http://www.yegor256.com/2014/05/13/why-null-is-bad.html

Why NULL is Bad?

13 May 2014 modified on 24 October 2014 Yegor Bugayenko

A simple example of `NULL` usage in Java:

```
public Employee getByName(String name) {
  int id = database.find(name);
  if (id == 0) {
    return null;
  }
  return new Employee(id);
}
```

What is wrong with this method?

It may return `NULL` instead of an object — that's what is wrong. `NULL` is a terrible practice in an object-oriented paradigm and should be avoided at all costs. There have been a number of opinions about this published already, including Null References, The Billion Dollar Mistake presentation by Tony Hoare and the entire Object Thinking book by David West.

Here, I'll try to summarize all the arguments and show examples of how `NULL` usage can be avoided and replaced with proper object-oriented constructs.

Basically, there are two possible alternatives to `NULL`.

The first one is **Null Object** design pattern (the best way is to make it a constant):

```
public Employee getByName(String name) {
  int id = database.find(name);
  if (id == 0) {
    return Employee.NOBODY;
  }
  return Employee(id);
}
```

The second possible alternative is to fail fast by throwing an **Exception** when you can't return an object:

```
public Employee getByName(String name) {
  int id = database.find(name);
  if (id == 0) {
    throw new EmployeeNotFoundException(name);
  }
  return Employee(id);
}
```

Now, let's see the arguments against `NULL`.

Besides Tony Hoare's presentation and David West's book mentioned above, I read these publications before writing this post: Clean Code by Robert Martin, Code Complete by Steve McConnell, Say "No" to "Null" by John Sonmez, Is returning null bad design? discussion at StackOverflow.

Ad-hoc Error Handling

Every time you get an object as an input you must check whether it is `NULL` or a valid object reference. If you forget to check, a `NullPointerException` (NPE) may break execution in runtime. Thus, your logic becomes polluted with multiple checks and if/then/else forks:

```
// this is a terrible design, don't reuse
Employee employee = dept.getByName("Jeffrey");
if (employee == null) {
```

```
  System.out.println("can't find an employee");
  System.exit(-1);
} else {
  employee.transferTo(dept2);
}
```

This is how exceptional situations are supposed to be handled in C and other imperative procedural languages. OOP introduced exception handling primarily to get rid of these ad-hoc error handling blocks. In OOP, we let exceptions bubble up until they reach an application-wide error handler and our code becomes much cleaner and shorter:

```
dept.getByName("Jeffrey").transferTo(dept2);
```

Consider `NULL` references an inheritance of procedural programming, and use 1) Null Objects or 2) Exceptions instead.

Ambiguous Semantic

In order to explicitly convey its meaning, the function `getByName()` has to be named `getByNameOrNullIfNotFound()`. The same should happen with every function that returns an object or `NULL`. Otherwise, ambiguity is inevitable for a code reader. Thus, to keep semantic unambiguous, you should give longer names to functions.

To get rid of this ambiguity, always return a real object, a null object or throw an exception.

Some may argue that we sometimes have to return `NULL`, for the sake of performance. For example, method `get()` of interface `Map` in Java returns `NULL` when there is no such item in the map:

```
Employee employee = employees.get("Jeffrey");
if (employee == null) {
```

```
  throw new EmployeeNotFoundException();
}
return employee;
```

This code searches the map only once due to the usage of `NULL` in `Map`. If we would refactor `Map` so that its method `get()` will throw an exception if nothing is found, our code will look like this:

```
if (!employees.containsKey("Jeffrey")) { // first search
  throw new EmployeeNotFoundException();
}
return employees.get("Jeffrey"); // second search
```

Obviously, this is method is twice as slow as the first one. What to do?

The `Map` interface (no offense to its authors) has a design flaw. Its method `get()` should have been returning an `Iterator` so that our code would look like:

```
Iterator found = Map.search("Jeffrey");
if (!found.hasNext()) {
  throw new EmployeeNotFoundException();
}
return found.next();
```

BTW, that is exactly how C++ STL map::find() method is designed.

Computer Thinking vs. Object Thinking

Statement `if (employee == null)` is understood by someone who knows that an object in Java is a pointer to a data structure and that `NULL` is a pointer to nothing (`0x00000000` , in Intel x86 processors).

However, if you start thinking as an object, this statement makes much less sense. This is how our code looks from an object point of view:

```
- Hello, is it a software department?
- Yes.
- Let me talk to your employee "Jeffrey" please.
- Hold the line please...
- Hello.
- Are you NULL?
```

The last question in this conversation sounds weird, doesn't it?

Instead, if they hang up the phone after our request to speak to Jeffrey, that causes a problem for us (Exception). At that point, we try to call again or inform our supervisor that we can't reach Jeffrey and complete a bigger transaction.

Alternatively, they may let us speak to another person, who is not Jeffrey, but who can help with most of our questions or refuse to help if we need something "Jeffrey specific" (Null Object).

Slow Failing

Instead of failing fast, the code above attempts to die slowly, killing others on its way. Instead of letting everyone know that something went wrong and that an exception handling should start immediately, it is hiding this failure from its client.

This argument is close to the "ad-hoc error handling" discussed above.

It is a good practice to make your code as fragile as possible, letting it break when necessary.

Make your methods extremely demanding as to the data they manipulate. Let them complain by throwing exceptions, if the provided data provided is not sufficient or simply doesn't fit with the main usage scenario of the method.

Otherwise, return a Null Object, that exposes some common behavior and throws exceptions on all other calls:

```
public Employee getByName(String name) {
  int id = database.find(name);
  Employee employee;
  if (id == 0) {
    employee = new Employee() {
      @Override
      public String name() {
        return "anonymous";
      }
      @Override
      public void transferTo(Department dept) {
        throw new AnonymousEmployeeException(
          "I can't be transferred, I'm anonymous"
        );
      }
    };
  } else {
    employee = Employee(id);
  }
  return employee;
}
```

Mutable and Incomplete Objects

In general, it is highly recommended to design objects with immutability in mind. This means that an object gets all necessary knowledge during its instantiating and never changes its state during the entire lifecycle.

Very often, `NULL` values are used in lazy loading, to make objects incomplete and mutable. For example:

```
public class Department {
  private Employee found = null;
  public synchronized Employee manager() {
    if (this.found == null) {
```

```
      this.found = new Employee("Jeffrey");
    }
    return this.found;
  }
}
```

This technology, although widely used, is an anti-pattern in OOP. Mostly because it makes an object responsible for performance problems of the computational platform, which is something an `Employee` object should not be aware of.

Instead of managing a state and exposing its business-relevant behavior, an object has to take care of the caching of its own results — this is what lazy loading is about.

Caching is not something an employee does in the office, does he?

The solution? Don't use lazy loading in such a primitive way, as in the example above. Instead, move this caching problem to another layer of your application.

For example, in Java, you can use aspect-oriented programming aspects. For example, jcabi-aspects has `@Cacheable` annotation that caches the value returned by a method:

```
import com.jcabi.aspects.Cacheable;
public class Department {
  @Cacheable(forever = true)
  public Employee manager() {
    return new Employee("Jacky Brown");
  }
}
```

I hope this analysis was convincing enough that you will stop `NULL` -ing your code :)