

**CCS Digital Services Directorate**

**Enterprise Level Test Automation**

**The Coding Standards**

**Version**: 0.3

**Status: Draft**

**Date issued**: 10/11/2020

**Version Control**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Amendment** | **Changed by** |

# 

**Distributions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Title** | **Date** | **Version** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# 

# 

**INTRODUCTION**

This document serves as the **complete** definition of CCS Enterprise level Test Automation coding standards for source code in the Java Programming Language. A Java source file is described as being in CCS Test Automation style if and only if it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the **hard-and-fast rules** that we follow universally, and avoids giving advice that isn't clearly enforceable (whether by human or tool).

Code conventions are important for a number of reasons:

* 80% of the lifetime cost of a piece of software goes to maintenance.
* Hardly any software is maintained for its whole life by the original author.
* Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
* If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

In this document, unless otherwise clarified:

1. The term *class* is used inclusively to mean an "ordinary" class, enum class, interface or annotation type (@interface).
2. The term *member* (of a class) is used inclusively to mean a nested class, field, method, *or constructor*; that is, all top-level contents of a class except initializers and comments.
3. The term *comment* always refers to *implementation* comments. We do not use the phrase "documentation comments", instead using the common term "Javadoc."

**Basic Rules for Source Files**

* **File name**: The source file name consists of the case-sensitive name of the top-level class it contains plus the .java extension.
* **File Encoding**: Source files are encoded in **UTF-8.**
* **Whitespace characters**: Aside from the line terminator sequence, the ASCII horizontal space character (0x20) is the only whitespace character that appears anywhere in a source file. This implies that:
* All other whitespace characters in string and character literals are escaped.
* Tab characters are **not** used for indentation.
* **Special escape sequences**: For any character that has a [special escape sequence](http://docs.oracle.com/javase/tutorial/java/data/characters.html) (\b, \t, \n, \f, \r, \", \' and \\), that sequence is used rather than the corresponding octal (e.g. \012) or Unicode (e.g. \u000a) escape.

**Structure of Source Files**

A source file consists of, **in order**:

* License or copyright information, if present
* Package statement
* Import statements
* Exactly one top-level class

**Exactly one blank line** separates each section that is present.

**Wildcard imports**, static or otherwise, **are not used**.

Imports are ordered as follows:

1. All static imports in a single block.
2. All non-static imports in a single block.

If there are both static and non-static imports, a single blank line separates the two blocks. There are no other blank lines between import statements.

Within each block the imported names appear in ASCII sort order. (**Note:** this is not the same as the import *statements* being in ASCII sort order, since '.' sorts before ';'.). Static import is not used for static nested classes. They are imported with normal imports.

#### Declaration of Classes

* **Exactly one top-level class declaration:** Each top-level class resides in a source file of its own.
* **Ordering of class contents:** The order you choose for the members and initializers of your class can have a great effect on learnability. However, there's no single correct recipe for how to do it; different classes may order their contents in different ways. What is important is that each class uses some**logical order**, which its maintainer could explain if asked. For example, new methods are not just habitually added to the end of the class, as that would yield "chronological by date added" ordering, which is not a logical ordering.
* **Overloads**: When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no other code in between (not even private members).

**Formatting Rules**

* **Braces**: Braces are used with if, else, for, do and while statements, even when the body is empty or contains only a single statement. Braces follow the Kernighan and Ritchie style ("[Egyptian brackets](http://www.codinghorror.com/blog/2012/07/new-programming-jargon.html)") for *nonempty* blocks and block-like constructs:
* No line break before the opening brace.
* Line break after the opening brace.
* Line break before the closing brace.
* Line break after the closing brace, *only if* that brace terminates a statement or terminates the body of a method, constructor, or *named* class. For example, there is *no* line break after the brace if it is followed by else or a comma.
* **Block indentation**: Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block.
* **One statement per line**: Each statement is followed by a line break.
* **Line wrapping:** When an expression will not fit on a single line, break it according to these general principles:
* Break after a comma.
* Break before an operator.
* Prefer higher-level breaks to lower-level breaks.
* Align the new line with the beginning of the expression at the same level on the previous line.
* If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

**Naming Conventions**

* Rules common to all identifiers

Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores. Thus each valid identifier name is matched by the regular expression \w+ .

In Google Style, special prefixes or suffixes are **not** used. For example, these names are not Google Style: name\_, mName, s\_name and kName.

* Rules by identifier type
* Package names

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, com.example.deepspace, not com.example.deepSpace or com.example.deep\_space.

* Class names

Class names are written in [UpperCamelCase](https://google.github.io/styleguide/javaguide.html" \l "s5.3-camel-case).

Class names are typically nouns or noun phrases. For example, Character or ImmutableList. Interface names may also be nouns or noun phrases (for example, List), but may sometimes be adjectives or adjective phrases instead (for example, Readable).

There are no specific rules or even well-established conventions for naming annotation types.

*Test* classes are named starting with the name of the class they are testing, and ending with Test. For example, HashTest or HashIntegrationTest.

* Method names

Method names are written in [lowerCamelCase](https://google.github.io/styleguide/javaguide.html" \l "s5.3-camel-case).

Method names are typically verbs or verb phrases. For example, sendMessage or stop.

Underscores may appear in JUnit *test* method names to separate logical components of the name, with *each* component written in [lowerCamelCase](https://google.github.io/styleguide/javaguide.html" \l "s5.3-camel-case). One typical pattern is *<methodUnderTest>*\_*<state>*, for example pop\_emptyStack. There is no One Correct Way to name test methods.

* Constant names

Constant names use CONSTANT\_CASE: all uppercase letters, with each word separated from the next by a single underscore. But what *is* a constant, exactly?

Constants are static final fields whose contents are deeply immutable and whose methods have no detectable side effects. This includes primitives, Strings, immutable types, and immutable collections of immutable types. If any of the instance's observable state can change, it is not a constant. Merely *intending* to never mutate the object is not enough. Examples:

These names are typically nouns or noun phrases.

* non-constant field names

Non-constant field names (static or otherwise) are written in [lowerCamelCase](https://google.github.io/styleguide/javaguide.html" \l "s5.3-camel-case).

These names are typically nouns or noun phrases. For example, computedValues or index.

* Parameter names

Parameter names are written in [lowerCamelCase](https://google.github.io/styleguide/javaguide.html" \l "s5.3-camel-case).

One-character parameter names in public methods should be avoided.

* Local variable names

Local variable names are written in [lowerCamelCase](https://google.github.io/styleguide/javaguide.html" \l "s5.3-camel-case).

Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

* Type variable names

Each type variable is named in one of two styles:

* A single capital letter, optionally followed by a single numeral (such as E, T, X, T2)
* A name in the form used for classes followed by the capital letter T (examples: RequestT, FooBarT).
* Camel case: defined

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve predictability, Google Style specifies the following (nearly) deterministic scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
   * *Recommended:* if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel case *per se*; it defies *any* convention, so this recommendation does not apply.
3. Now lowercase *everything* (including acronyms), then uppercase only the first character of:
   * ... each word, to yield *upper camel case*, or
   * ... each word except the first, to yield *lower camel case*
4. Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded.



