OOP Alternative to Utility Classes

A utility class (aka helper class) is a "structure" that has only static methods and encapsulates no state. StringUtils, IOUtils, FileUtils from Apache Commons; Iterables and Iterators from Guava, and from JDK7 are perfect examples of utili…

(yegor256.com)

Alex Besogonov • 2 months ago

This is a bad design. It violates SRP by confusing algorithms and data. And it's not really that lazy either, unless lazy arguments are also used.

Calling static functions 'singletons' is misleading - they don't have ANY state, so no arguments against singletons apply. However, static functions that DO change the implicit state (like process environment or the current working directory) are evil and shouldn't even exist.

Regarding this particular example - it solves no real need and this style will mask the real problems. For example, consider an HttpNumber that throws an exception inside a crucial loop.

Brian • 4 months ago

That is a tremendous amount of work to do nothing more than change the way you call a function. Just imagine if you had to create objects like that to do simple arithmetic.

Integer x = new Add (a, b).intValue ();

Nothing would ever get done.

2 ^ | v • Reply • Share ›

Yegor Bugayenko author ➚ Brian • 4 months ago

That would be great, actually.

^ | v • Reply • Share ›

rtoal ➚ Yegor Bugayenko • 3 months ago

The idea of creating an adder with two inputs and triggering its behavior with a message _does indeed_ have merit. Provided, of course, that the whole process is syntactically suguared with a (prefix or infix) + operator, and the compiler recognizes this behavior as able to take place within an ALU of a target machine.

The idea of "nothing ever getting done" seems to be directed at using Java syntax (ugh) for the whole process and having the runtime actually construct an adder object every time it is used, which, however cheap hardware ever gets, we _do_ need to compute over ridiculously large data sets that efficiency will often matter. Slowing down a computation 10x for readability is fine, but 1000x needs justification! But this is really tangential, because we know how to implement languages where the programmer models everything as an object but compilers and interpreters do what they need to do.

By the way one need not have to sugar to an infix operator. One can write `Adder(x,y) sum` (to make the message passing explicit) or `Adder sum(x,y)` to move things around a bit. The concept is that of constructing an immutable adder object with x and y, then invoking the sum only when needed. One can also write `Maxer(3,5,2) max` or even `Max of [3 5 2]`. It's just syntax.

Anyway, Yegor's ideas are completely in the spirit of OO as envisioned by Alan Kay and the other creators of Smalltalk, and telling a Smalltalker they don't get anything done would not go over well. In Smalltalk, however, adders and maxers are not the objects; numbers are. Kay's contribution was showing how to recursively decompose the computer into little computers, NOT into data structures and algorithms! That is the beauty of objects. In one paper (I forget which) they talked about finally figuring out how to make the number 3 a computer! (In Smalltalk we send the message +5 to the object 3.) But it's implemented fast. Kay wrote "It would be terrible if a + b incurred any overhead if a and b were bound, say, to “3″ and “4″ in a form that could be handled by the ALU. The operations should occur full speed using look-aside logic (in the simplest scheme a single and gate) to trap if the operands aren’t compatible with the ALU. Now all elementary operations that have to happen fast have been wrapped without slowing down the machine." Let the compiler make it fast. That's not a difficult problem.

I'm not sure why people love utility classes so much. Or their singletons.

Maybe they feel a comfort level with procedural programming? Spend some time thinking about making objects that know how to do the things you want to do, or even (yes) adding those behaviors to existing objects. You will probably find a way and will be happy you avoided utilities and singletons.

Perhaps there might be an aversion to adding so much "behavior" to a number class? Might there be name clashes if every numeric algorithm became a method on the number class? Should we use mixins? Maybe mixins will be treated in a future article.

1 • Reply • Share ›

Radek Postolowicz • 4 months ago

I'd rather write:

```
int max = Max.of(10, 5);
```

instead of:

```
int max = new Max(10, 5).intValue();
```

It's more readable and it's more desired than pure OO design.

2 • Reply • Share ›

Brian Stempin ➦ Radek Postolowicz • a month ago

Also, it avoids the creation of a throw-away object.

• Reply • Share ›

Oliver Doepner • 5 months ago

I have to say, your code examples are terrible.

A Max object that "implements Number"? Are you talking about java.lang.Number? That is not even an interface.

Anyway, why create a Max throw-away object each time you want to compare two numbers? If you mainly don't like that Math.max(..) is static and does not implement an interface, then use a MyMath class that implements an interface Math with a max(..) method. And why only for 2 numbers? How about a much more flexible varargs method max(Number... numbers) and/or max(Iterable<number> numbers)? In general, a stateless long-lived service object is much preferable to a throw-away object that can only used for a single purpose.

The 2nd example is also terrible. When I saw that you suggest implementing Collection(!) for iterating over the lines of a text file, I stopped reading. All the methods of that interface, just to iterate over lines of a file? Have you actually looked at http://docs.oracle.com/javase/... before you came up with that idea? Just as one simple example: The Collection class has a remove(Object o) method. Your FileLines class would have to search through the file and remove all occurrences of o (assuming o is a String). Also, Collection has an iterator() method, so you would have to create stateful iterators,

each requiring their own stream behind the scenes. All that complexity for the simple purpose of iterating over the lines of a file? You must be kidding. Not to speak of the leakiness of your abstraction when a Collection suddenly has to throw IOException (probably you would have to wrap them in some form).

All that without even mentioning the standard Java IO way using a reader based on a file input stream.

Overall your code examples look like you quickly created them for your blog post, not like something from a mature code base that you and your team have actually used anywhere. I mean, "implements Number" would not even compile.

5 ^ | v • Reply • Share ›

gbr • 6 months ago

I'm glad I saw this, I've been experimenting with some similar ideas in a recent project. Not in java, but I feel like you have said a lot of the things I've been feeling as I've tried to see if I can sort out a really well designed immutable OO model. The nature of the problem means that readability will be about one billion times more important than performance, and I've been loving this approach.

2 ^ | v • Reply • Share ›

Oliver Weiler • 6 months ago

Horrible advice. The 'Everything is an object' is one of Java many design failures. There is a reason that nearly every JVM language which came after Java allowed plain functions.

7 ^ | v • Reply • Share ›

Carlos Alexandro Becker ↗ Oliver Weiler • 3 months ago

As far as I know, not everything is an object in Java. Primitives and methods, for example, are not objects...

1 ^ | v • Reply • Share ›

aRRRRrrr • 6 months ago

I have a hard time understanding the statement about the complexity. both implementation would have similar complexity as the algorithm don't differ greatly and there is no way to transform n line of a file with a constant complexity.

^ | v • Reply • Share ›

Bruno Martins • 6 months ago

Yegor, your articles are a great read and i feel your passion about OOP.

Regarding to this topic; i get what you are saying, but, having a OOP puritanism view of things might cloud your attention to other important aspects. There are reasons for why this problem and others you wrote about, ("why NULL is bad"; "Objects should be immutable") exist and aren´t exactly solvable.

Developers tend to pay a lot of focus on code readability & patterns, and have ever so more, a gap of understanding of human readability -> code -> machine generated instructions.

OOP purity will make you believe that creating objects to try to solve every single problem is actually a good thing. This is understandable, because its easier for us humans read it. Not realising that this has deep implications (like Memory & Performance).

This may also lead to problems when you have a developer that all of a sudden, has to interact with the native part of things, like JNI (Java Native Interface) or even other platforms.

I believe that this is something every architect should be aware. Having a pretty, clean design might look cool, but ultimately, we all want efficient systems for our users. And this requires way more deep considerations then having the best patterns and the most readable code.

7 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ Bruno Martins • 6 months ago

Thanks for reading :) I understand your point and I disagree. I actually have an article about it: http://www.yegor256.com/2014/1... I believe that memory & performance concerns are far less important in modern software development than readability and maintainability. Why? Because computers are cheaper than programmers. One hour of my time in front of computer trying to understand a 2000-lines class will cost most than a new RAM stick into the server. That's why, I think, we should start thinking about performance only when the code is pure and clean. See my logic?

3 ▲ | ▼ • Reply • Share ›

Bruno Martins ➚ Yegor Bugayenko • 6 months ago

Right, that´s a very common mindset of developers. But it´s not like the current standards are impossible to read or maintain. In fact, any average java developer, understands null checks, utility classes and mutable objects for example (since you wrote about those). They have their flaws, but it´s all about the context you´re dealing with. The fact that they exist is due to the way the OOP languages were built to provide the best flexibility possible.

Additionally, relying on hardware to just perform better with less-performing software is in some way is contemptible (I´m speaking as an engineer here, not a manager). You also see everyday poor-performance, poor memory management in modern applications.

And Yes, Hardware is cheaper and better, sure, but feature demand of software is also higher. And what about in systems & platforms where resources are extremely important, like peripherals, wearables, games, expert-systems or even mobile devices. If your only concern is simple, clean applications, with abundant resources i can get that. But for really performing & relying solutions, you need to understand what you write in code has deep consequences in what the compiler will generate and instruct the machine to do. It´s also important to teach people to design &

implement software taking all this in considerations regardless if they are "allowed" to be cheap, write the most human-readable code & perfect design. Which is not the most important.

I admire your passion for software design & architecture; and there is a lot of your articles i do agree with it. But in some cases, i think you missed focus on the engineering (in applicable&practical terms) and not just human-readable, nice, pretty patterns (or anti).

8 ▲ | ▼ • Reply • Share ›

Dreaming Vertebrate • 7 months ago

Cool!

1 ▲ | ▼ • Reply • Share ›

Dave Millar-Haskell • 7 months ago

I like what you say in most of your articles but this one not so much. You seem to be obsessed with OOP purity. When I have a procedural block to implement (do this, do that) that is not coupled to exisiting classes then I write procedural code. No matter to me that static methods did the job as long as it's transparent and understandable. The way you shoehorned

int max = NumberUtils.max(10, 5);

into

int max = new Max(10, 5).intValue();

does not do anything good for my client code. You just made me instantiate an object because ...er because ,,, everything in OOP MUST be an object!? I am concerned for you...OOP is not a religion. That way lies madness.

6 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➚ Dave Millar-Haskell • 7 months ago

This will look less crazy for you if you imagine the following situation. I have two sources of text lines. One is a huge XML file (100Gb), another one is a big SQL table (100m lines). I want to compare the number of XML elements with the amount of SQL records that match some regular expression, then get the maximum. Obviously, this calculation will take time (a few minutes). I want to do it in a lazy way, meaning that if the maximum is not used by anyone, I want to avoid the calculation. What is your solution? Here is mine:

```
Number max = new Max(
  new XMLElements(), // implements Number
  new SQLRecords() // implements Number
);
// later, when I need (IF I NEED IT!) the actual number
System.out.println(max.intValue());
```

Compare it with your code and tell me which one looks crazier :)

• Reply • Share ›

Alex Besogonov ➚ Yegor Bugayenko • 2 months ago

So why do you even create a "Number max" if you don't need it? To stress-test the garbage collector?

• Reply • Share ›

as_cii ➚ Yegor Bugayenko • 6 months ago

> What is your solution?

I cannot speak for Dave Millar-Haskell, but I would avoid to separate instantiation from execution, thus avoiding the object altogether and simply using some Math library.

Your version:

```
class MyUI

{

  private final Number max;

  public MyUI(XMLElements xmlElements, SQLRecords sqlRecords)

  {

     this.max = new Max(xmlElements, sqlRecords);

  }

  public void onButtonClick()

  {

     this.text.updateText(this.max.intValue());

  }
```

My version:

```
class MyUI

{
  private final XMLElements xmlElements;
  private final SQLRecords sqlRecords;
```

```
public MyUI(XMLElements xmlElements, SQLRecords sqlRecords)
{
   this.xmlElements = xmlElements;
   this.sqlRecords = sqlRecords;
}

public void onButtonClick()
{
   Number max = Math.max(this.xmlElements, this.sqlRecords);
   this.text.updateText(max.intValue());
}
```

Still lazy, but no unnecessary object involved.

1 ▲ | ▼ • Reply • Share ›

Sihle • 7 months ago

Yegor, I like what you say about OO in your posts, thanks a ton. But what is your view on the subject of AI, especially Machine Learning, where the assumption is that the system learns from its data? I find it (ML) to be an intrigueing topic, but it seems to go against the grain of "there is no data, there are only objects and their behaviors". How do we successfully address ML needs in OO, if its possible at all? A post with an example would be nice!

▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➦ Sihle • 7 months ago

I have very small experience in machine learning, but I can't see anything against OOP in it. I'll give you another example. Currently we're working with a data-encoding project, where bytes and bits are transformed on-fly. And we're doing it in a pure OO way, without any static methods, getters or mutable classes. I'm planning to make a few more posts soon, with practical examples. Stay in touch :)

▲ | ▼ • Reply • Share ›

Patrick Sagan • 7 months ago

Good article. Thanks.

1 ▲ | ▼ • Reply • Share ›

Nathan Green • 8 months ago

Your `Max` class stores two values and uses only one: that's a waste of memory. How about this?

```
public final class Max extends Number {
    private final int max;

    public Max(int a, int b) {
        max = a > b ? a : b;
    }
```

```
    public static Max of(int a, int b) {
        return new Max(a, b);
    }

    @Override public int intValue() {
        return max;
    }
}
```

There's no need to clutter up methods with conditionals (which often need to be repeated across other, similar methods) when you can just move them into the constructor :) I also added a static method so your example could change to `int max = Max.of(10, 5).intValue();` But saving a single keystroke is hardly notable: I just point it out because I dislike the `new` keyword.

Edit: I forgot to mention: it's a travesty that `Number` is an abstract class and not an interface. Too many core Java libraries are classes rather than interfaces.

• Reply • Share ›

Yegor Bugayenko author ➚ Nathan Green • 8 months ago

Imagine this situation (I'm using your class):

```
Number max = new Max(new HttpNumber(), 5);
if (/* something goes wrong */) {
  throw new Exception("no need to continue");
}
System.out.println("max is: " + max);
```

Your class will make a call to `HttpNumber` and will perform the calculations I may not need.

The static method you added... I'm against any static methods in general. I don't see why people like these "factories" while we have constructors.

• Reply • Share ›

Alex Besogonov ➚ Yegor Bugayenko • 2 months ago

Don't do this. This is a HORRIBLE design.

HttpNumber is _fundamentally_ different beast compared to a regular number. It's about 9 orders of magnitude slower! Doing a typical HTTP request can take 10-1000 milliseconds versus a couple of cycles for a local variable. And then there's a question of failure handling.

• Reply • Share ›

Nathan Green ➚ Yegor Bugayenko • 8 months ago

The nice thing about being an architect is you get to dictate whether these static methods are allowed at all. (If you want to do so, anyway.)

• Reply • Share ›

Yegor Bugayenko author ➚ Nathan Green • 8 months ago

Yes, being an architect has same benefits and drawbacks as being a dictator. If you succeed they worship you, if you fail they eat you. See http://www.yegor256.com/2014/1...

1 ^ | v • Reply • Share ›

Nathan Green ➚ Yegor Bugayenko • 8 months ago

My class just stores a primitive, as does your example. If you're thinking lazy evaluation would be a good idea (it very well might), then switching the inputs to Number and storing them would work. There were no requirements, so I was just working with your original code.

I understand the dislike of static methods, I just also dislike using `new` because it forces you to use a particular implementation and you lose the ability to swap the implementation at runtime. Not that there are any perfect solutions, but using interfaces and acquiring object instances at runtime (via some configuration class, DI container, whatever) avoids coupling code to a particular implementation unnecessarily.

If you have an eagerly evaluating class, how do you switch to a lazily evaluating class without recompiling your code? Assuming both classes implement the same interface, in the ideal case your code only depends on the interface, not the implementation.

1 ^ | v • Reply • Share ›

Yegor Bugayenko author ➚ Nathan Green • 8 months ago

I agree with everything you just said. Besides, have to say that "primitives" like `int`, `double`, `boolean` are counter-OOP. That's where Ruby beats Java.

^ | v • Reply • Share ›

Kanstantsin Kamkou • a year ago

Only one addition/question here. It seems like you are talking only about languages which are support multiple constructors (like java, c#, c++, etc.). Or factory-constructor-like languages like go language. Because I can't find a good approach for languages like python, php or ruby.

The easiest example I can see is the [Utils]Password class. Lets assume that we do have two methods: getSalt and encode. The first one doesn't require this.password variable, but it is important for the second function.

And an example I want to show is:
string aNewSalt = new Password().getSalt().
string encodedPassword = new Password(originalPassword).encode();

In this case we can change the signature of the encode function, and accept string, but I'm loosing a point of OOP style here. Could you clarify it?

• Reply • Share ›

Yegor Bugayenko author ➚ Kanstantsin Kamkou • a year ago

I think I would hide this salting functionality inside `Password` class. Getting something from a class in order to use it later, is a bad practice on its own. We should not use a class as a storage of data. We ask our password to print itself to a string. How the password will be salted, encrypted, and encoded is not our business. Password can take care of itself. In PHP:

```
$password = new Password("secret");
echo $password->__toString();
```

However, I agree that method overloading (you called it "multiple constructors") is a missed feature in many OO languages. For example, we want to make it our password smart enough to use an internal salt or a provided one, in Ruby:

```
first = Password.new("secret")
second = Password.new("secret", "my-salt")
```

This SO discussion explains how this can be done in Ruby. There are similar workarounds in other languages.

• Reply • Share ›

Kanstantsin Kamkou ➚ Yegor Bugayenko • a year ago

I have prepared a piece of PHP code, from a "File Manager" class. Could you spend some time re-factoring it into the OOP-Utils style:

```
$itDirectory = new RecursiveDirectoryIterator(__DIR__);
$itRecursive = new RecursiveIteratorIterator($itDirectory);
$itRecursive->setMaxDepth(2);
$itRegexp = new RegexIterator($itRecursive, '~\.txt$~', RecursiveRegexIt

foreach ($itRegexp as $fileinfo) {
    if (!$fileinfo->isDir()) {
        var_dump($fileinfo->getFileName());
    }
}
```

The current signature looks like this:

```
public static function list($path, $mask = '~^\.txt$~', $depth = 0){}
$list = \Utils\Folder::list(__DIR__);
```

Is it all about "a room for subjectivity", way of improvement, or just a way to overload things in place where we can keep it simple?

• Reply • Share ›

Yegor Bugayenko author ➚ Kanstantsin Kamkou • a year ago

This is how your `Folder` class looks now:

This is how your `Folder` class looks now:

```
class Folder {
  public static function list($path) {
    $i = // construct an iterator to find all necessary files
    $array = [];
    foreach ($i as $f) {
      $array[] = $f;
    }
    return $array;
  }
}
```

This is not OOP. It is a procedural programming. You are using `Folder` class as a namespace for method `list()`. Instead, you should design your `Folder` class so that it exposes a behavior of a collection of files. Again, `Folder` itself should be a collection of files. It should not return a collection, but **be** a collection and **behave** as a collection:

```
class Folder implements Iterator {
  public function __construct($path) {
  }
  // you figure out yourself how to design its methods
}
```

Got the idea?

1 ▲ | ▼ • Reply • Share ›

Guest ➦ Yegor Bugayenko • a year ago

Yes, I got it. Thank you for the explanation!

▲ | ▼ • Reply • Share ›

Elvis • a year ago

Very curios about this purist OOP approach.
I don't think it's a really good idea to do something like
int max = new Max(10, 5).intValue();
especially when you are in a loop. You would be creating objects for no reason.
Of course, you can have setters on your Max object considering you would instantiate it right above the loop, but still when you call the intValue method you should verify that both numbers are set.
Which seems kinda overengineering to me, but then again there is room for subjectivity especially in the OOP context.

2 ▲ | ▼ • Reply • Share ›

Yegor Bugayenko author ➦ Elvis • a year ago

I disagree about "room for subjectivity" :) I think that OOP can be and should be very strict and objective. Instead, procedural programming is very subjective (take Perl, for example)

Perl, for example).

What's wrong about creating objects in a loop? Object creation should be a very cheap operation. And garbage collection should destroy them after each loop cycle.

Besides that, if you need to create multiple `Max` objects in a loop - something is wrong with the algorithm and it needs refactoring.

• Reply • Share ›

rtoal ➚ Yegor Bugayenko • 3 months ago

Any decent compiler can see that `int max = new Max(10, 5).intValue();` is an expression that creates an anonymous object and pulls data from it immediately and maintains no references to it. Its maxing operation can be completely inlined so there are no performance concerns at all. The "purist OOP approach" is a beautiful way to model things because it decomposes objects into objects into objects into objects and so on. Nothing about OOP as a modeling approach has anything that requires heap allocation with malloc and garbage collection or performance penalties. Never underestimate the skills of good compiler writers.

1 • Reply • Share ›

Marcos Douglas Santos ➚ Yegor Bugayenko • 6 months ago

If we can not use static methods and classes should be immutable as you wrote in another article, ie, not use setters. How can I use Max class inside a loop without create it multiple times?

• Reply • Share ›

Yegor Bugayenko author ➚ Marcos Douglas Santos • 6 months ago

There is no way. You will have to make a new instance of `Max` class in each loop cycle. I don't think it's very harmful, to be honest. This could be a potential performance bottleneck, but in very rare cases. As usual, I'm in favor of code readability and clean design, versus performance, see http://www.yegor256.com/2014/1...

• Reply • Share ›

Paul Polishchuk ➚ Yegor Bugayenko • a year ago

to "What's wrong about creating objects in a loop?"
PMD would shout: "Avoid instantiating new objects inside loops" :)

5 • Reply • Share ›

Lukas Eder • a year ago

You're probably serious about this, right?

How about using that nice `java.math.Math::max` method reference as a `IntBinaryOperator` and pass it to the new `IntStream.reduce()` method? Doesn't look *too* evil to me. But then again, I'm not very dogmatic...

3 • Reply • Share ›

Yegor Bugayenko author ↗ Lukas Eder • a year ago

Well, passing a method reference to another method is a pure functional approach, which became possible in Java8. It's not evil, but it's not OOP: