http://www.yegor256.com/2014/05/05/oop-alternative-to-utility-classes.html

OOP Alternative to Utility Classes

5 May 2014 modified on 24 October 2014 Yegor Bugayenko

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 `Files` from JDK7 are perfect examples of utility classes.

This design idea is very popular in the Java world (as well as C#, Ruby, etc.) because utility classes provide common functionality used everywhere.

Here, we want to follow the DRY principle and avoid duplication. Therefore, we place common code blocks into utility classes and reuse them when necessary:

```
// This is a terrible design, don't reuse
public class NumberUtils {
  public static int max(int a, int b) {
    return a > b ? a : b;
  }
}
```

Indeed, this a very convenient technique!?

Utility Classes Are Evil

However, in an object-oriented world, utility classes are considered a very bad (some even may say "terrible") practice.

There have been many discussions of this subject; to name a few: Are Helper Classes Evil? by Nick Malik, Why helper, singletons and utility classes are mostly bad by Simon Hart, Avoiding Utility Classes by Marshal Ward, Kill That Util Class! by Dhaval Dalal, Helper Classes Are A Code Smell by Rob Bagby.

Additionally, there are a few questions on StackExchange about utility classes: If a "Utilities" class is evil, where do I put my generic code?, Utility Classes are Evil.

A dry summary of all their arguments is that utility classes are not proper objects; therefore, they don't fit into object-oriented world. They were inherited from procedural programming, mostly because most were used to a functional decomposition paradigm back then.

Assuming you agree with the arguments and want to stop using utility classes, I'll show by example how these creatures can be replaced with proper objects.

Procedural Example

Say, for instance, you want to read a text file, split it into lines, trim every line and then save the results in another file. This is can be done with `FileUtils` from Apache Commons:

```
void transform(File in, File out) {
  Collection<String> src = FileUtils.readLines(in, "UTF-8");
  Collection<String> dest = new ArrayList<>(src.size());
  for (String line : src) {
    dest.add(line.trim());
  }
  FileUtils.writeLines(out, dest, "UTF-8");
}
```

The above code may look clean; however, this is procedural programming,

not object-oriented. We are manipulating data (bytes and bits) and explicitly instructing the computer from where to retrieve them and then where to put them on every single line of code. We're defining *a procedure of execution*.

Object-Oriented Alternative

In an object-oriented paradigm, we should instantiate and compose objects, thus letting them manage data when and how *they* desire. Instead of calling supplementary static functions, we should create objects that are capable of exposing the behaviour we are seeking:

```
public class Max implements Number {
  private final int a;
  private final int b;
  public Max(int x, int y) {
    this.a = x;
    this.b = y;
  }
  @Override
  public int intValue() {
    return this.a > this.b ? this.a : this.b;
  }
}
```

This procedural call:

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

Will become object-oriented:

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

Potato, potato? Not really; just read on...

Objects Instead of Data Structures

This is how I would design the same file-transforming functionality as above but in an object-oriented manner:

```
void transform(File in, File out) {
  Collection<String> src = new Trimmed(
    new FileLines(new UnicodeFile(in))
  );
  Collection<String> dest = new FileLines(
    new UnicodeFile(out)
  );
  dest.addAll(src);
}
```

`FileLines` implements `Collection<String>` and encapsulates all file reading and writing operations. An instance of `FileLines` behaves exactly as a collection of strings and hides all I/O operations. When we iterate it — a file is being read. When we `addAll()` to it — a file is being written.

`Trimmed` also implements `Collection<String>` and encapsulates a collection of strings (Decorator pattern). Every time the next line is retrieved, it gets trimmed.

All classes taking participation in the snippet are rather small: `Trimmed`, `FileLines`, and `UnicodeFile`. Each of them is responsible for its own single feature, thus following perfectly the single responsibility principle.

On our side, as users of the library, this may be not so important, but for their developers it is an imperative. It is much easier to develop, maintain and unit-test class `FileLines` rather than using a `readLines()` method in a 80+ methods and 3000 lines utility class `FileUtils`. Seriously, look at its source code.

An object-oriented approach enables lazy execution. The `in` file is not read until its data is required. If we fail to open `out` due to some I/O error, the

first file won't even be touched. The whole show starts only after we call `addAll()`.

All lines in the second snippet, except the last one, instantiate and compose smaller objects into bigger ones. This object composition is rather cheap for the CPU since it doesn't cause any data transformations.

Besides that, it is obvious that the second script runs in O(1) space, while the first one executes in O(n). This is the consequence of our procedural approach to data in the first script.

In an object-oriented world, there is no data; there are only objects and their behavior!