# Ugly Code #1

**Let us talk about ugly and beautiful string concatenation, and unnamed constants.**

```
//Ugly example 1:
Runtime rt = Runtime.getRuntime();

Process pr = rt.exec("C:\\Windows\\Microsoft.NET\\Framework64\\
v4.0.30319\\csc.exe /R:System.Diagnostics.Contracts.dll /nologo /
target:library /out:" + f + codeCompExt + " " + f + codeExt);

System.out.print("Code compilation: ");
```

```
//Ugly example 2:
for (String methodName : testMethodNames) {
    classStr.append("\t\ttry {\n");

    classStr.append("\t\t\t" + methodName + "();\n");

    classStr.append("\t\t} catch (Exception e) {\n");
    classStr.append("\t\t\t//throw new RuntimeException(e);\n");
    classStr.append("\t\t\tSystem.out.println(\"
            + methodName
            + ": \" + e.getClass().getName() + \": \" +
e.getMessage());\n");
    classStr.append("\t\t}\n");
}
```

### The Ugly

Why do I consider this code ugly? Firstly, it is hard to parse with just a quick glance; secondly, I have to read the strings to know what they represent; and thirdly, applying changes is harder than it should be. Let us take a closer look at these three problems.

Quite often we have to navigate through several source code files to find the place we have to work on. If we do not know where exactly this location is, we scroll through the code and glance at every source code line until something catches our attention. Ideally, this will only be the line we are searching for. However, in the case of both examples, the sheer length of the lines will catch our attention for no reason. This slows us down and distracts us. The source code highlighting of our editor ➔

is of course of no help, as in most cases it highlights strings with highly contrasted colours. I understand if you think that this point is not as important as I think that it is. That's fine, as the following remarks are more important.

The second problem is that I have to parse the strings to understand exactly what the lines do. This includes realizing that in Java, the string "\\" results in a string containing one backslash, that "\t" is replaced by a tabulator, "\n" by a new line and so on. Surely this is not a huge problem as we get used to parsing such very widely used place holders. However, something like "\t\ttry {\n" is already not an easy situation to realize that the result is going to be a string with two tabs followed by a "try {". If you still think this is not a problem, just take a look at the second example and try to explain to yourself what the output is going to be without needing a lot of time to just parse the string constants. This can be hugely improved.

**I am not going to be a happy guy if I have to spend time finding all those locations.**

What if you have to change the path in the first example? What if you want to add some code to the output in the second?

In the first example, you could change the one string, but not quite that easily, as the source file contains three more locations with the same path. Therefore, you have to check the whole source file for possible changes. Very impractical. You do not think such changes happen often enough to care? Well, I could not run the underlying application of the first example as it expects the .NET compiler to be at the default file path. I have multiple systems where this is not true and I would have to change it every-

```
//Beauty example 1:
private static final String DOT_NET_COMPILER_FILEPATH = "C:\\Windows\\Mi
        crosoft.NET\\Framework64\\v4.0.30319\\csc.exe";
private static final String DOT_NET_COMPILER_FLAGS = "/R:System.Diag
        nostics.Contracts.dll /nologo /target:library";
private static final String DOT_NET_COMPILER_EXEC_FORMATSTR = "%s %s /
        out:%1$s.dll %1$s.cs";
}

Runtime rt = Runtime.getRuntime();
Process pr = rt.exec(String.format(DOT_NET_COMPILER_EXEC_FORMATSTR, DOT_
        NET_COMPILER_FILEPATH, DOT_NET_COMPILER_FLAGS, f));
System.out.print("Code compilation: ");
```

```
//Beauty example 2:
private static final String METHOD_CALL_FORMATSTR = "\t\ttry {\n"
                + "\t\t\t%1$s();\n"
                + "\t\t} catch (Exception e) {\n"
                + "\t\t\t//throw new RuntimeException(e);\n"
                + "\t\t\tSystem.out.println(\"%1$s: \" + e.getClass().
                    getName() + \": \" + e.getMessage());\n"
                + "\t\t}\n";

for(String methodName : testMethodNames) {
        classStr.append(String.format(METHOD_CALL_FORMATSTR, methodNa
                me));
}
```

where. If I need to spend time trying to find all those locations, I will not be happy.

To add new code to the output in the second example you will always spend some time trying to find out which line has to be modified, and how to structure the code from scratch to make it look less cluttered. All those changes are cumbersome and can be made easier to do.

### The beauty

Just look at how nice the for-each loop now looks in the improved second example. Let us first talk about the introduced improvements, followed by some downsides.

First of all, the most important improvement is the readability of the source code. We have removed the unnamed constant strings and put them at the top of our class. Now, if we are reading the method bodies, we can think about what they do and what their business logic is, instead of what the exact calls/strings that are produced look like. If we want to change the strings, we know where to find them. There is one single convenient and well visible place for all of them. A change to the strings now applies to all the locations in which they are used for calls.

Giving those strings meaningful names improves our understanding of what each line does. Just look at the variable "f" in the first example: do you know right away what it stands for? Of course not, although you may already have the intuition that it is a file name. However, if you look at our names for the string constants, you have immediate knowledge of what they stand for. This is in many situations very handy and much better than parsing those long strings at every line.

➔

Of course, there are some downsides to this approach. Firstly, formatting strings in Java, as in many other languages, uses a bit of an obscure place holder syntax. For example, "%1$s" is a place holder for a string and uses the second parameter passed to `String.format`. However, we also did get used to "\t", "\n", "\\", and so on. Why should we not also get used to this notation? There is also a notion of order in which we pass the values `toString.format`: changing it will generate wrong results. This can be improved by using libraries that allow named place holders — `"{FILENAME}.cs"` — and taking some kind of key-value pair structures to replace the place holders with the expected value. For example, Python provides such functionality out of the box. Never underestimate the power of meaningful names!

```
DOT_NET_COMPILER_FILEPATH = r'C:\Windows\Microsoft.NET\Framework64\
        v4.0.30319\csc.exe'
DOT_NET_COMPILER_FLAGS = '/R:System.Diagnostics.Contracts.dll /nologo /
        target:library'
DOT_NET_COMPILER_EXEC_FORMATSTR = '%(COMPILER_FILEPATH)s %(COMPILER_
        FLAGS)s /out:%(FILENAME)s.dll %(FILENAME)s.cs'

print(DOT_NET_COMPILER_EXEC_FORMATSTR % ({
    'COMPILER_FILEPATH': DOT_NET_COMPILER_FILEPATH,
    'COMPILER_FLAGS': DOT_NET_COMPILER_FLAGS,
    'FILENAME': 'someFile'
}))
```