http://www.yegor256.com/2014/04/27/typical-mistakes-in-java-code.html

Typical Mistakes in Java Code

27 April 2014 modified on 1 April 2015 Yegor Bugayenko

This page contains most typical mistakes I see in the Java code of people working with me. Static analysis (we're using qulice) can't catch all of the mistakes for obvious reasons, and that's why I decided to list them all here.

Let me know if you want to see something else added here, and I'll be happy to oblige.

All of the listed mistakes are related to object-oriented programming in general and to Java in particular.

Class Names

Your class should be an abstraction of a real life entity with no "validators", "controllers", "managers", etc. If your class name ends with an "-er" — it's a bad design. BTW, here are my seven virtues of a good object. Also, this post explains this idea in more details: Don't Create Objects That End With -ER.

And, of course, utility classes are anti-patterns, like `StringUtils`, `FileUtils`, and `IOUtils` from Apache. The above are perfect examples of terrible designs. Read this follow up post: OOP Alternative to Utility Classes

Of course, never add suffixes or prefixes to distinguish between interfaces and classes. For example, all of these names are terribly wrong: `IRecord`,

`IfaceEmployee`, or `RecordInterface`. Usually, interface name is the name of a real-life entity, while class name should explain its implementation details. If there is nothing specific to say about an implementation, name it `Default`, `Simple`, or something similar. For example:

```
class SimpleUser implements User {};
class DefaultRecord implements Record {};
class Suffixed implements Name {};
class Validated implements Content {};
```

Method Names

Methods can either return something or return `void`. If a method returns something, then its name should explain *what it returns*, for example (don't use the `get` prefix ever):

```
boolean isValid(String name);
String content();
int ageOf(File file);
```

If it returns `void`, then its name should explain *what it does*. For example:

```
void save(File file);
void process(Work work);
void append(File file, String line);
```

There is only one exception to the rule just mentioned — test methods for JUnit. They are explained below.

Test Method Names

Method names in JUnit tests should be created as English sentences without spaces. It's easier to explain by example:

```
/**
 * HttpRequest can return its content in Unicode.
 * @throws Exception If test fails
 */
public void returnsItsContentInUnicode() throws Exception {
}
```

It's important to start the first sentence of your JavaDoc with the name of the class you're testing followed by `can`. So, your first sentence should always be similar to "somebody *can* do something".

The method name will state exactly the same, but without the subject. If I add a subject at the beginning of the method name, I should get a complete English sentence, as in above example: "HttpRequest returns its content in unicode".

Pay attention that the test method doesn't start with `can`.Only JavaDoc comments start with 'can.' Additionally, method names shouldn't start with a verb.

It's a good practice to always declare test methods as throwing `Exception`.

Variable Names

Avoid composite names of variables, like `timeOfDay`, `firstItem`, or `httpRequest`. I mean with both — class variables and in-method ones. A variable name should be long enough to avoid ambiguity in its scope of visibility, but not too long if possible. A name should be a noun in singular or plural form, or an appropriate abbreviation. More about it in this post: A Compound Name Is a Code Smell. For example:

```
List<String> names;
void sendThroughProxy(File file, Protocol proto);
private File content;
public HttpRequest request;
```

Sometimes, you may have collisions between constructor parameters and in-class properties if the constructor saves incoming data in an instantiated object. In this case, I recommend to create abbreviations by removing vowels (see how USPS abbreviates street names).

Another example:

```
public class Message {
  private String recipient;
  public Message(String rcpt) {
    this.recipient = rcpt;
  }
}
```

In many cases, the best hint for a name of a variable can ascertained by reading its class name. Just write it with a small letter, and you should be good:

```
File file;
User user;
Branch branch;
```

However, *never* do the same for primitive types, like `Integer number` or `String string`.

You can also use an adjective, when there are multiple variables with different characteristics. For instance:

```
String contact(String left, String right);
```

Constructors

Without exceptions, there should be only *one* constructor that stores data in object variables. All other constructors should call this one with different arguments. For example:

```
public class Server {
  private String address;
  public Server(String uri) {
    this.address = uri;
  }
  public Server(URI uri) {
    this(uri.toString());
  }
}
```

One-time Variables

Avoid one-time variables at all costs. By "one-time" I mean variables that are used only once. Like in this example:

```
String name = "data.txt";
return new File(name);
```

This above variable is used only once and the code should be refactored to:

```
return new File("data.txt");
```

Sometimes, in very rare cases — mostly because of better formatting — one-time variables may be used. Nevertheless, try to avoid such situations at all costs.

Exceptions

Needless to say, you should **never** swallow exceptions, but rather let them bubble up as high as possible. Private methods should always let checked exceptions go out.

Never use exceptions for flow control. For example this code is wrong:

```
int size;
try {
  size = this.fileSize();
} catch (IOException ex) {
  size = 0;
}
```

Seriously, what if that `IOException` says "disk is full"? Will you still assume that the size of the file is zero and move on?

Indentation

For indentation, the main rule is that a bracket should either end a line or be closed on the same line (reverse rule applies to a closing bracket). For example, the following is not correct because the first bracket is not closed on the same line and there are symbols after it. The second bracket is also in trouble because there are symbols in front of it and it is not opened on the same line:

```
final File file = new File(directory,
  "file.txt");
```

Correct indentation should look like:

```
StringUtils.join(
  Arrays.asList(
    "first line",
    "second line",
    StringUtils.join(
```

```
      Arrays.asList("a", "b")
    )
  ),
  "separator"
);
```

The second important rule of indentation says that you should put as much as possible on one line - within the limit of 80 characters. The example above is not valid since it can be compacted:

```
StringUtils.join(
  Arrays.asList(
    "first line", "second line",
    StringUtils.join(Arrays.asList("a", "b"))
  ),
  "separator"
);
```

Redundant Constants

Class constants should be used when you want to share information between class methods, and this information is a characteristic (!) of your class. Don't use constants as a replacement of string or numeric literals — very bad practice that leads to code pollution. Constants (as with any object in OOP) should have a meaning in a real world. What meaning do these constants have in the real world:

```
class Document {
  private static final String D_LETTER = "D"; // bad practice
  private static final String EXTENSION = ".doc"; // good practice
}
```

Another typical mistake is to use constants in unit tests to avoid duplicate string/numeric literals in test methods. Don't do this! Every test method should work with its own set of input values.

Use new texts and numbers in every new test method. They are independent. So, why do they have to share the same input constants?

Test Data Coupling

This is an example of data coupling in a test method:

```
User user = new User("Jeff");
// maybe some other code here
MatcherAssert.assertThat(user.name(), Matchers.equalTo("Jeff"));
```

On the last line, we couple `"Jeff"` with the same string literal from the first line. If, a few months later, someone wants to change the value on the third line, he/she has to spend extra time finding where else `"Jeff"` is used in the same method.

To avoid this data coupling, you should introduce a variable.