DISQUS Home 9+ Inbox Discover Discuss

Community

256 yegor256.com

68 Comments • Created a month ago

Constructors Must Be Code-Free

How much work should be done within a constructor? It seems reasonable to do some computations inside a constructor and then encapsulate results. That way, when the results are required by object methods, we'll have them ready. Sound…

(yegor256.com)

68 Comments

Recommend 3 Share Sort by Newest

Join the discussion…

Джек • a day ago

An object owner should not know implementation details. So, it should not know if first() is fast or it is slow and should be cached.

Following your advice a programmer will be forced to use Cacheable/Builders and write all this boilerplate code everywhere, because he should presume that he does not know if it is fast or should be cached.

On the other side, I completely agree that a constructor should be code-free.

When having two different approaches that have their own pros/cons I usually select the one that requires to write less code.

You gave me a good idea. I will probably try to do something like:

CacheableField<string> name = new CacheableField({text.toString().split(" ", 2)[0]});

And lazy-execute and cache it with name.get()

• Reply • Share ›

Yegor Bugayenko author → Джек • a day ago

"Less code" is always a good objective, but "cleaner code" is more important. If there has to be more code, but it will be more obvious for the reader (not for the writer!), we should always to make it that way - cleaner!

• Reply • Share ›

Krzysztek • 5 days ago

Did you think about memory usage? Processing in constructor and reducing 'problem' consumes less memory than storing original objects. So it cannot be a general rule, but it's very interesting idea.

• Reply • Share ›

Yegor Bugayenko author → Krzysztek • 5 days ago

Hm... I believe that storing an object (which is technically just an 8-byte pointer) takes less memory than storing a data. Or I misunderstood your question?

• Reply • Share ›

Krzysztek → Yegor Bugayenko • 4 days ago

Storing pointer is equal to storing memory for the whole object. For example: 1) new B(new A()) - You store 2 objects (B and original A), 2) new B(new A()) - You store 1 object (B) and "side effect". PS I really like your article. Thanks.

• Reply • Share ›

Kata Tunix → Yegor Bugayenko • 5 days ago

I guess he means:

```
class A
{
    private X x;
    private Y y;
    private Z z;
    public A()
    {
        x = new X();
        if (x.error()) throw new Exception();
        // or x.doSomethingThatCanThrowException();
        // following code will not be executed thus memory is saved
        y = new Y();
        z = new Z();
    }
}
```

His constructor is heavyweight so throwing exceptions can reduce

His constructor is heavyweight so throwing exceptions can reduce "problem". I could not compare this to yours because you encourage lightweight constructors.

• Reply • Share ›

Esteban • 12 days ago

Can you please explain "But please don't make CachedName mutable and lazily loaded" a little more? How would it look like? Thanks

• Reply • Share ›

Yegor Bugayenko author Esteban • 12 days ago

Here is a mutable and lazy-loaded version of CachedName:

```
public final class CachedName implements Name {
  private final Name origin;
  private String cached = null;
  public CachedName(final Name name) {
    this.origin = name;
  }
  @Override
  public String first() {
    synchronized (this.origin) {
      if (this.cached == null) {
        this.cached = this.origin.first();
      }
      return this.cached;
    }
  }
}
```

It's terrible. See this: http://www.yegor256.com/2014/0...

• Reply • Share ›

Graham Lea Yegor Bugayenko • 6 days ago

Isn't that pretty much exactly what Cache does internally? You've just implemented memoization, which doesn't make the class mutable, because it still can't be changed. If the above is mutable then so is you cache version, irrespective of the fact that all the fields a final.

• Reply • Share ›

Yegor Bugayenko author Graham Lea • 6 days ago

That's pretty much exactly what all in-memory caches do, but the presence of mutable properties and NULL make this code extremely un-desirable in a high-quality application, see for example this: http://www.yegor256.com/2014/0...

• Reply • Share ›

Suseika • 18 days ago

Hi Yegor.

This article had a great impact on me, but I think that the approach with @Cacheable annotation is not quite good. Using an external cache that is managed by "flushing" is not a very elegant approach. So, inspired by the idea of behavior being over storage details, I implemented an extension for Project Lombok that generates the right bytecode for storing the method result inside the object itself:

https://github.com/rzwitserloo...
(source code, usage example and equivalent plain Java code)

During compilation, it creates a private field in the class and modifies the method to save method's result to the field on the first invocation, and then retrieves the content of that field on subsequent invocations. I think this is a much better approach than the Cacheable annotation, let alone the explicit code for lazy initialization. The main advantage is that the cached result is garbage-collected together with the object whose behavior it represents.

I'd like to know your opinion, what do you think? Would you use it in your projects? Do you think that Project Lombok is a more cumbersome dependency than AspectJ?

• Reply • Share ›

Yegor Bugayenko author Suseika • 18 days ago

I think it's a very good idea. In general, Lombok is a more effective approach than AspectJ, I believe. +1 for the pull request.

• Reply • Share ›

David Johnston • 25 days ago

The constructor should not take a string to be parsed. It should have one argument for each corresponding field. Create a static construction method (not constructor) on the class, with an appropriate name for wht it is doing, that takes a string to parse, parses it, then passes the parsed fields into the constructor.

• Reply • Share ›

Kata Tunix David Johnston • 24 days ago

So, your class is nothing but an util class and it's easy to violate Single Responsibility as it will handle English, French, German ... names also.

And your object is nothing but a data holder and not composable.

This is not OOP.

3 • Reply • Share ›

David Johnston Kata Tunix • 24 days ago

I didn't choose the example...but given such this is the best implementation. Each language should indeed have its own class and set of static construction methods if such logic is desirable for creating immutable instances.

Nothing about the current example prevents it from being used for language specific names anyway...and indeed is can be a component in another object quite easily.

A name is an attribute, not an object, so it makes no sense to try make it so.

• Reply • Share ›

Riccardo Cardin • 25 days ago

@Yegor Bugayenko, I believe that in some circumstances your "lazy initialization" approach could bring to some advantages. Take as an example "infinite streams" or the "pass by reference" tecnique of Scala. Also, in cases where the object to create is very heavy your approach is a good idea. Take as an example the "virtual proxy" pattern of GoF. But in the majority of cases it is not a good idea to defer the execution of code that creates an object. Doing so, you are violating the *FAIL FAST* principle, that is one of the basic principles of programming.

1 • Reply • Share ›

David Raab → Riccardo Cardin • 25 days ago

> Also, in cases where the object to create is very heavy your approach is a good idea.

It depends what you do. In a game you never want to have any "heavy" things lazy loaded. You always want that everything is pre-calculated what can be pre-calculated. To minimize the amount that is done at runtime. It is a killer to do any heavy things at runtime.

1 • Reply • Share ›

Riccardo Cardin → David Raab • 24 days ago

> It is a killer to do any heavy things at runtime.

Wait a minute: where have I said that yegor's idea is always appliable?

• Reply • Share ›

David Raab → Riccardo Cardin • 24 days ago

Where did i say that you said it? I just added that "if it is a good idea" depends on the situation.

• Reply • Share ›

Yegor Bugayenko author → Riccardo Cardin • 25 days ago

I got your point, but I disagree. I think that we should **not** fail during object costruction, but only during execution. Or, in other words, only when we ask our objects to expose some behavior, they have a right to fail.

2 • Reply • Share ›

Marcos Douglas Santos → Yegor Bugayenko • 24 days ago

I agree.

Objects mean behavior. If is need to validate some critical data before continues, ask for another object to validate (ValidatedName) using a method (behavior) for this. If data Ok, pass the data (name) for the EnglishName and everything will be good. But if you (app) do not need know if the name is Ok right now, EnglishName is good to do the job. He can use ValidatedName object inside itself for this too. But the important thing is EnglishName will validate when he want.

PS: You need to do what you saying, for example, here you throw exceptions in constructors:
https://github.com/yegor256/ta...
https://github.com/yegor256/ta...

2 ^ | ˅ • Reply • Share ›

Riccardo Cardin ➦ Marcos Douglas Santos • 23 days ago

What you're trying to describe is a debug-hell application. And it also violates the FAIL FAST principle. Also, if a person class is made of a name and a surname, why the hell you want to allow to build a person without that attributes setted?!

^ | ˅ • Reply • Share ›

Marcos Douglas Santos ➦ Riccardo Cardin • 23 days ago

Will fail fast when you need the behavior.
IMHO you are thinking in data, not objects.

^ | ˅ • Reply • Share ›

David Raab ➦ Marcos Douglas Santos • 22 days ago

"when you need the behaviour" is not fail fast, it is fail slowly. Because it delays the failing at the last point it is even possible to fail. "Fail Fast" would mean to fail as soon as you knew something is invalid.

^ | ˅ • Reply • Share ›

Marcos Douglas Santos ➦ David Raab • 22 days ago

I understand what is fail fast and I consider this just one more concept. I do not think one them is wrong, I'm just considering one approach better that another, for most often.

^ | ˅ • Reply • Share ›

Riccardo Cardin ➦ Marcos Douglas Santos • 23 days ago

Clearly, I will you use fail fast if it was possible and if it was a convenient approach. But we are returning to the main point. The approach suggested by Yegor is not wrong in the principles. The wrong side is the idea to apply it for all objects.

1 ^ | ˅ • Reply • Share ›

Marcos Douglas Santos ➦ Riccardo Cardin • 22 days ago

Ok, maybe not for all objects but consider do not fail in the constructors for most objects.

• Reply • Share ›

David Raab Marcos Douglas Santos • 23 days ago

So, you (and Yegor) also think that

```
int x = "foo"
```

is good when it don't throw an error? What you basically want then is a dynamically typed language.

1 • Reply • Share ›

Marcos Douglas Santos David Raab • 23 days ago

The data type wont break (using EnglishName example).
We talking about only about data, not data type... right?

• Reply • Share ›

David Raab Marcos Douglas Santos • 22 days ago

"int" is a type that should only allow integers. "foo" is not an integer so it should fail. The class "EnglishName" is a newly created type. Usually any new type has some logic applied to it when it is valid or not. You often use other more generic types in the implementation. Like a "string" is used to save the EnglishName. So the EnglishName "will not break", but that doesn't mean it is a valid EnglishName. And also, you shouldn't care on how it is implemented. It is an EnglishName not a string. So that it doesn't break is not something positive. It is negative that it doesn't break. If it is not a Valid EnglishName the constructor should directly fail.

Allowing invalid EnglishName is just a really bad practice, that is comparable to null. And in some other posts down i described already why it is the same as null.

And by the way, it even violates basic object-orientation. One main point of object-orientation is the ability to hide implementation details, especially state. That is important so users can't modify critical state and creates invalid objects. For example you also cannot modify the "Length" / "Count" attribute of an list (however the attribute is named in the particular language). Because if you could do that you would be able to create invalid objects if you could set the size for example to 5, while it only holds 3 elements. The main point of OO is to only allow valid states. So being able to create invalid EnglishName does not even violates basic OO (i don't think that is purely an argument, but that is often used by Yegor as a killer argument if he don't have any arguments), it even violates the whole purpose of having a type system.

What he suggest is a "fail slowly" approach. It doesn't fail as fast as he can. Actually already in the constructor you knew when something is valid or not. Allowing something invalid to pretend it is an EnglishName while it is not an EnglishName makes absolutely no sense at all. Failing just when you need it just increases the amount you have to code/validate. Also for this one i give an explanation below.

So what he suggest is the same as the language would allow

> ```
> int x = "foo"
> ```

and that code would not break. When would it break? Only when you start using x, for example in arithmetic calculation or trying to print it. But that again is Failing Slowly. If it is not a valid int then you already knew that every operation will fail. There is no point of delaying the failing. It just means that every time you use EnglishName you not only have to check if it is not null. You also have to check if it is really a valid EnglishName. And that is just stupid.

if it is not a valid EnglishName, then it should not pretend to be an EnglishName. It is the same as "foo" that not fails and pretend to be a valid "int" as long as you don't start to use it.

P.S.: And as a side-note. Data and Data Types are linked. Data without a type doesn't really exists. Everything is of some sort of a type.

1 • Reply • Share ›

Marcos Douglas Santos David Raab • 22 days ago

> [...] If it is not a Valid EnglishName the constructor should directly fail.

Your opinion.

> Allowing invalid EnglishName is just a really bad practice, that is comparable to null. And in some other posts down i described already why it is the same as null.

Not the same. NULL is an invalid pointer until EnglishName is a valid object with a valid behavior but, for your system, maybe this object wont returns a valid data... maybe.

> And by the way, it even violates basic object-orientation. One main point of object-orientation is the ability to hide implementation details, especially state. That is important so users can't modify critical state and creates invalid objects. For example you also cannot modify the "Length" / "Count" attribute of an list (however the attribute is named in the particular language). Because if you could do that you would be able to create invalid objects if you

> could set the size for example to 5, while it only holds 3 elements. The main point of OO is to only allow valid states. So being able to create invalid EnglishName does not even violates basic OO (i don't think that is purely an argument, but that is often used by Yegor as a killer argument if he don't have any arguments), it even violates the whole purpose of having a type system.

This is not a violates the basic object-orientation.
You cannot modify the "Length" / "Count" attribute because they are state, not data. If you modify Count you will raise an exeption, but this not occurs in EnglishName object.

> What he suggest is a "fail slowly" approach. It doesn't fail as fast as he can. Actually already in the constructor you knew when something is valid or not. Allowing something invalid to pretend it is an EnglishName while it is not an EnglishName makes absolutely no sense at all. Failing just when you need it just increases the amount you have to code/validate. Also for this one i give an explanation below.

Objects could be composable so, if you will need to fail fast, do it as I said before.

> So what he suggest is the same as the language would allow

> ```
> int x = "foo"
> ```

I did not suggest this. You can't put "foo" inside a int type. But a "123456" is a text/string... so I can use it in a argument string type.

> You also have to check if it is really a valid EnglishName. And that is just stupid.

I will check nothing, the object will do, this is not a NULL, he has behavior not a NULL pointer. But if you think this is stupid because you have your own true that will work ever, what can I say...

• Reply • Share ›

David Raab ➦ Marcos Douglas Santos • 22 days ago

> Your opinion.

Yes, but i also gave some arguments why that is my opinion.

http://www.yegor256.com/2015/0...
http://www.yegor256.com/2015/0...

I don't really see arguments from you or Yegor. Just an "don't do it". But sadly from Yegor i didn't expect anything of that sort anymore. He has some great old articles. But all his new articles has some attitude like: You believe me, or you are dump.

> Not the same. NULL is an invalid pointer until EnglishName is a valid object with a valid behavior but, for your system, maybe this object wont returns a valid data... maybe.

You comparing it on a too technically level. Technically there are different. Logically they are the same. NULL can pretend to be an EnglishName that isn't really an EnglishName. And if you follow what Yegor suggest you even have an EnglishName that is not an EnglishName but pretends to be an EnglishName.

> You cannot modify the "Length" / "Count" attribute because they are state, not data. If you modify Count you will raise an exeption, but this not occurs in EnglishName object.

You are just playing with words here. And on typically List you also can't modify Count, you should not even have the ability to change it. You only can read it, so it can't even throw an Exception. And in an Immutable object you also can't change the name, because it is immutable. And if an exception occurs in EnglishName is just a matter of an implementation.

> Objects could be composable so, if you will need to fail fast, do it as I said before.

Composability of objects don't change if you do calculation in the constructor or in some methods. If that is what you wanted to say.

> I did not suggest this. You can't put "foo" inside a int type. But a "123456" is a text/string... so I can use it in a argument string type.

You don't really have an object thinking here. If you have an "EnglishName" class then your type is an EnglishName not a string. It internally can use a string as his representation and can accept a string in the constructor, but an EnglishName is not a String. A Constructor is something that creates something new. A constructor takes a "string" and returns you an EnglishName.

Sure you can put "123456" in a string and you can use a string as an internal design to save an EnglishName. But Is "123456" an EnglishName? No its not. You also can use a string as the internal representation. But does it makes sense that "123456" can pretend to be a valid URL? Absolutely not. When you create a "new URL()" or "new EnglishName()" you create URL and EnglishName as types, not just strings.

You completely miss the point of why you create a class in the first place. It is logically the exact same to provide an "foo" to an int. Because an "foo" cannot be converted to an int. It is invalid. "123456" cannot be converted to an EnglishName. So it is invalid. And if you need other proper examples.

string -> URL

int -> Kelvin

If you create a new URL you also provide it as a string, but when you get an URL back it is an URL, not a string anymore. So "123456" should fail, because it is not an URL. Even if it still uses a string as it internal representation. And if you create a Kelvin class then it converts an int to a Kelvin. Can every int be a Kelvin? No, because Kelvin can only be positive. So a "new Kelvin(-10)" don't make any sense at all.

That you believe that passing "123456" to an EnglishName it is okay because the string does not complain. Yes for the string it is okay. But it is not okay for being an EnglishName. You miss the whole point why you create a class.

And that is the whole point of object-orientation. If you don't add validation to it, then you are just creating procedural functions. Just do an

```
string EnglishName(string x)
```

and its done. And you also have to do validation in every function, because you just have a string. But as soon as you create a class. You have a new type named "EnglishName", you don't just have a string anymore. And that objects needs proper validation. And sure. Don't allowing an invalid state is one main point of object-orientation. That is the whole purpose of the existence of "encapsulation". If it would be okay to create invalid objects in the first place you would never need encapsulation in the first place. Just make everything public, and just allow that everybody can change everything. And as you say, if someone invalidates the object, well in your opinion or Yegor opinion that is totaly fine.

> I will check nothing, the object will do

You are the one that has to write the class. So you are also doing to write the validation. And no, you even have to do the validation if you get it back from an API, even if you did not write the whole class at all. Because if you get "EnglishName" back, well sometimes it just pretends to be an EnglishName even if it is not one. So you have to do.

1) Check if you don't get null back
2) Check if the EnglishName is a valid EnglishName

and that is the reason why the idea is exactly the same as null. You now have to check if what you get back was a valid EnglishName. And you have to do it twice!

1 ^ | ˅ • Reply • Share ›

Marcos Douglas Santos ➦ David Raab • 22 days ago

> I don't really see arguments from you or Yegor. Just an "don't do it". But sadly from Yegor i didn't expect anything of that sort anymore. He has some great old articles. But all his new articles has some attitude like: You believe me, or you are dump.

I'm not Yegor. I'm just expose my thoughts about OOP. Works to me so, I'm sharing.
Nobody is wrong, just using different style. If raise an exception on constructor is good for you (and many), that's Ok. If you ask me if I never used this approach, I tell you of course I used before. Depends the case. But I think we talking more about theory, so in that case I think is better do not raise an exception...

> You comparing it on a too technically level. Technically there are different. Logically they are the same. NULL can pretend to be an EnglishName that isn't really an EnglishName. And if you follow what Yegor suggest you even have an EnglishName that is not an EnglishName but pretends to be an EnglishName.

No! NULL haven't behavior, EnglishName has. That's the difference!
If EnglishName = "12345" pretends to be an "english name", NULL is NOTHING.
In that case you wont convince me, sorry. NULL is nothing. You can't compare with an object even an invalid object.

(...)

> Composability of objects don't change if you do calculation in the constructor or in some methods. If that is what you wanted to say.

I know. I meant if you need to raise an exception when you read the data (following the example EnglishName) you can compose two objects, something like:

```
v = new ValidedNames.add(new EnglishName("12345")).check();
```

I mean: you do not need to test every object one-by-one. You could create something more automatic, adding objects in that list programmatically at runtime.

(...)

So, all your words that I cut above I can't answer one by one because you confused everything. You confused about OOP, data, encapsulation...

Now I ask you: If an EnglishName needs your data perfectly on the constructor, how could you test all fields of a form, to persist, using a simple CRUD? **You will raise an exception for every bad english name typed by for user?**

> ...and that is the reason why the idea is exactly the same as null. You now have to check if what you get back was a valid EnglishName. And you have to do it twice!

No, its not and doesn't need validate twice.

• Reply • Share ›

David Raab ➦ Marcos Douglas Santos • 21 days ago

> Nobody is wrong, just using different style. If raise an exception on constructor is good for you (and many), that's Ok. If you ask me if I never used this approach, I tell you of course I used before. Depends the case. But I think we talking more about theory, so in that case I think is better do not raise an exception...

Yes all is theory. And you are saying you think it is better to not raise an exception. Why? In a Discussion you usually have to explain something. Not just saying "Don't do it". And i already provided some examples. So for example when you not throw an Exception at construction time, it now means you have to check again if the thing that was provided was valid.

```
void SomeMethod(EnglishName name) {
    if ( name == null ) {
        throw new Exception("name cannot be null");
    }
    if ( !EnglishName.IsValid(name) ) {
        throw new Exception("name is not a valid EglishName");
    }
}
```

So your code bloats even more. And why do you think it makes sense that something can pretend to be an EnglishName if it is not an EnglishName? It even violates the type system. And null is a failure in itself and already does that. We don't really need another thing to violate the type system.

> No! NULL haven't behavior, EnglishName has.

An EnglishName that is not valid and throws Exceptions on every method also has no behavior. That is why they are the same. Both has no behavior. EnglishName could only have a behavior if it is at some point level valid. It doesn't mean by the way that some methods can throw exceptions. I really talking here about completely invalid things.

If i have a URL class there is no point of allowing "123456" to be an URL. At first it violates basic OO like already said. "123456" is not an URL, so it should not pretend to be an URL. But you also can't do *anything* with it. You cannot fetch the URL, because it is not an URL. There is nothing to fetch, as soon as someone wants to do *anything* with it, it has to fail. So explain to me why you think it is useful that somebody can create objects that can do nothing, like null. Or why is it useful that something can pretend to be an URL if it is not an URL?

> I know. I meant if you need to raise an exception when you read the data (following the example EnglishName) you can compose two objects, something like:
>
> v = new ValidedNames.add(new EnglishName("12345")).check();

That is completely redundant for me. If i have a type EnglishName i already expect it to be a valid EnglishName. A "ValidEnglishName" class makes no sense to me. It is like you have a "Male" and a "Female" class. And then you create a "ValidFemale" class to really check if a Female is really a Female. What the hell?

What you propose is by the way exactly what "null" already does. Because we already have to check if something is really a EnglishName and not null. "null" is already that abnormal thing that pretends to be an EnglishName and you can't do nothing with it, only to fail. And that is why i'm saying it is the same null. Now you have to check if it is not null and is really the thing it says it is.

If i have a type system. Then it should be clear. When something says it is an "int". It should only be an int. Not a "null" and not some "invalid int". If i have an EnglishName it should be an EnglishName. And not some invalid EnglishName that only can crash the program as soon as i try to use it somehow. null already that worse things. And null alone is already one too much.

> So, all your words that I cut above I can't answer one by one

> because you confused everything. You confused about OOP, data, encapsulation...

I don't really confused anything. One main purpose of OO is to encapsulate state, to make sure that an object is always in a valid state. If you think it makes sense to create something invalid. It makes in general no sense, not even in an OO world.

> If an EnglishName needs your data perfectly on the constructor, how could you test all fields of a form, to persist, using a simple CRUD? You will raise an exception for every bad english name typed by for user?

Raising an exception is one way. You also could use others. But yeah. You cannot create an EnglishName if it was invalid. And what does that mean? You have to provide your user an error that what he entered was invalid. What else should you do?

So in your opinion it makes sense to even allow the user to input something invalid. And just accept that data. And as long as you don't need it, you accept even the invalid? So for example you somewhere want an email address. You even accept an invalid email address in the form. And then, what happens if you later want to send an email to the user? Then you fail there, and how do you correct it? The user is already gone. How do you contact the user that he put in some invalid email?

It makes no sense at all to ever allow invalid data to enter the system. As soon as you knew something is invalid you don't accept it.

> No, its not and doesn't need validate twice.

Sure, i provided code for it. Once again. If you allow some invalid objects, your code change to something like that.

```
void SomeMethod(EnglishName name) {
    if ( name == null ) {
        throw new Exception("name cannot be null");
    }
    if ( !EnglishName.IsValid(name) ) {
        throw new Exception("name is not a valid EglishName");
    }
}
```

Code don't lie. You have to check it twice.

1 ^ | ˅ • Reply • Share ›

Riccardo Cardin ➦ Yegor Bugayenko • 24 days ago

I think the point here is that the approach you're suggesting is not wrong at

I think the point here is that the approach you're suggesting is not wrong at the basis. The thing that is less comprehensible is that you're saying that your approach has to be applied for *every object*. Having lazy evaluation applied to every object brings to a code that is hard to debug and to maintain, in my opinion. Putting the initialization code in methods that should have other responsibilities according to their name is not correct.

• Reply • Share ›

Fabrício Cabral Yegor Bugayenko • 25 days ago

So, in your opinion, a constructor never should throws an exception? And the basic OOP that says: "I must not permit create an object with invalid state"? For example, if a programmer did Name n = new EnglishName("") (a name cannot be an empty string), the EnglishName's constructor must not throws an exception? How would you do it?

3 • Edit • Reply • Share ›

Yegor Bugayenko author Fabrício Cabral • 24 days ago

Yes, I think that input validation should **not** happen in ctor. Even if you pass NULL as a single argument to `new EnglishName()`, it should not complain. If later, you never call its `first()` method, you will never know that the object you created was not "complete". It was a perfect employee for "doing nothing" job. But it's an incompetent employee for "fetching first name" job. See the point? Unless you give me a task, you will never know how good am I as an employee. Same here.

• Reply • Share ›

Fabrício Cabral Yegor Bugayenko • 24 days ago

I see your point of view, but I disagree. If you create an object whose state is invalid you will, at least, waste memory space. On the other hand, if you valid your object before create it, this problem does not occur. For exemple, if you do `Name n = new EnglishName("123456789");` it will waste memory space, because you just will detect it is a invalid object only when call its `first()` method. Using your same example: I don't need give a task to an employee to know if an employee is good or not. I can check its curriculum to know if it has the necessary skills to do the job. It makes senses to you?

1 • Edit • Reply • Share ›

Yegor Bugayenko author Fabrício Cabral • 24 days ago

But for some job that `EnglishName("123456789")` was good enough, right? This "employee" can't code in Java, but he can make good coffee. Why not giving him a chance to show his coffee making skills while we don't have any Java coding tasks? :)

• Reply • Share ›

Fabrício Cabral Yegor Bugayenko • 24 days ago

Because:

1) its violates the Single Responsibility Principle. EnglishName "employee" must do not more than one thing, right? It makes a good coffee or a good Java code;

2) If you want to a English Name, but its state is invalid, is more likely to fail when executing its main functionality than executing a peripheral functionality. So you will waste space memory just to "maybe" use some other functionality of the created object.

Does it make sense?

• Edit • Reply • Share ›

David Raab Fabrício Cabral • 23 days ago

I would not argue about memory. Todays computers have so much memory, even if it wastes memory it is so less, it isn't even worth it to mention. As long as something don't have memory leaks you just can ignore some bytes or less. But the main point why it is a bad idea to not validate it in the constructor is, it makes types completely useless. You just just increase the error rate of a program and the amount you have to code, and once again, you don't get nothing positive out of it.

And we already have one thing that violates that principle. It is "null". Why is null so bad? Because "null" can be everything. Everything can be null, without even asking, so you also need to check everywhere if it is null. Imagine null don't would exists, you could write code just like this, with some imaginary API

```
String version = computer.getSoundcard().getUSB().getVersion();
```

But because null exists. And silently every method can be null. You have to write something like this if not want that a NullReferenceException crashes your whole application.

```
String version = "UNKNOWN";
if(computer != null) {
    Soundcard soundcard = computer.getSoundcard();
    if(soundcard != null) {
        USB usb = soundcard.getUSB();
        if(usb != null) {
            version = usb.getVersion();
        }
    }
}
```

It is just an ugly mess. You multiplied your code. Now the code is also not declarative anymore. Because most of your code have to detail for failures that the language allow. Most of the code handles things from the "language space" not from the "problem space".

"null" is really a bad thing, because you are forced to check first everywhere if what you have is really an int, string or whatever you expected. You cannot just write

```
void SomeMethod(int x) {
    // Work with x
}
```

no, because x also can be null, to be safe you need to write

```
void SomeMethod(int x) {
    if ( x == null ) {
        throw nex Exception("x cannot be null")
    }
}
```

or some other error handling, because a user provided something that was not an "int" even we wanted an int. The same is true for other classes. If you write a method that want a "Person" object, you first have to check whether it is really a Person and not null. Already this mistake in history bloates codes and makes it unreadably like hell. The idea that something can pretend to of type X but is not really X is just a completely design-flaw.

Even the inventor of null says it is his multi-billion dollar mistake in history. And now Yegor suggest new multi-billion dollar mistakes.

The idea that something can pretend for example being a "EnglishName" but is not really an "EnglishName" is just the same as null. Now instead of just checking for "null". Someone has to additional test if what was passed in, is even valid. So it bloates the code even more. Now someone has to write.

```
void SomeMethod(EnglishName name) {
    if ( name == null ) {
        throw new Exception("name cannot be null");
    }
    if ( !EnglishName.IsValid(name) ) {
        throw new Exception("name is not a valid EglishName");
    }
}
```

and you have to do that test every time you want to access something on EnglishName. Or either it just throws you an exception saying it can't do anything because the thing was not valid in the first place. But if it was not valid, why was it possible to create it in the first place? The idea is really simple. If something pretends to be an "EnglishName", it also has to be an "EnglishName".

That is the whole point behind types. If i say something has to be an "int" i already defining it at that moment that something has to be an int. it is just mind-blowing that i have to test once again in a method if it really was an int. I already have to do that in languages like Java/C# because there exists something like null. But with the idea of Yegor you now even have to further increase the validation. now you have to test if it isn't null, and again test if it really was an int. And the best thing: You have to do it everywhere in your code. If you write three methods that wants an "EnglishName", you have to test it in three cases if it really is an "EnglishName".

You just repeat boiler-code over and over again. A sane solution would be when "null" doesn't exists, and something only can be pretend of being of some type if it really is a valid type of those. The first thing, you can't really fix in languages like Java/C# and so on. Well with Optional in Java 8 you can make it a little better. But the last thing you can fix easily. Just throw errors at construction time if you already knew at this point something is invalid. If something is not an "EnglishName", don't pretend it is an EnglishName.

If someone writes a method and expect that an argument has to be of the type "EnglishName". Someone should not test again if the thing that was passed as an argument was really an EnglishName. If someone has to do that, it makes typing completely useless.

But well, as we all know, nobody learns from history and it gets repeated. Now Yegor suggest a more evil version on top of null.

2 ^ | ⌄ • Reply • Share ›

Kata Tunix ➦ Fabrício Cabral • 23 days ago

Having more than one jobs/methods doesnt mean violating SR. As long as these jobs have only one reason to change, they still satisfy SR (said by Uncle Bob). Coding Java and making coffee are just examples. Dont focus too much on this.

Saving some bytes/KB is not critical. As long as there is no memory leaking, dont focus too much on this.

So, throwing exceptions in constructors or not? It's up to you. If you feel comfortable with your approach, then do it.

But the reason I like Yegor's approach is his object thinking. He loves and respects his objects. He treats objects like living creatures. Think about your children just before they were born. If you knew they are abnormal, would you kill them immediately? Or let the children be born and hope that they are still useful. In case you are sure 100% your children will never be useful, throwing exceptions in constructors might be reasonable.

2 ^ | ⌄ • Reply • Share ›

Kata Tunix ➦ Fabrício Cabral • 25 days ago

Thats why Yegor said that constructors should be lightweight. Containing only assigments is very good.

Hence, in your example, passing "" or null to the constructor is okay. Exceptions will be thrown when calling first().

1 ^ | ⌄ • Reply • Share ›

Fabrício Cabral ➦ Kata Tunix • 25 days ago

But we still have an object with invalid state. Searching in the internet, I found this page: http://stackoverflow.com/quest..., that explain about two schools: one is about two-stage object construction. Another is about one-stage object construction. I think one-stage object construction makes more sense (and agree with the other reasons explained in the link) than two-stage object construct approach.

^ | ⌄ • Edit • Reply • Share ›

David Raab ➦ Fabrício Cabral • 23 days ago

What is described here has nothing to do with one-stage or two-stage. two-stage is something like.

```
var x = new X();
x.setFoo("Foo");
x.setBar("Bar");
```

Well, i think that design has so many flaws i will not even mention it here. But if you believe in immutability you only create one-stage classes. That means everything is set-up at construction time. Setters don't exists in an immutable class, because that would mean the object itself can change. And you wouldn't have an immutable object in the first-place.

You can have some similar API in an immutable world, but then you don't have setters, they are methods that returns something new, with a change applied. For example something like

```
var x = X.empty.foo("Foo").bar("bar");
```

The difference is. In the two-stage thing you allow to start with something invalid. Something that pretends to be an X, but at that point is not an X. The API expects you that you have to set additional things until the object gets valid.

In the one-stage thing, you always start with something valid. For

In the one-stage thing, you always start with something valid. For example even "X.empty" needs to be a valid thing. You for example see this API in functional languages or in immutable Lists. For example in C# you could do.

```
var list = ImmutableList<something>.Empty.Add(x).Add(y);
```

But even "Empty" is valid. But at this point, it is just an Empty list. If you believe in an immutable design. Not everywhere it is possible to create something like an "Empty". But then your constructor should expect all things that are needed in the constructor, and only if all of that are valid returns you a valid object of the specified type. You later can still provide methods to change things. For example assume a HTTPRequest API like that

```
var request = HTTPRequest("http://slashdot.com");
var postRequest = request.Method("POST");
```

So your HTTPRequest class expect a valid URL in the constructor. If it is not valid you fail. The Request Method has a default of "GET", but with "Method" you can create a new HTTPRequest object that returns you a POST Request with the same url.

If something is not valid, it directly fails. For example if the string provided was not a valid URL it fails. When the Request Method that was provided doesn't exists it fails, and so on.

That is a clean immutable design that reduces the amount you have to code a lot, and it makes for a sane API. For example it makes no sense to allow

```
var request = HTTPRequest("http://slashdot.com");
var newRequest = request.Method("FooBar");
...
// Somewhere else
var content = newRequest.fetch();
```

Such a design means, you have to validate everything in fetch(). For example you have to validate if the Request Method was valid in the fetch(). But not only at fetch(). At every method you need a validation. If you have a ".head()" method that just returns the HTTP Header of the request you need to validate the URL/Method again. You also get an error at the wrong place. Now your fetch() or head() returns an error. A sane solution would be that ".Method("FooBar")" already complain that "FooBar" is not a valid Method and it didn't even let you allow to create an object with invalid state.

So a clean immutable design is. One-stage initialization, and validating everything in the constructor so you always have valid objects. Because it is immutable it also means the constructor need at least every attributes that are needed to form a valid object. two-stage doesn't exists in an immutable world because you don't change an object itself.

Well if you have people that don't do this. I only can say. Somewhere that bad programmers must exists that everybody is talking about. If they think it makes sense so create objects with invalid state then let them create such objects. As long as you knew what makes sense and you don't have to code with such people, i think everything is fine.

1 ^ | ⌄ • Reply • Share ›

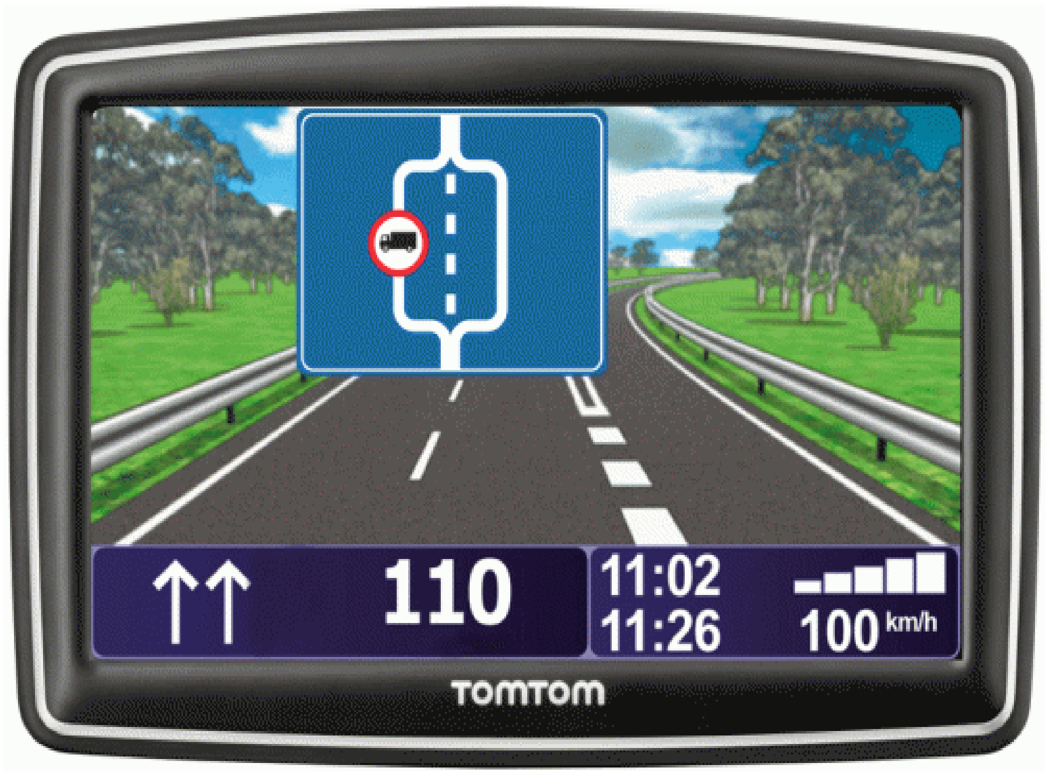
David Raab ➦ Yegor Bugayenko • 25 days ago

This is the code I could never understand:

```
X x = new X("test")
try {
    x.method()
}
catch {
    System.out.println("x is invalid");
}
```

I have been trying to find a proper metaphor to explain its incorrectness. Today I finally found it.

try-catch is a forking mechanism of procedural programming. The CPU either goes to the left and then does something or goes to the right and does something else. Imagine yourself driving a car and seeing this sign:

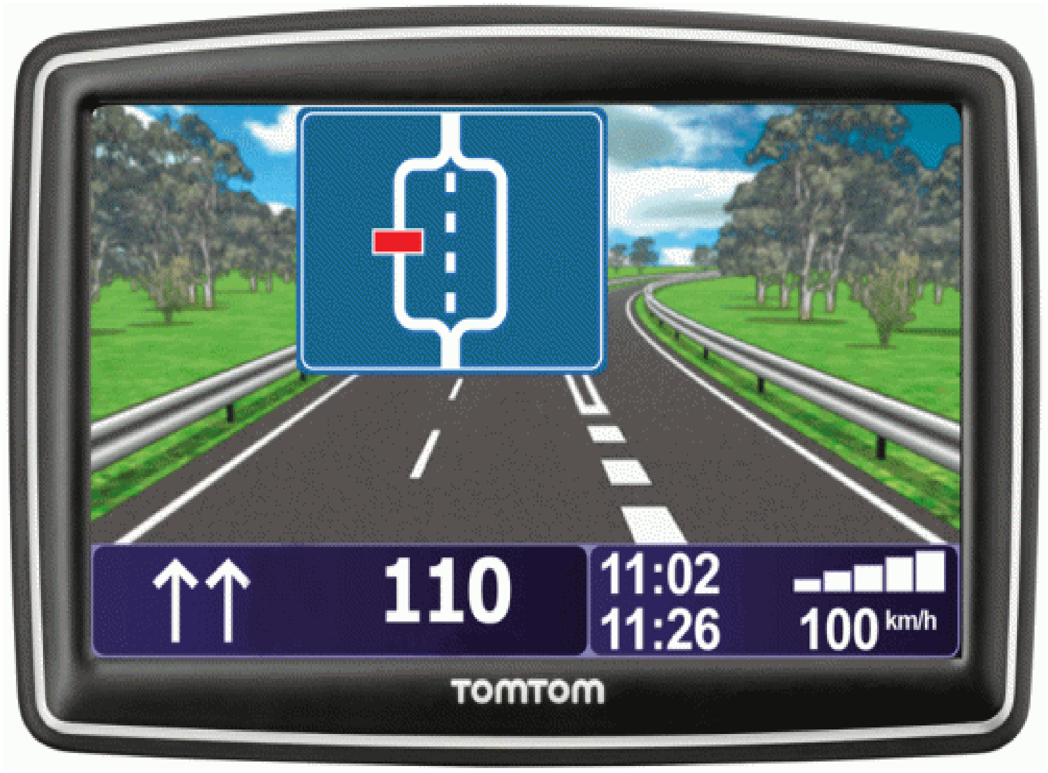


It looks logical, doesn't it? You can go in the left lane if you're not driving a

It looks logical, doesn't it? You can go in the left lane if you're not driving a truck. Otherwise you should go in the right lane. Both lanes meet up in a while. No matter which one you choose, you will end up on the same road. This is what this code block does:

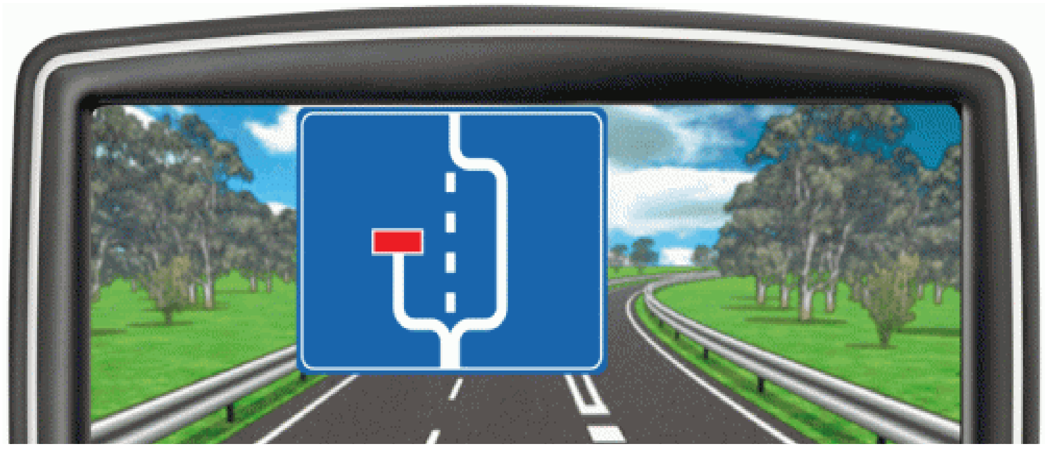
```
X x = new X("test");
if ( some-statement ) {
    x.method1();
}
else {
    x.method2();
}
```

Now, try to imagine this sign:



It looks very strange to me, and you will never see this sign anywhere simply because a dead end means an end, a full stop, a finish. What is the point of drawing a lane after the dead end sign? There is no point.

This is how a proper sign would look:



This is how a proper code block would look:

```
try {
    X x = new X("foo");
}
catch {
    System.out.println("X is invalid");
}
```

The same is true for loops. This is wrong:

```
for (X x : xs) {
    try {
        x.method();
    }
    catch {
        System.out.println("x is invalid");
    }
}
```

While this is right:

```
// xs only contains valid X, they already fail at construction time when
for (X x : xs) {
    x.method() // Only constructor throws exception, so we don't need tr
}
```

There is no road after the dead end! If you draw it, your code looks like this very funny snippet I see all over the place by a lot of some very well-paid developer in nearly every serious company:

```
x.method(); // Throws an exception because x was not a valid X and user
```

Don't do this.

Modified from: http://www.yegor256.com/2015/0...

1 • Reply • Share ›

Oleg Majewski David Raab • 24 days ago

Code says more then 1000 words:

```
public List<foo> getFoo(long id) {
    String sql = "SELECT name FROM for WHERE id = ?";
    List<foo> listOfFoos = new ArrayList<>();
    try (Connection con = DriverManager.getConnection(url);
         PreparedStatement ps = con.prepareStatement(sql);) {
        ps.setLong(1, id);
        try (ResultSet rs = ps.executeQuery();) {
            while(rs.next()) {
                listOfFoos.add(new Foo(rs.getString("name")));
            }
        }
    } catch (SQLException e) {
        // logs + metrics
        // it is NOT thrown because something is invalid, like ir
        // BUT because you lost network connection in PROD
    }
    return listOfFoos;
}
```

• Reply • Share ›

Kata Tunix • 25 days ago

Thanks for your post. I got your idea.

Let me add another benefit for your design: when state of input objects (the CharSequence in your example) are changed, doing everything in the constructor will not give us an up-to-date output (the first() in your example). Note that: "state changed" doesn't mean the object is mutable, it means each time we call a method of the object, we may get a different value.

However, I still have a concern. In case the interface Name has two more methods: middle(), last(); all of these information can be also extracted from the CharSequence. Assume that split() is very slow. Now I want to call split() only one time, not three times (even with the CachedName class, we still need to call split() three times). How is your design in this situation?

1 • Reply • Share ›

Yegor Bugayenko author → Kata Tunix • 24 days ago

In that case, you implement an internal cache inside the object. I will probably write an article about it soon, but I'm sure you got the idea.

1 • Reply • Share ›

David Raab → Kata Tunix • 25 days ago

> when state of input objects (the CharSequence in your example) are changed, doing everything in the constructor will not give us an up-to-date output

Objects should be immutable. And that is also what Yegor always suggest. So there never happens that something "changes". That is exactly the reason why you already can do it in the constructor. The idea of not doing it in the constructor only makes sense with "mutable objects".

> Note that: "state changed" doesn't mean the object is mutable, it means each time we call a method of the object, we may get a different value.

Ehm sure that means it. When you call a method and every time some state changed and you also always get different values back, then you have a mutable object. An Immutable object doesn't have any side-effects, never changes and always returns the same values from a method.

You can always a simple check if something is immutable or not. Just think that you have two reference to an object at some different spots in your code. And assume the objects get used a lot. Even if the first spot in your code calls a method it always have to return the same. No matter what other code do or call on the object or pass the object to something. Or whenever in time your call happens. Only if those restrictions are true, you have an immutable object.

If you have an object that now can return something, and if you call it tomorrow and it returns something else, even if none of its internal state changed, then it is not an immutable object.

There exists some exceptions to this. For example caching is one. With every call to a function a cache can be mutated. But if the function still returns the same for every input and returns the same input you can still consider it immutable from the outside. It is mutable from the inside, but behaves immutable from the outside.

• Reply • Share ›

Oleg Majewski • a month ago

like :)

• Reply • Share ›

David Raab • a month ago

So much wrong in this article, i don't even knew where to start.

> Before I start proving, though, let me ask you to read this article:Composable Decorators vs. Imperative Utility Methods

I also can highlight my comment there where i prove that everything you said there is wrong.

> It explains the difference between a static method and composable decorators. The first snippet above is very close to an imperative utility method, even though it looks like an object. The second example is a true object.

Both are not object-oriented and not true objects. There are just boilerplate-code for static methods.

> In the first example, we are abusing the new operator and turning it into a static method, which does all calculations for us right here and now. This is what imperative programming is about. In imperative programming, we do all calculations right now and return fully ready results. In declarative programming, we are instead trying to delay calculations for as long as possible.

Nothing what you said is by far true. Imperative is a programing style where you describe how things are done step-by-step. Declarative is when the code just describes what to do, without a step-by-step code. What you here describes is laziness and has absolutely nothing to do with declarative or imperative. It doesn't matter at all if you do calculation in the constructor or in some method. The code doesn't get more imperative or declarative to just moving code around.

Laziness is by the way completely irrelevant for any paradigm, because you can turn any function into a lazy function if you just wrap it in another function. The whole thing where you do the calculation instead of constructor or the method is also not important because objects should be immutable. One point of immutability is that you can cache objects as long as you want, because they didn't change. And because they didn't change you also can do all calculation beforehand.

> In the first line of this snippet, we are just making an instance of an object and labeling it name. We don't want to go to the database yet and fetch the full name from there, split it into parts, and encapsulate them inside name.

You don't want *any* database action at all. Not in the constructor and not in a method. Your Storage returns you an immutable "Model" class that represents something A Book/Car/Banana/Whatever. This "Model" is immutable, completely free of any side-effects and as a result be "Persistence Ignorance".

You never want to do database action in a constructor or method. Both are bad designs.

> Such a parsing behavior would be a side effect for us and, in this case, will slow down the application.

Just a parsing is not a side-effect. The reading from the database is a side-effect. And if you now care for performance it even makes more sense to already do it in the constructor, so calculation is just done once. And the same data is not recalculated again, and again, and again on every method call.

> As you see, we may only need name.first() if something goes wrong and we need to construct an exception object.

Exception are the same level as bad as "null". And that is by the way the reason why you should do it the constructor. An Immutable object should always be valid if you have one. Either way you can create an Immutable object, or it just fails in the constructor. Throwing exceptions is in general bad, but in languages like Java or C# something should just fail at construction time or never.

> My point is that having any computations done inside a constructor is a bad practice and must be avoided because they are side effects and are not requested by the object owner.

A pure calculation is not a side-effect. And something don't get "not side-effect" by moving it from the constructor to a method. You clearly don't knew what a side-effect is. A side-effect is, when some state changes or in addition to the return result of a function/method something is changed anywhere. A database read for example is always a side-effect. And it is a side-effect in a constructor and also in a method. Side-effects will always be side-effect and they never get "not side-effects" if you move the code to some other places.

> To solve that, we create another class, a composable decorator, which will help us solve this "re-use" problem:

Or just do it in the constructor. Solving "problems" for "problems you created" is never a good idea.

• Reply • Share ›

Oleg Majewski David Raab • a month ago

> Exception are the same level as bad as "null"

WTF?

• Reply • Share ›

David Raab Oleg Majewski • a month ago

I don't knew if your problem is that you still belive that exceptions or null are a good thing. But just read:

http://www.oracle.com/technetw...

It explains why null is bad. And everything that is true for null is also true for exceptions.

You can forget to check for null -> You can forget to try/catch.
It is not obvious that a null can be returned -> It is not obvious that an exception can be thrown
It makes code more boilerplate to check for every null -> It makes code more boilerplate to check for every exceptions
....

In fact sure it is also true. Because "null" is typically used to indicate an error, and that is also the purpose of an exception. Just that exceptions don't really solve any problem. In fact what you see here as "Optional" is exactly what you do in a functional language. You have something that represent a success with a value OR an error. In Haskell it is called the Maybe Monad for example instead of Optional. Functional languages doing that like forever. But also newer some languages like Swift or Rust are doing that like forever. But also newer sane languages like Swift or Rust are doing Error handling this way.

Instead of "Exception" that is just a new word for "goto".

• Reply • Share ›

Oleg Majewski ➦ David Raab • 25 days ago

> I don't knew if your problem is that you still belive that exceptions or null are a good thing. But just read:

The WTF was about RuntimeExceptions

• Reply • Share ›

Kata Tunix ➦ David Raab • 25 days ago

The purpose of exceptions is making code clean.

/* You can forget to check for null -> You can forget to try/catch. */ You're right. But sometimes I intend to forget to try/catch because I want to try/catch is a higher level of call-stack. With null, you cannot do that.

/* It is not obvious that a null can be returned -> It is not obvious that an exception can be thrown */ Okay, they are the same. But in Java, the prototype of methods show all kind of exceptions that can be thrown. Nevertheless, it is a good practice to comment/document clearly for methods prototype.

/* It makes code more boilerplate to check for every null -> It makes code more boilerplate to check for every exceptions */ This is not correct. You don't need to check for every exception. Just check in a higher level of call-stack and all sub-exceptions will be checked too.

The only bad thing about exceptions might be performance, compared to null.

• Reply • Share ›

David Raab ➦ Kata Tunix • 25 days ago

> The purpose of exceptions is making code clean.

And they never delivered.

> But sometimes I intend to forget to try/catch because I want to try/catch is a higher level of call-stack. With null, you cannot do that.

I also never said that null would not have the same problems. I said exception are the **same level worse than null**. And by the way. Sure, you also could just return null again up to the level you want to

Sure, you also could just return null again up to the level you want to catch (I don't say it is a good idea, i'm just saying you can easily solve your problem what you described, in fact both are bad, throwing an exception or returning null). But what i propose is completely different and has nothing to do with exceptions or null. null should also eliminated like exceptions. What i propose is what a lot of functional languages (Haskell, F#, ML) or newer languages do (Rust, Swift, Go). But for this you need an "algebraic type-system" (Go has not an algebraic-type system, but error handling there is still comparable to it). And Java or C# and a lot of other typical languages like C/C++ and so on don't have one.

So to be practical in Java or C# and so on, you still use Exception, because the language doesn't provide a sane solution. But someone should reduce exceptions to the constructor. Either way you have a valid immutable object or you can't even create an immutable object. It also makes it clear to a programmer that you only have to check for exceptions at construction time. It makes it easier for a programmer, you don't need to remember which methods can throw exceptions. Only the constructor can do it. And that's it. Everything that makes programming easier is a good thing.

Creating an immutable object and then later calling a method on it that throws an exception is just mind-blowing. Because an Immutable object didn't change it will always throw an exception. So it makes no logical sense to allow an invalid immutable object to be created, and then on a later point it throws an error when you want to work with it, because it was in some illegal state.

> Okay, they are the same. But in Java, the prototype of methods show all kind of exceptions that can be thrown. Nevertheless, it is a good practice to comment/document clearly for methods prototype.

I could quote here Yegor, but i have the same opinion here like him. The best code is code that don't need documentation. The problem is always that documentation is not up-to-date and will at some point later be old. Code itself should be self-documenting. And Exceptions cannot provide that.

Did you read from Yegor the following post:
http://www.yegor256.com/2015/0...

He describes that it makes no logical sense that you have an if/else construct than in one branch can throw an error. It makes no sense, because it really is not a branch there. The same is true for exception. If you read a method it has the same problems. Even more.

```
int Foobar(int x)
int Foobar(int x)
```

1) It isn't even clear that you have a branch here, the branch is hidden from the programmer.
2) The logic is the same. For special x there exists a branch that will always throw errors. That makes no logical sense. If some "int" are illegal then you need a new type that only represents the valid things. So for example if zero is not allowed you create

```
int Foobar(NonZeroInt x)
```

The code gets more self-documented. And also no errors will be thrown calling Foobar with a NonZeroInt. But creating a NonZeroInt can thrown an error. But it makes it easy, because you knew, only constructors can throw errors. In a truly object-oriented world you by the way never use "int, string" or other primitives things. Because they are always wrong. Something can be represented as a string, but isn't a string. For example an URL can be represented as a string, but not every possible string is also an URL.

So you typically should create an URL class, that in the constructor test if the thing that you provide is a valid URL. Either way it fails in the constructor, saying it is not a valid URL, or it doesn't fail at all. If you later call a method on it and then it tells you it is not a valid URL, that is just mind-blowing. It would be the same as if you allow the code

```
int x = "foobar"
```

and it just runs fine. And it just fails at runtime when you access it, and it just says at runtime that it is not valid. If what is assigned to x is not a valid "int", then the creation of it should already fail. Not at a later point when you try to access it. What this language does is passing a value around that assumes it is a "int" but is not an "int". That is exactly the same problem that "null" have.

The problem is, that is how error handling in Java/C# works, and you can't really fix it. You need like i mentioned an algebraic type system to fix those problems, and Java don't have one. But that doesn't make exception "good". There are still worse. There are just the best-worst solution you can pick in Java. But it makes sense to try to eliminate exceptions as much as possible. And just restrict it to a constructor is a good way.

> It makes code more boilerplate to check for every exceptions */
> This is not correct. You don't need to check for every exception. Just check in a higher level of call-stack and all sub-exceptions will be checked too.

That isn't what i mean. But okay, what i wrote is also not a good description. And by the way. "higher" is one way. But what is when you want to check for an error in a deeper level?

Let's assume following pseudo-code

```
let x = SomeFunction(foo)
let y = SomeOtherFunction(x)
```

What do you need to do if you want to pass x in SomeOtherFunction. But SomeFunction() can fail. But even more. You want that when SomeFunction Fails SomeOtherFunction can probably react to it? Or just a more descriptive one.

SomeFunction would try to open a file and "x" would be a FileHandle, and SomeOtherFunction would try to load the image. But lets assume that the file provided didn't exists, or couldn't be opened, you want to provide a "Default Image".

If you want to do the error handling you have to do it directly on this level. You need to catch SomeFunction() Exceptions and react on them. And the logic of picking the default image would not be in SomeOtherFunction() it would be a level higher. Not to forget that if you forget to try/catch you get a runtime error, what is never what someone want.

With an algebraic type system your code with error-handling could look like this

```
let x = SomeFunction(foo)
let y = SomeOtherFunction(x)
```

Where is the error handling? Well x is an algebraic type that represent either a valid File OR some error. And now you pass the whole thing to SomeOtherFunction(). Now the error handling is in SomeOtherFunction() and this function can decide what to do with an error. For example if it sees that no valid file could be opened, it automatically provide a default image. So error handling is passed down the chain. And that is really what you often want. Other functions want to handle this error completely different. In a UI you for example want an error dialog that tells the user that an image could not opened, it was not an image and so on. An image control should show a default image instead.

Every object/function/method should provide its own code encapsulated what it does when an error appeared. And not some higher-level code tries to correct lowel-level components. If the Image container should show a default image, if not a valid image is provided, than this code belongs in the image container. With exceptions you have to deal for this error on a higher level and pass the Default image to the image container on a higher level. The logic the Default image to the image container on a higher level. The logic of the default image is not even in the image container. Or you open the image directly in the Image Container. But that would even be more worse, because an Image Container should not care about where the image comes from. You seriously don't want a "FileHandleImageContainer", "MemoryImageContainer" and so on.

The point is. On a higher level your code looks the same even with error handling applied. So your high-level code don't get distracted by error handling. But that is what usually happens with exceptions.

And don't miss-interpret it what you did in C. Overloading the return type to also include errors. Like something returns an int. And magically "-1" represents some error. Algebraic types is a higher type where you can describe that something can be of multiple different types, but only one specific. You can create types that are either an "int" or "error1", "error2" or "error3". But when you try to access such a type, you must check whether it is an "int", "error1", "error2" or "error3". If you don't do that, your code doesn't even compile.

Just look at the "Optional" documentation from Oracle that was introduced with Java 8 that i linked above. It is a simple rebuild of a simple algebraic-type.

> The only bad thing about exceptions might be performance, compared to null.

I don't compare it to null. Like i said, exceptions are the same level as worse than null. But Exceptions still have the following problems:

* Performance
* Not clear that something can throws exceptions
* You are not forced to handle errors
* You cannot pass errors to deeper levels
* Makes code hard to read
* You often have to handle errors at the wrong place (because you can't pass errors as values)
* It is basically just a modern "goto". You jump through your code and you end up with spaghetti-code. By the way. "Exceptions" in C are simulated with a "goto". You just do "goto ERROR1" in C to simulate that.

• Reply • Share ›

Oleg Majewski David Raab • 22 days ago

> Performance

from my point of view now days performance of calculating something is not the issue (including exceptions), the issue is writting clean and maintanable code (what does that mean?) and writing clean and maintanable code (what does that mean?), and the hardware outperforms the software by many factors.

> Not clear that something can throws exceptions
> You are not forced to handle errors

the name for this is checked exceptions, and this experiment has failed

> You cannot pass errors to deeper levels

you can, nobody restricts you to exceptions, this is only one tool, use another one, if exceptions does not do the job.
Btw "pass errors to deeper levels" using exceptions means just: don't throw it in your method, let throw an exception by a method which is called from your method.

> Makes code hard to read

example? why? If you write bad code then it's exceptions fault?

> You often have to handle errors at the wrong place (because you can't pass errors as values)

what is wrong place? Do it in the right place. You can always wrap unwanted exceptions/bad api into something wanted and good (facade pattern)

> It is basically just a modern "goto". You jump through your code and you end up with spaghetti-code. By the way. "Exceptions" in C are simulated with a "goto". You just do "goto ERROR1" in C to simulate that.

wtf? throw new Exception() how is it a goto ??? It does not tell anything about where to jump and which part of code will be executed, this is up to the code that is using a method that throws something. If you compare Exception to goto, than an Exception is rather an opposite of the goto.

• Reply • Share ›

David Raab Oleg Majewski • 21 days ago

> from my point of view now days performance of calculating something is not the issue (including exceptions), the issue is writting clean and maintanable code (what does that mean?), and the hardware outperforms the software by many factors.

Yes, i think maintanable and readability is the most important thing. But that doesn't mean that Performance is irrelevant. Would you use a Java without a JIT again? Performance is also important. The best thing would be if something is more readable, less error-prone and also better performance.

> the name for this is checked exceptions, and this experiment has failed

No it is not. Yes, you have to always check an exception. But if you had read my comment you had seen that this is not the answer. The answer is an "Algrebaic Type-System" and i even gave an example that you can pass errors to deeper levels that are not possible with checked exception. checked exception was one idea to solve the problem. And yes they failed. But that doesn't mean they are the only. Read my comment before again, i'm not talking about checked exceptions.

> you can, nobody restricts you to exceptions, this is only one tool, use another one, if exceptions does not do the job.

Yeah exactly. Use another tool instead of exceptions. Sadly most languages doesn't support an algebraic type system.

> example? why? If you write bad code then it's exceptions fault?

If you use Exceptions, then you write bad code. So yes, it is there fault.

> what is wrong place? Do it in the right place. You can always wrap unwanted exceptions/bad api into something wanted and good (facade pattern)

Then explain to me how the code above

```
let x = SomeFunction(foo)
let y = SomeOtherFunction(x)
```

work, when SomeFunction throws an exception. Without that i have to add code at that level. Sure i can try/catch the error, wrap it and so on. And i instantly have error handling at that level. And i have what? 10 lines more code to just be able to pass the error to SomeotherFunction? And i also have to create new classes for it, that all of that suff works? If you really thought about what you are saying then you would realise how bad your approach is, and how much code you have to add, in order that all of that stuff can work. Just that you can pass 1 error from 1 function to another.

> wtf? throw new Exception() how is it a goto ???

a "goto" jumps to some place. Exceptions also does that,

> It does not tell anything about where to jump and which part of code will be executed,

Yeah, that is the reason why Exceptions even are more bad than "goto". And it seems you don't really get why "goto" are bad. Exceptions jump through your code exactly like goto did. And it is not the case that exceptions are not bad because you don't define where you jump to. Goto where bad just because they "jumped".

> If you compare Exception to goto, than an Exception is rather an opposite of the goto.

An opposite to "goto" would be code that doesn't jump at all. And that is exactly what an algebraic type system gives you. Instead of jumping to some undefined code with an exception, or aborting the program. You return an union type that can contain an error.

• Reply • Share ›

Oleg Majewski David Raab • 21 days ago

> But that doesn't mean that Performance is irrelevant. Would you use a Java without a JIT again? Performance is also important.

sure performance is very import, but in the very most cases it just not the issue at all and if people say don't write this and that because of performance, this is mostly just not true. This includes don't use exceptions, they are NOT slow. What is slow, is network communication.
Exceptions is not something you should use for your normal flow execution. They are only for exceptional cases == abnormal behavior of the application.

> Yeah exactly. Use another tool instead of exceptions. Sadly most languages doesn't support an algebraic type system.

use clojure, scala and so on if you like it more...yes it supports what you want. In java you can make them by your self, it's just a class implementing an interface...

> Then explain to me how the code above
>
> let x = SomeFunction(foo)
> let y = SomeOtherFunction(x)
>
> work, when SomeFunction throws an exception.

There is a difference between Exception and a business logic error like validation. What you want sounds more like business logic error. In your use case I would not use exceptions. But if SomeFunction comes from 3rd party library and throws exceptions, then yes you have to add some code, this is in the same way valid in clojure or scala, you have to wrap it in your algebraic way first ;-)

> wtf? throw new Exception() how is it a goto ???
>
> a "goto" jumps to some place. Exceptions also does that, ...

so let assume you use a monad for error handling. Somewhere in some function you do something like:

```
result.error = badThing;
return result;
```

How do you know what effect it has?
Now somewhere (you don't know where) some other function handles it:

```
if (!result.error.isEmpty()) {
   handleError();
}
```

How is that different from throwing a Runtime Exception in the way not being a goto?
One more additional question: how do you know what caused the error? If you use exceptions right, your stacktrace becomes usable and the first entry will show you the exact place where the problem (not validation) occurred...

• Reply • Share ›

David Raab Oleg Majewski • 21 days ago

> sure performance is very import, but in the very most cases it just

> not the issue at all and if people say don't write this and that because of performance, this is mostly just not true.

Yes, i also think that way. But i don't see a problem to still list it as a disadvantage if it is still one. Even if it is not so important. And usually i would never had listed it. But i listed it here because the person i answered listed it as the first and only disadvantage.

> use clojure, scala and so on if you like it more...yes it supports what you want. In java you can make them by your self, it's just a class implementing an interface...

Not that you really get the same functionality with it compared to an algebraic type. And sure i would use Clojure or Scala. But which language i would/should use doesn't change if Exceptions are a good idea or not.

Even the "just" feels a little bit "outplaced". So you need to create a full class and interface just as a wrapper for a return value for a single method? That is really a lot to write if you do that. And the other problem is. Nothing in the language forces you really to handle the error. Well sure you can use lambdas for it. So you create an interface with a class, and you have some method like "Call", "Transition" well you don't care for the method name, and this method name expects for every state a lambda for every case.

Sure you can do that. And you end up with, huh? 1 additional interface and 1 additional class just because a single method can throw an error. Here the problem is not to "somehow solve it". The problem is that a solution like that is not practical at all.

> There is a difference between Exception and a business logic error like validation. What you want sounds more like business logic error. In your use case I would not use exceptions.

Yes sure, i talk about business logic error. Real exceptions should not be catch-able to begin with (or in some limited scope). They are a completely different kind of errors.

So, how would you solve it? Even the case i don't like Exceptions and i think they are bad. In an OO language like Java i would throw an Exception also on an Validation error, because it is the most practical one. Well, would i program in Java i would use the Optional class when it is appropriate. But not always is Optional enough.

> How is that different from throwing a Runtime Exception in the way not being a goto?

There is not a jump. The error is passed on. And every step can do its own error handling. One step can log it, another step can react different on it. Every step itself only knew how to react on an error. While with an exception you have some code above that has to knew what to do. The same as goto. You expect some place to handle the error instead of giving every step its own chance to handle the error.

But the more important thing is that languages usually forces you to think about the error case. If you do pattern matching you get a warning if you didn't handle a case. What do you get if you didn't handle a exception? Nothing, your code just compiles fine. You have to hope that your test suits is good enough to spot errors. Otherwise your program might crash sometimes at runtime (production), what is usually not what you want. But the best thing is when i don't have to write a test in the first place and the language sees errors like that.

And like i said. I don't see Checked Exception as an answer to that. Because with a checked Exception you always have to handle the error when calling a method, and that is not always wanted. And you still have that one global place instead of giving every method itself a way to react on an error.

> One more additional question: how do you know what caused the error? If you use exceptions right, your stacktrace becomes usable and the first entry will show you the exact place where the problem (not validation) occurred...

Answer to that is more complicated. At first. You usually don't care what caused the error. You only care which error is thrown. And a stack trace is only important with Exceptions. Why do you only care for that there? Because the language didn't forces you to catch errors. And because you don't get forced by it, you can forget it. And when you can forgot it, and that error happens at runtime. Well, then you are interested at the spot where it was thrown. Because you want to add code that this not happens again anymore in the future.

But with an algebraic type system that can't even happen. If you call a function you are forced somewhere to unwrap the type. And that unwrapping forces you to think about every case, not just about the "good path".

That you don't care for the actual spot becomes even more obvious if you have a try/catch added to your source. Did you ever executed some code with try {} and you later wanted to now in the "catch{}" block at which line or class the error happened because you have some logic in it and i behaves differently depending on the line where it throws? The answer would sure be "no". You only care which error was thrown. Where the error was thrown is never important.

Somewhere in the try it was thrown, you don't care at the exact spot. That is even the whole purpose of Exceptions that you don't care for the exact spot. Nobody cares for the exact method or submethod the error was thrown.

So sure, a stack trace is only important if some error happens at runtime and the error handling in your language is so bad that it can't even see at compile time that you didn't write code for the error path.

• Reply • Share ›

Oleg Majewski David Raab • 21 days ago

> So you need to create a full class and interface just as a wrapper for a return value for a single method?

yes

> That is really a lot to write if you do that.

no

> And the other problem is. Nothing in the language forces you really to handle the error

why should it? look at closure, it's syntax forces you to nothing. In my opinion the language should not restrict you too much and give enough freedom to programm what you need, not what the language or framework thinks is the best.

> So, how would you solve it

something like (it's not the full code, you might want to use more polymorphism and so on, but it gives you the idea):

```
public class ValidatedResult {

    List<validationerror> errors;

    Result result;

}

... in your function ...

if (!validatedResult.errors.isEmpty()) {

   // handle errors
}
```

This class is fully use case specific to your application and has nothing to do with the language, lambdas, exceptions, or anything else.

> There is not a jump. The error is passed on. And every step can do its own error handling. One step can log it, another step can react different on it. Every step itself only knew how to react on an error

All of this you can do with exceptions as well. You catch them, then you do log, metrics, whatever else, and then you rethrow so that next decorator can handle it.

> You expect some place to handle the error instead of giving every step its own chance to handle the error.

In worst case it is thrown to the end user, I don't expect it will be handled by my logic

> But with an algebraic type system that can't even happen. If you call a function you are forced somewhere to unwrap the type. And that unwrapping forces you to think about every case, not just about the "good path".

my simple ValidatedResult class does the same job, I don't need an extra language for it. Also this is MY class, I am not forced to do it in the way the author of language thinks it is good for me and my use case ...

> In an OO language like Java i would throw an Exception also on an Validation error, because it is the most practical one

Java is not longer a OO language, actually it never was a pure OO language. It allows you to do OO, functional or procedural, like you

like.
The problem is with that you would use exceptions for validation, and this is the problem, you should not and this is not the fail of exceptions ;)

• Reply • Share ›

David Raab Oleg Majewski • 21 days ago

> no

I think of the opposite, especially if someone also applies the rule that every interface/class should be in its own file.

> why should it?

Because your program has bugs and crashes otherwise. If you don't see that as a problem we just have different opinions on quality.

> something like (it's not the full code, you might want to use more polymorphism and so on, but it gives you the idea):

Yeah i get the idea. It is a crappy version of what an Union Types gives you. Just as an example. It makes no sense at all to use "Result" if you didn't also check "errors" before. What is the content of Results anyway when errors happened? null? And that snippet that you posted is basically the same as the Optional Type. You either have an error or a valid result. And if you don't want just Some/None what Optional already provide, well here is a simple implementation in F# of your code

```
type ValidateResult<'a> =
     | Result of 'a
     | Error of string

let stringStartsWithJ (s:string) =
    match s.StartsWith("J") with
   | true  -> Result s
   | false -> Error "Doesn't start with J"

let validate = stringStartsWithJ "foo"

match validate with
| Result x -> printfn "%s" x
| Error  x -> printfn "%s" x
```

That does everything what your code does. Is also reusable. And it forces me to write code for every case. In this case it is either a "Result" or an "Error". While your code still allow to just use Result without checking error first, and when that happen then your application does what? Crash your whole application?

And yeah, i don't need 5 files and classes to solve this problem. And i don't see a reason why the language should not force it. "Freedom!, Freedom!" is a really lame excuse. I also could say that the language should allow me to assign a string to an integer. Why? Freedom! No Restriction to my code! Buuuuh!

Yeah, why not allowing to pass 10 arguments to a 2 arguments function. Because, Freedom!

> The problem is with that you would use exceptions for validation, and this is the problem, you should not and this is not the fail of exceptions ;)

I don't see any benefits of using your approach. It is more in the other direction. An Exception at least can be seen directly and forces a programer to look into the code. While your code even allows to use "Result" even if an error happened. And i don't even knew in what state Result is when an Error appeared. In the case of an error it should not be possible to acces Result at all. Let's assume it is null, your code can even lead to more headaches, because you end up with NullReferenceException at completely different spots in your code and someone even can wonder what the hell went wrong.

So in contrast to your code i still think Exception are better. And sure, it is the fault of the language if it didn't provide a better solution.

> All of this you can do with exceptions as well. You catch them, then you do log, metrics, whatever else, and then you rethrow so that next decorator can handle it.

What you describe is not what i described. re-throwing only goes the call stack up. What i described was passing the error as a value deeper, and let every function handle it.

> In worst case it is thrown to the end user, I don't expect it will be handled by my logic

Well in what i described, there didn't exists a worst-case. The program just runs. Or you could say, the worst-case is that the compiler already sees that you didn't handle an error. So the worst-case is actually a good-case.

^ | v • Reply • Share ›

Oleg Majewski ➦ David Raab • 21 days ago

> Because your program has bugs and crashes otherwise

More from **yegor256.com**

yegor256.com recently published

How to Implement an Iterating Adapter

13 Comments Recommend

yegor256.com recently published

How Cookie-Based Authentication Works in the Takes Framework

11 Comments Recommend

yegor256.com recently published

How to Protect a Business Idea While Outsourcing

8 Comments Recommend