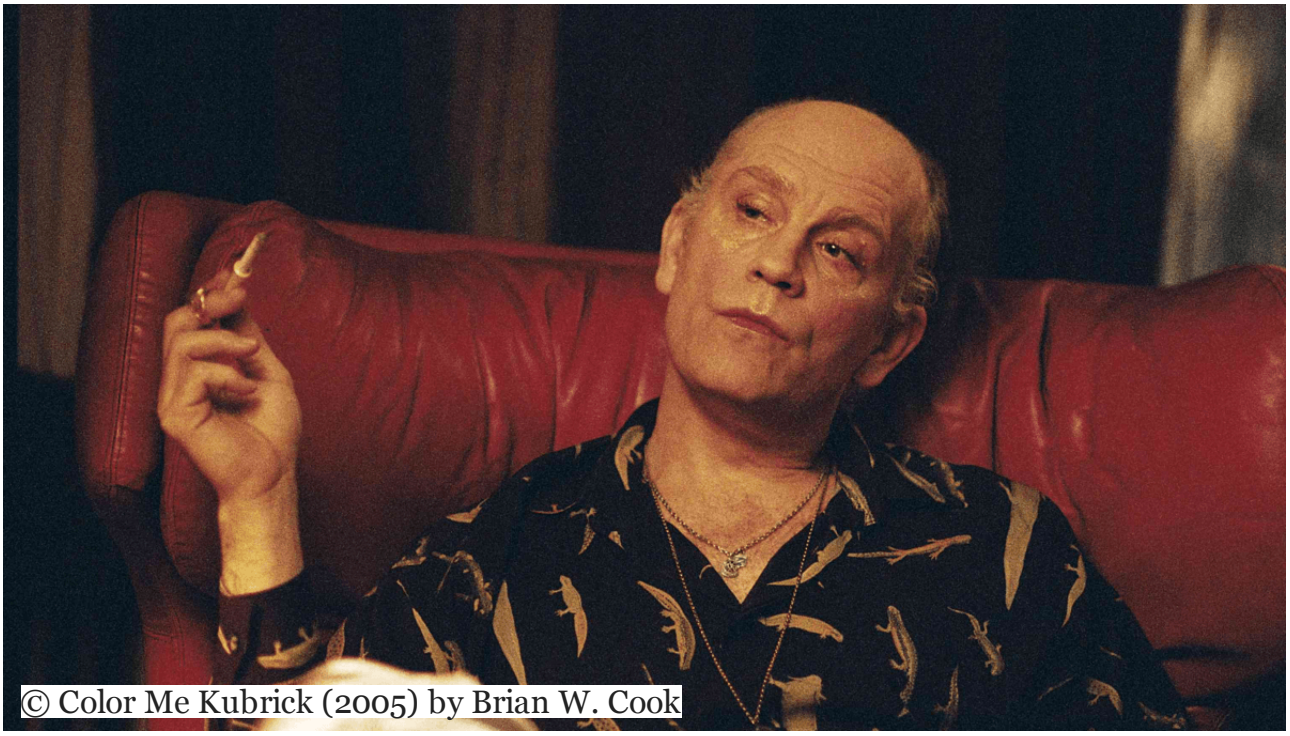
http://www.yegor256.com/2015/02/20/utility-classes-vs-functional-programming.html

Utility Classes Have Nothing to Do With Functional Programming

20 February 2015 modified on 3 March 2015 Yegor Bugayenko

I was recently accused of being against functional programming because I call utility classes an anti-pattern. That's absolutely wrong! Well, I do consider them a terrible anti-pattern, but they have nothing to do with functional programming. I believe there are two basic reasons why. First, functional programming is declarative, while utility class methods are imperative. Second, functional programming is based on lambda calculus, where a function can be assigned to a variable. Utility class methods are not functions in this sense. I'll decode these statements in a minute.

In Java, there are basically two valid alternatives to these ugly utility classes aggressively promoted by Guava, Apache Commons, and others. The first one is the use of traditional classes, and the second one is Java 8 lambda. Now let's see why utility classes are not even close to functional programming and where this misconception is coming from.


© Color Me Kubrick (2005) by Brian W. Cook

Here is a typical example of a utility class `Math` from Java 1.0:

```
public class Math {
  public static double abs(double a);
  // a few dozens of other methods of the same style
}
```

Here is how you would use it when you want to calculate an absolute value of a floating point number:

```
double x = Math.abs(3.1415926d);
```

What's wrong with it? We need a function, and we get it from class `Math`. The class has many useful functions inside it that can be used for many typical mathematical operations, like calculating maximum, minimum, sine, cosine, etc. It is a very popular concept; just look at any commercial or open source product. These utility classes are used everywhere since Java was invented (this `Math` class was introduced in Java's first version). Well, technically there is nothing wrong. The code will work. But it is not object-oriented programming. Instead, it is imperative and procedural. Do we

care? Well, it's up to you to decide. Let's see what the difference is.

There are basically two different approaches: declarative and imperative.

Imperative programming is focused on describing **how** a program operates in terms of statements that change a program state. We just saw an example of imperative programming above. Here is another (this is pure imperative/procedural programming that has nothing to do with OOP):

```
public class MyMath {
  public double f(double a, double b) {
    double max = Math.max(a, b);
    double x = Math.abs(max);
    return x;
  }
}
```

Declarative programming focuses on **what** the program should accomplish without prescribing how to do it in terms of sequences of actions to be taken. This is how the same code would look in Lisp, a functional programming language:

```
(defun f (a b) (abs (max a b)))
```

What's the catch? Just a difference in syntax? Not really.

There are many definitions of the difference between imperative and declarative styles, but I will try to give my own. There are basically three roles interacting in the scenario with this `f` function/method: a *buyer*, a *packager* of the result, and a *consumer* of the result. Let's say I call this function like this:

```
public void foo() {
  double x = this.calc(5, -7);
  System.out.println("max+abs equals to " + x);
```

```
}
private double calc(double a, double b) {
  double x = Math.f(a, b);
  return x;
}
```

Here, method `calc()` is a buyer, method `Math.f()` is a packager of the result, and method `foo()` is a consumer. No matter which programming style is used, there are always these three guys participating in the process: the buyer, the packager, and the consumer.

Imagine you're a buyer and want to purchase a gift for your (girl|boy)friend. The first option is to visit a shop, pay $50, let them package that perfume for you, and then deliver it to the friend (and get a kiss in return). This is an **imperative** style.

The second option is to visit a shop, pay $50, and get a gift card. You then present this card to the friend (and get a kiss in return). When he or she decides to convert it to perfume, he or she will visit the shop and get it. This is a **declarative** style.

See the difference?

In the first case, which is imperative, you force the packager (a beauty shop) to find that perfume in stock, package it, and present it to you as a ready-to-be-used product. In the second scenario, which is declarative, you're just getting a promise from the shop that eventually, when it's necessary, the staff will find the perfume in stock, package it, and provide it to those who need it. If your friend never visits the shop with that gift card, the perfume will remain in stock.

Moreover, your friend can use that gift card as a product itself, never visiting the shop. He or she may instead present it to somebody else as a gift or just exchange it for another card or product. The gift card itself becomes a product!

So the difference is what the consumer is getting — either a product ready to be used (imperative) or a voucher for the product, which can later be converted into a real product (declarative).

Utility classes, like `Math` from JDK or `StringUtils` from Apache Commons, return products ready to be used immediately, while functions in Lisp and other functional languages return "vouchers". For example, if you call the `max` function in Lisp, the actual maximum between two numbers will only be calculated when you actually start using it:

```
(let (x (max 1 5))
  (print "X equals to " x))
```

Until this `print` actually starts to output characters to the screen, the function `max` won't be called. This `x` is a "voucher" returned to you when you attempted to "buy" a maximum between `1` and `5`.

Note, however, that nesting Java static functions one into another doesn't make them declarative. The code is still imperative, because its execution delivers the result here and now:

```
public class MyMath {
  public double f(double a, double b) {
    return Math.abs(Math.max(a, b));
  }
}
```

"Okay," you may say, "I got it, but why is declarative style better than imperative? What's the big deal?" I'm getting to it. Let me first show the difference between functions in functional programming and static methods in OOP. As mentioned above, this is the second big difference between utility classes and functional programming.

In any functional programming language, you can do this:

```
(defun foo (x) (x 5))
```

Then, later, you can call that `x`:

```
(defun bar (x) (+ x 1)) // defining function bar
(print (foo bar)) // passing bar as an argument to foo
```

Static methods in Java are not *functions* in terms of functional programming. You can't do anything like this with a static method. You can't pass a static method as an argument to another method. Basically, static methods are procedures or, simply put, Java statements grouped under a unique name. The only way to access them is to call a procedure and pass all necessary arguments to it. The procedure will calculate something and return a result that is immediately ready for usage.

And now we're getting to the final question I can hear you asking: "Okay, utility classes are not functional programming, but they look like functional programming, they work very fast, and they are very easy to use. Why not use them? Why aim for perfection when 20 years of Java history proves that utility classes are the main instrument of each Java developer?"

Besides OOP fundamentalism, which I'm very often accused of, there are a few very practical reasons (BTW, I am an OOP fundamentalist):

Testability. Calls to static methods in utility classes are hard-coded dependencies that can never be broken for testing purposes. If your class is calling `FileUtils.readFile()`, I will never be able to test it without using a real file on disk.

Efficiency. Utility classes, due to their imperative nature, are much less efficient than their declarative alternatives. They simply do all calculations right here and now, taking processor resources even when it's not yet necessary. Instead of returning a promise to break down a string into chunks, `StringUtils.split()` breaks it down right now. And it breaks it

down into all possible chunks, even if only the first one is required by the "buyer".

Readability. Utility classes tend to be huge (try to read the source code of `StringUtils` or `FileUtils` from Apache Commons). The entire idea of separation of concerns, which makes OOP so beautiful, is absent in utility classes. They just put all possible procedures into one huge `.java` file, which becomes absolutely unmaintainable when it surpasses a dozen static methods.

To conclude, let me reiterate: Utility classes have nothing to do with functional programming. They are simply bags of static methods, which are imperative procedures. Try to stay as far as possible away from them and use solid, cohesive objects no matter how many of them you have to declare and how small they are.