Community

yegor256.com

13 Comments • Created 2 months ago

Class Casting Is a Discriminating Anti-Pattern

Type casting is a very useful technique when there is no time or desire to think and design objects properly. Type casting (or class casting) helps us work with provided objects differently, based on the class they belong to or the interface t…

(yegor256.com)

13 Comments

Blue Pants • 2 months ago

Iterable does not have a size() method for a good reason, we don't know if it comes from an infinite stream (e.g., natural numbers) or from a lazy stream (e.g., very large file).

What they are doing is an optimization, and if properly documented, this is perfectly ok. The general case "always" work (if you give it an infinite stream it will never return but hey, you asked its size), and in the case Collection changes name, the most that will happen is that we have performance degradation.

What you could ask is if it makes sense to ask the size of an Iterable, but since it is in Guava, probably there are some cases where it is useful.

John Fielder • 2 months ago

Both classes look like really bad design to me. Why would you ask a foo object the size of another object? This seems like it breaks encapsulation (looks like a utility class).

2 • Reply • Share ›

Ross William Drew ➚ John Fielder • 2 months ago

True but I think it was just a bad example used to illustrate a point, I think (hope) that Yegor wasn't suggesting the demonstration class/method is a useful one.

• Reply • Share ›

Oliver Doepner • 2 months ago

I think Ross William Drew hit the nail on the head: Overloading lets your type distinction depend on compile-time types, not run-time types. But often, run-time types are the more important truth, especially if you have overloaded myMethod(A a) and myMethod(B b) where B is a subtype of A.

Java Language Specification, section 8.4.9. on "Overloading":
https://docs.oracle.com/javase...

The subtype overloading can lead very easily to the weird "anti-discrimination" effect of "treat this guy a general human being, because his birth certificate says he is human" (compile-time type), instead of a more specialized "treat him as a terrorist, because we know his DNA was found at the assassination site of our last president" (run-time type).

Besides my elaboration and illustration of the factual type distinction argument that was already made by Ross William Drew , do you see where your silly "discrimination" analogy is getting us when carried on, Yegor?

1 • Reply • Share ›

zhanggoo • 2 months ago

After preaching instanceof is "bad" or "smell", I have not 100% on this camp anymore after going through Haskell and Scala where they fully utilizes "type" pattern matching. The fact is that you do not have full ownership of _type_. Iterable and Collection are types that are defined in standard Java. It would have been better if we could have associated "size" to type "Iterable" but we cannot. So utility function (the example should have been a static function) - I do not mind using type pattern matching.

• Reply • Share ›

Ross William Drew • 2 months ago

But if you have a Collection which is being held as an Iterable and is passed to the method then you enter the Iterable method contract end up doing unecccessary counting on a Collection by treating it as Iterable, in your analogy (which I find a little ignorant) you still segregate but end up doing white person activities with black people. I would suggest that the solution here is that Iterable (or Iterator) should simply have a size method and can be asked for it's own size, rather than having to be externally counted.

• Reply • Share ›

Yegor Bugayenko author ➚ Ross William Drew • 2 months ago

You're right, but that only confirms my point. The decision of how you treat me, an object coming to your method, should be fully explained in the method signature and nowhere else. If I declare my java.util.List as java.lang.Iterable and pass it to the

method expecting **either** List or Iterable, this means that I want you to treat me as Iterable and never touch my size() method. It's my decision, not yours :)

1 • Reply • Share ›

Ross William Drew ➚ Yegor Bugayenko • 2 months ago

Maybe, but your solution is just asking for mistakes in that the user of your method/class needs to remember to upcast all the collections they pass to you and if they don't, they'll slow or break execution. If they have a collection of downcast items that all need to be sent to your class/method in turn, your solution simply moves the 'instanceof' selection tree up the chain.
My suggestion is that the ideal solution is to only accept something that is wrapped in an interface who's behaviour is consistent across objects (i.e. contains size()) then the user is forced to write good OO code and isn't tempted to throw 'instanceof' in every location your class/method is used.

Always write your code like the next person using it is a violent psychopath who knows where you live? I'd say also write your interfaces the same way.

• Reply • Share ›

Riccardo Cardin • 2 months ago

This time I totally agree with you. As Java developers, we should avoid the use of instanceof operator. As you said, it is index of bad design. For example, it could be used a Strategy pattern or a Template Method pattern in many cases, instead of using the instance of operator. Note that the sizeOf version which uses instanceof also violates the *Open/Closed Principle*. In Joshua Bloch book, "Effective Java", it is descibed only one situation in which it is allowed to use the instanceof operator , which is in overriding equals method.

3 • Reply • Share ›

Oliver Doepner • 2 months ago

And what do your overloaded methods do? They even "segregate" the parameter objects by their type, i.e. the Collection instance can only get into the special method that you defined for her. You don't even give poor Collection objects a chance to get into the method that takes Iterable parameters. Poor Collection objects - what if they also wanted to be iterated over to have their items counted one by one? Are overloaded methods special treatment, or are they like having to sit in a designated part of the bus?

What I am trying to say is that your "discrimination" theme and analogies to race and gender do not work. In typical Yegor style you tried to pour some moralistic juice into your article, since you seem to always feel a need for declaring coding practices that you don't approve of as "bad" or even "evil". Why don't you simply elaborate on the disadvantages from a Software Engineering standpoint?

In fact, you are weakening your line of argument by mixing in the far-fetched and illogical analogy of gender and racial discrimination. You might get some flame-war style comments from this, but ultimately you will just polarize the readers of your blog post and distract from the intended discussion about instanceof and type casts.

1 • Reply • Share ›

David Johnston ➚ Oliver Doepner • 2 months ago

@Oliver - I agree that the rhetoric dilutes the argument. The example is likewise flawed since there is good reason for the Iterable interface not exposing a size() property natively; and size is a very personal characteristic that you really should ask an object to provide you instead of - rudely - performing measurements on it without asking :)

Back to the point - the decision to discriminate for Collection means that you've already agreed to risk its interface changing. Whether you end up changing the if-block or the dedicated method is going to be a very minor difference. The advantage of the dedicated method is that it is self-documenting while the if-block version is not. The disadvantage is run-time decision making to figure out which method is going to be called.

So, instanceof as a means of discrimination - bad; method overloading as a means of discrimination - good. Got It!

2 ^ | v • Reply • Share ›

David Raab ➚ David Johnston • 2 months ago

> So, instanceof as a means of discrimination - bad; method overloading as a means of discrimination - good. Got It!

The point that Oliver Doepner tries to say is that it is important to provide arguments for a statement. Just saying that one thing is bad and another is good are not arguments. The typical way how it should work is showing both ways and giving a pro/contra list. It is also important to note that nothing really is just bad or just good. Everything has is Pro/Contra. But just having this one-sided black/white view isn't good at all.

And just to give one example. Typical functional languages don't provide method/function overloading at all. If you want to differentiate types then you have to do it what is compared a little bit of "instanceof" checking in Java, well just better. And they are a lot of reason why functional languages not decide to provide function overloading. The first is type-inference. When there is no function overloading the compiler always can infer the type on its own. For example lets assume you have the following F# function

```
let add (x:int) y = x + y
```

This is just an "add" function that takes two parameters x and y. x was notated with the type "int". Because it is really strong typed it is also clear that y has to be an "int". Lets now assume you now write another new function.

```
let foo x = add x 10
```

This is a function "foo" with one parameter "x". There is no type specified at all. But the compiler exactly knew here that the x in this function also have to be an "int". If you call the function foo lets say with a string, it also gives you

be an "int". If you call the function like lets say with a string, it also gives you an compile time error saying that it is not an int. How does the compiler knew that it must be an "int"? It knew all of this stuff because the foo function calls the add function and uses the x parameter as the first argument of the add. And the first argument of the add function have to be an "int". So it also knows that the x in the "foo" function also have to be an int.

All of that can only inferred because function overloading is strictly forbidden. If the language would support to write multiple "add" functions with different types, then the compiler could not infer the type automatically, that means you have to provide all the types everywhere.

That by the way doesn't mean you can't have something similar as you are used as method overloading in C#/Java. But it just differs how you archive that. In F# for example you would create a Discriminated Union. And a function would have the type of this discriminated union. If you now write a function you have to use pattern matching. That means you test which specific type was passed in and then you decide what to do based on that in the function. The difference is that the language force you to write code for all cases, because if you don't do that, you get a compile-time error. (See http://blog.ploeh.dk/2013/10/2...)

Sure in a language like Java and how "instanceof" works is a little bit different, instanceof can't be compared to pattern matching and Discriminated unions, but the more important thing was that there also exists reason to not do method overloading because method overloading destroys type-inference. Sure you can't fix this anymore in Java, or C#, C++ and some other languages, so the whole thing is in general more a matter of how to design a language.

But the most important part that i try to explain here is that a specific technique is never just good or just bad. Everything has is pro and contras. Now here instanceof vs. method overloading is not a matter of designing a language and it is already built into the language and can't be changed. But it shouldn't be too hard to describe the disadvantages of instanceof or what makes method overloading better.

I don't say that in a language like Java/C# etc. method overloading is not the way to go, but just saying the one is bad is really pretty bad. Well here in my country we have a saying of a child that touches a hot stove. It means that you can tell a child as often as you want to not touch the hot stove, it will still do it. It just means someone has to learn why something is bad, just telling a child it is bad doesn't change that he will still do it. And actually that is still true for adults. Also an adult has to learn/knew why something is bad. If you just tell someone something is bad and just don't do it, it is probably even more likely he will do it.

And this article is exactly such an article. It even says that differentiate different type is like to differentiate between gender, race or religion. That is by far one of the worst article i have read in a long time! And it even get's

more worse if his solutions still differentiates types and just the way *how* types get separated changed.

How the article is currently written needs to be completly deleted and re-written.

• Reply • Share ›

Kanstantsin Kamkou ➚ David Raab • a month ago

More or less this particular example is like the "hot stove" you mentioned ("Modern C++" c.17 explains the problem precisely for one language. Python suffers with things like this: http://aroberge.blogspot.de/20... and it makes no sense to mention php, js and other poor languages without strong type hinting and overloading).

More from yegor256.com

yegor256.com recently published

A Few Thoughts on Unit Test Scaffolding

11 Comments Recommend

yegor256.com recently published

How Cookie-Based Authentication Works in the Takes Framework

11 Comments Recommend

yegor256.com recently published

How to Implement an Iterating Adapter

13 Comments Recommend