we're ignoring primitives here but you get the idea). Personally I favour the approach taken by Nice, though we could make it so that methods can be invoked on nullable types and promote NPEs to checked exceptions. This would have to be done via a compiler switch though as it would break all existing code :(– CurtainDog Aug 16 '10 at 6:36

Common "problem" in Java indeed.

First, my thoughts on this:

I consider that it is bad to "eat" something when NULL was passed where NULL isn't a valid value. If you're not exiting the method with some sort of error then it means nothing went wrong in your method which is not true. Then you probably return null in this case, and in the receiving method you again check for null, and it never ends, and you end up with "if != null", etc..

So, IMHO, null must be a critical error which prevents further execution (that is, where null is not a valid value).

The way I solve this problem is this:

First, I follow this convention:

1. All public methods / API always check its arguments for null
2. All private methods do not check for null since they are controlled methods (just let die with nullpointer exception in case it wasn't handled above)
3. The only other methods which do not check for null are utility methods. They are public, but if you call them for some reason, you know what parameters you pass. This is like trying to boil water in the kettle without providing water...

And finally, in the code, the first line of the public method goes like this:

```
ValidationUtils.getNullValidator().addParam(plans, "plans").addParam(persons,
"persons").validate();
```

Note that addParam() returns self, so that you can add more parameters to check.

Method `validate()` will throw checked `ValidationException` if any of the parameters is null (checked or unchecked is more a design/taste issue, but my `ValidationException` is checked).

```
void validate() throws ValidationException;
```

The message will contain the following text if, for example, "plans" is null:

"**Illegal argument value null is encountered for parameter [plans]**"

As you can see, the second value in the addParam() method (string) is needed for the user message, because you cannot easily detect passed-in variable name, even with reflection (not subject of this post anyway...).

And yes, we know that beyond this line we will no longer encounter a null value so we just safely invoke methods on those objects.

This way, the code is clean, easy maintainable and readable.

edited Sep 19 '15 at 19:00

community wiki
3 revs, 3 users 79%
Oleg

1 Absolutely. Applications that just throw the error and crash are of higher quality because there is no doubt when they are not working. Applications that swallow the errors at best degrade gracefully but usually don't work in ways that are difficult to notice and don't get fixed. And when the problem is noticed they are much harder to debug. – Paul Jackson Nov 14 '11 at 5:06

Asking that question points out that you may be interested in error handling strategies. Your team's architect should decide how to work errors. There are several ways to do this:

1. allow the Exceptions to ripple through - catch them at the 'main loop' or in some other managing routine.
    - check for error conditions and handle them appropriately

Sure do have a look at Aspect Oriented Programming, too - they have neat ways to insert `if( o == null ) handleNull()` into your bytecode.

answered Nov 7 '08 at 9:27

community wiki
xtofl

In addition to using `assert` you can use the following:

```
if (someobject == null) {
    // Handle null here then move on.
}
```

This is slightly better than:

```
if (someobject != null) {
    .....
    .....



    .....
}
```

edited Sep 19 '15 at 17:14

community wiki
2 revs, 2 users 91%
fastcodejava

2 Mh, why that? Please don't feel any defensive, I'd just like to learn more about Java :) – Matthias Meid Aug 17 '10 at 5:54

9 @Mudu As a general rule, I prefer the expression in an if statement to be a more "positive" statement, rather than a "negative" one. So if I saw `if (!something) { x(); } else { y(); }` I would be inclined to refactor it as `if (something) { y(); } else { x(); }` (though one could argue that `!= null` is the more positive option...). But more importantly, the important part of the code is not wrapped inside `{}` s and you have one level less of indentation for most of the method. I don't know if that was fastcodejava's reasoning but that would be mine. – MatrixFrog Jun 22 '11 at 0:05

1 This is what I tend to do as well.. Keeps the code clean in my opinon. – Koray Tugay Dec 15 '15 at 21:44

Just don't ever use null. Don't allow it.

In my classes, most fields and local variables have non-null default values, and I add contract statements (always-on asserts) everywhere in the code to make sure this is being enforced (since it's more succinct, and more expressive than letting it come up as an NPE and then having to resolve the line number, etc.).

Once I adopted this practice, I noticed that the problems seemed to fix themselves. You'd catch things much earlier in the development process just by accident and realize you had a weak spot.. and more importantly.. it helps encapsulate different modules' concerns, different modules can 'trust' each other, and no more littering the code with `if = null else` constructs!

This is defensive programming and results in much cleaner code in the long run. Always sanitize the data, e.g. here by enforcing rigid standards, and the problems go away.

```
class C {
    private final MyType mustBeSet;
    public C(MyType mything) {
       mustBeSet=Contract.notNull(mything);
    }
   private String name = "<unknown>";
   public void setName(String s) {
      name = Contract.notNull(s);
   }
}


class Contract {
    public static <T> T notNull(T t) { if (t == null) { throw new
ContractException("argument must be non-null"); return t; }
}
```

The contracts are like mini-unit tests which are always running, even in production, and when things fail, you know why, rather than a random NPE you have to somehow figure out.

edited Sep 19 '15 at 19:42

community wiki
3 revs, 3 users 68%
iangreen

why would this be downvoted? in my experience, this is far superior to the other approaches, would love to know why not – ianpojman Oct 31 '12 at 21:31

I agree, this approach prevents problems related to nulls, rather than fixing them by speckling code null checks everywhere. – ChrisBlom Jul 27 '13 at 10:48

7 The problem with this approach is that if name is never set, it has the value "<unknown>", which behaves like a set value. Now let's say I need to check if name was never set (unknown), I have to do a string comparison against the

special value "<unknown>". – Steve Kuo Dec 17 '13 at 7:02

1 True good point Steve. What I often do is have that value as a constant, e.g. public static final String UNSET="__unset" ... private String field = UNSET ... then private boolean isSet() { return UNSET.equals(field); } – ianpojman Jan 26 '15 at 19:45

IMHO, This is a implementation of Null Object Pattern with a yourself implementation of Optional (Contract). How it behaves on persistence class class? I do not see applicable in that case. – Kurapika Dec 9 '17 at 19:56

Guava, a very useful core library by Google, has a nice and useful API to avoid nulls. I find UsingAndAvoidingNullExplained very helpful.

As explained in the wiki:

> `Optional<T>` is a way of replacing a nullable T reference with a non-null value. An Optional may either contain a non-null T reference (in which case we say the reference is "present"), or it may contain nothing (in which case we say the reference is "absent"). It is never said to "contain null."

Usage:

```
Optional<Integer> possible = Optional.of(5);
possible.isPresent(); // returns true
possible.get(); // returns 5
```

edited Aug 7 '14 at 21:03

community wiki
2 revs
Murat

@CodyGuldner Right, Cody. I provided a relevant quote from the link to give more context. – Murat Derya Özen Aug 7 '14 at 21:05

This is a very common problem for every Java developer. So there is official support in Java 8 to address these issues without cluttered code.

Java 8 has introduced `java.util.Optional<T>`. It is a container that may or may not hold a non-null value. Java 8 has given a safer way to handle an object whose value may be null in some of the cases. It is inspired from the ideas of Haskell and Scala.

In a nutshell, the Optional class includes methods to explicitly deal with the cases where a value is present or absent. However, the advantage compared to null references is that the Optional<T> class forces you to think about the case when the value is not present. As a consequence, you can prevent unintended null pointer exceptions.

In above example we have a home service factory that returns a handle to multiple appliances available in the home. But these services may or may not be available/functional; it means it may result in a NullPointerException. Instead of adding a null `if` condition before using any service, let's wrap it in to Optional<Service>.

WRAPPING TO OPTION<T>

Let's consider a method to get a reference of a service from a factory. Instead of returning the service reference, wrap it with Optional. It lets the API user know that the returned service may or may not available/functional, use defensively

```
public Optional<Service> getRefrigertorControl() {
      Service s = new  RefrigeratorService();
       //...
      return Optional.ofNullable(s);
   }
```

As you see `Optional.ofNullable()` provides an easy way to get the reference wrapped. There are another ways to get the reference of Optional, either `Optional.empty()` & `Optional.of()` . One for returning an empty object instead of retuning null and the other to wrap a non-nullable object, respectively.

SO HOW EXACTLY IT HELPS TO AVOID A NULL CHECK?

Once you have wrapped a reference object, Optional provides many useful methods to invoke methods on a wrapped reference without NPE.

```
Optional ref = homeServices.getRefrigertorControl();
ref.ifPresent(HomeServices::switchItOn);
```

Optional.ifPresent invokes the given Consumer with a reference if it is a non-null value. Otherwise, it does nothing.

```
@FunctionalInterface
public interface Consumer<T>
```

Represents an operation that accepts a single input argument and returns no result. Unlike most other functional interfaces, Consumer is expected to operate via side-effects. It is so clean and easy to

understand. In the above code example, `HomeService.switchOn(Service)` gets invoked if the Optional holding reference is non-null.

We use the ternary operator very often for checking null condition and return an alternative value or default value. Optional provides another way to handle the same condition without checking null. Optional.orElse(defaultObj) returns defaultObj if the Optional has a null value. Let's use this in our sample code:

```
public static Optional<HomeServices> get() {
    service = Optional.of(service.orElse(new HomeServices()));
    return service;
}
```

Now HomeServices.get() does same thing, but in a better way. It checks whether the service is already initialized of not. If it is then return the same or create a new New service. Optional<T>.orElse(T) helps to return a default value.

Finally, here is our NPE as well as null check-free code:

```
import java.util.Optional;
public class HomeServices {
    private static final int NOW = 0;
    private static Optional<HomeServices> service;

public static Optional<HomeServices> get() {
    service = Optional.of(service.orElse(new HomeServices()));
    return service;
}

public Optional<Service> getRefrigertorControl() {
    Service s = new  RefrigeratorService();
    //...
    return Optional.ofNullable(s);
}

public static void main(String[] args) {
    /* Get Home Services handle */
    Optional<HomeServices> homeServices = HomeServices.get();
    if(homeServices != null) {
        Optional<Service> refrigertorControl =
homeServices.get().getRefrigertorControl();
        refrigertorControl.ifPresent(HomeServices::switchItOn);
    }
}

public static void switchItOn(Service s){
```

```
            //...
        }
    }
```

The complete post is *NPE as well as Null check-free code … Really?*.

edited Sep 19 '15 at 20:39

community wiki
2 revs, 2 users 87%
Yogesh Devatraj

I like articles from Nat Pryce. Here are the links:

- Avoiding Nulls with Polymorphic Dispatch
- Avoiding Nulls with "Tell, Don't Ask" Style

In the articles there is also a link to a Git repository for a Java Maybe Type which I find interesting, but I don't think it alone could decrease the checking code bloat. After doing some research on the Internet, I think **!= null** code bloat could be decreased mainly by careful design.

edited May 12 '16 at 2:49

community wiki
2 revs, 2 users 74%
Mr Palo

Michael Feathers has written an short and interesthing text about approaches like the one you mentioned: manuelp.newsblur.com/site/424 – ivan.aguirre Jul 12 '13 at 20:59

I've tried the `NullObjectPattern` but for me is not always the best way to go. There are sometimes when a "no action" is not appropiate.

`NullPointerException` is a *Runtime exception* that means it's developers fault and with enough experience it tells you exactly where is the error.

Now to the answer:

Try to make all your attributes and its accessors as private as possible or avoid to expose them to the clients at all. You can have the argument values in the constructor of course, but by reducing the scope you

don't let the client class pass an invalid value. If you need to modify the values, you can always create a new `object`. You check the values in the constructor only **once** and in the rest of the methods you can be almost sure that the values are not null.

Of course, experience is the better way to understand and apply this suggestion.

Byte!

edited Sep 30 '13 at 22:06

community wiki
3 revs, 2 users 91%
OscarRyz

Probably the best alternative for Java 8 or newer is to use the `Optional` class.

```
Optional stringToUse = Optional.of("optional is there");
stringToUse.ifPresent(System.out::println);
```

This is especially handy for long chains of possible null values. Example:

```
Optional<Integer> i = Optional.ofNullable(wsObject.getFoo())
    .map(f -> f.getBar())
    .map(b -> b.getBaz())
    .map(b -> b.getInt());
```

Example on how to throw exception on null:

```
Optional optionalCarNull = Optional.ofNullable(someNull);
optionalCarNull.orElseThrow(IllegalStateException::new);
```

Java 7 introduced the `Objects.requireNonNull` method which can be handy when something should be checked for non-nullness. Example:

```
String lowerVal = Objects.requireNonNull(someVar, "input cannot be null or
empty").toLowerCase();
```

edited Sep 29 '16 at 7:35

community wiki
3 revs, 2 users 74%
Raghu K Nair

May I answer it more generally!

We **usually** face this issue when the methods get the parameters in the way we not expected (bad method call is programmer's fault). For example: you expect to get an object, instead you get a null. You expect to get an String with at least one character, instead you get an empty String ...

So there is no difference between:

```
if(object == null){
   //you called my method badly!

}
```

or

```
if(str.length() == 0){
   //you called my method badly again!
}
```

They both want to make sure that we received valid parameters, before we do any other functions.

As mentioned in some other answers, to avoid above problems you can follow the **Design by contract** pattern. Please see http://en.wikipedia.org/wiki/Design_by_contract.

To implement this pattern in java, you can use core java annotations like **javax.annotation.NotNull** or use more sophisticated libraries like **Hibernate Validator**.

Just a sample:

```
getCustomerAccounts(@NotEmpty String customerId,@Size(min = 1) String
accountType)
```

Now you can safely develop the core function of your method without needing to check input parameters, they guard your methods from unexpected parameters.

You can go a step further and make sure that only valid pojos could be created in your application. (sample from hibernate validator site)

```
public class Car {

   @NotNull
   private String manufacturer;
```

```
    @NotNull
    @Size(min = 2, max = 14)
    private String licensePlate;

    @Min(2)
    private int seatCount;

    // ...
}
```

edited Apr 24 '14 at 14:48

community wiki
2 revs, 2 users 96%
Alireza Fattahi

`javax` is, by definition, *not* "core Java". – Tomas Oct 30 '15 at 4:02

I highly disregard answers that suggest using the null objects in every situation. This pattern may break the contract and bury problems deeper and deeper instead of solving them, not mentioning that used inappropriately will create another pile of boilerplate code that will require future maintenance.

In reality if something returned from a method can be null and the calling code has to make decision upon that, there should an earlier call that ensures the state.

Also keep in mind, that null object pattern will be memory hungry if used without care. For this - the instance of a NullObject should be shared between owners, and not be an unigue instance for each of these.

Also I would not recommend using this pattern where the type is meant to be a primitive type representation - like mathematical entities, that are not scalars: vectors, matrices, complex numbers and POD(Plain Old Data) objects, which are meant to hold state in form of Java built-in types. In the latter case you would end up calling getter methods with arbitrary results. For example what should a NullPerson.getName() method return?

It's worth considering such cases in order to avoid absurd results.

edited Feb 4 '16 at 11:24

community wiki
4 revs
luke1985

The solution with "hasBackground()" has one drawback - it's not thread-safe. If you need to call two methods instead of one, you need to synchronize the whole sequence in multi-threaded environment. – pkalinow Feb 3 '16 at 12:26

@pkalinow You made a contrived example only to point out that this solution has a drawback. If code is not meant to run in multithreaded application then there is no drawback. I could put at you probably 90% of your code that is not thread safe. We're not speaking here about this aspect of code, we're speaking about design pattern. And multithreading is a topic on its own. – spectre Feb 3 '16 at 15:29

Of course in a single-thread application it's not a problem. I've given that comment because sometimes it is a problem. – pkalinow Feb 4 '16 at 10:34

@pkalinow If you study this topic closer you will find out that Null Object design pattern won't fix the multithreading problems. So it's irrelevant. And to be honest, I've found places where this pattern would fit in nicely, so my original answer is a bit wrong, actually. – spectre Feb 4 '16 at 10:57

1. Never initialise variables to null.
2. If (1) is not possible, initialise all collections and arrays to empty collections/arrays.

Doing this in your own code and you can avoid != null checks.

Most of the time null checks seem to guard loops over collections or arrays, so just initialise them empty, you won't need any null checks.

```
// Bad
ArrayList<String> lemmings;
String[] names;

void checkLemmings() {
    if (lemmings != null) for(lemming: lemmings) {
        // do something
    }
}



// Good
ArrayList<String> lemmings = new ArrayList<String>();
String[] names = {};

void checkLemmings() {
    for(lemming: lemmings) {
        // do something
```

```
    }
}
```

There is a tiny overhead in this, but it's worth it for cleaner code and less NullPointerExceptions.

edited Oct 2 '13 at 13:50

community wiki
2 revs
Stuart Axon

stackoverflow.com/questions/1386275/… – Terence Jul 1 '13 at 7:45

3 +1 This I agree with. You should never return half initialized objects. Jaxb related code and bean code is notiroius for this. It is bad practice. All collections should be initialized, and all objects should exist with (ideally) no null references. Consider a object that has a collection in it. Checking that the object is not null, that the collection is not null and that the collection does not contain null objects is unreasonable and foolish. – ggb667 Sep 23 '13 at 14:52

This is the most common error occurred for most of the developers.

We have number of ways to handle this.

Approach 1:

```
org.apache.commons.lang.Validate //using apache framework
```

notNull(Object object, String message)

Approach 2:

```
if(someObject!=null){ // simply checking against null
}
```

Approach 3:

```
@isNull @Nullable  // using annotation based validation
```

Approach 4:

```
// by writing static method and calling it across whereever we needed to check
the validation
```

```
static <T> T isNull(someObject e){
   if(e == null){
      throw new NullPointerException();
   }
   return e;
}
```

edited Mar 24 '15 at 17:13

community wiki
2 revs, 2 users 81%
Sireesh Yarlagadda

Ad. 4. It is not very useful - when you check if a pointer is null, you probably want to call a method on it. Calling a method on null gives you the same behavior - NullPointerException. – pkalinow Feb 3 '16 at 12:18

```
public static <T> T ifNull(T toCheck, T ifNull) {
    if (toCheck == null) {
           return ifNull;
    }
    return toCheck;
}
```

edited Sep 19 '15 at 17:17

community wiki
2 revs, 2 users 67%
tltester

2 What's wrong with this method, I think @tltester just want to give a default value if the it's null, which make sense. – Sawyer Aug 17 '11 at 5:42

1 There is such a method in Apache commons-lang: `ObjectUtils.defaultIfNull()` . There is one more general: `ObjectUtils.firstNonNull()` , which can be used to implement a degrading strategy: `firstNonNull(bestChoice, secondBest, thirdBest, fallBack);` – ivant Apr 16 '15 at 8:04

1 2 next

protected by Mr. Alien May 11 '13 at 21:47

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?