General-Purpose Style Guide

Ethan Ruffing

December 8, 2014

Contents

Chapter 1

Introduction

In my personal experience of writing programs and code, it has come to my attention that it is beneficial to have predetermined conventions of coding style. Furthermore, I have learned that these conventions must necessarily vary from one programming language to the next due to the widely varying uses and though processes behind each programming environment.

Therefore, I have decided to write this general-purpose style guide as a reference on which conventions are most suitable under which circumstances.

This guide is organized in a hierarchical structure, first by programming paradigm, then by programming language, and, finally, by feature. Any section of this guide is intended to inherit the instructions of those sections above it, and all are intended to inherit those instructions in the "Everywhere" section of this of this document.

Chapter 2

Everywhere

File Organization

Source code should be written so that no line exceeds eighty characters in length.

A header block should be placed at the top of each file giving the author, the date of creation, and, if applicable, the version number. This header block should also include a copyright notice and license header if applicable.

File names should be clear and concise so that someone can tell the general purpose of the file without having to open and read it or its header block.

Commenting

Comments should be included where necessary to give a better understanding of code. When writing longer comments, follow Strunk & White's *The Elements of Style* for English grammar guidelines.

When writing dates and times, use the ISO 8601 standard. That is, for dates, YYYY-MM-DD; for times, hh:mm:ss; and, for date-times, YYYY-MM-DDThh:mm:ss.

Indentation

There seems to be an ongoing war regarding indentation in programming. For the purposes of this style guide, we will require the use of hard tabs (an actual `\t` character). IDE's and text editors should be set to use a tab width of four, and alignment of things such as comments should be based on this assumption.

Exceptions

There are two exceptions to this rule.

First, when formatting block comments, all text within the block should be aligned using spaces in order to guarantee readable documentation everywhere. (NOTE, however, that the comment block itself should be indented using tabs.)

Second, in languages that are whitespace sensitive, such as Python, use spaces for all indentation.

In both of these exceptions, when indenting using spaces, indent each level at four spaces past the last.

Chapter 3

Object-Oriented Programming

All languages

Commenting

Comments should be included that give detailed descriptions of all functions, classes, and instance variables.

Documentation

When possible, DocBlock-style comments should be provided for *every* class and method (public or private). DocBlock comments should be formatted using the `/** ... */` system, as illustrated in the example further down.

Note that the available tags vary between interpreters for these comments. Here are some of the interpreters:

- JavaDoc comes with the Java Development Kit, and is used almost universally for Java documentation.
- Doxgen supports many languages and is the most commonly used tool for C and C++ documentation.
- ApiGen (based on phpDocumentor) works well for PHP documentation and has built-in support in NetBeans. This blog post at sitepoint.com can act as a good reference for the tags that ApiGen supports.
- JSDoc is a very full-featured system for documenting JavaScript source code.

Author and date (since) should *always* be given for a class DocBlock. If using version numbering for the project, the version tag should also *always* appear in the class DocBlock.

At a minimum, functions/methods should *always* have the date (since) tag. If multiple authors have contributed to the same class, they should also include the author tag. (If not given, the author is to be assumed to be the same as the class author.)

Functions/methods should also have `@param` and `@return` tags where applicable. Any method other than a simple getter/setter (one which does not operate on the variable, but merely sets or returns its value) must have a description of its functionality. The description portion of these tags should be located on the next line (after the tag and variable name) and indented one level past the tag itself.

Class and instance variables and fields should have a DocBlock comment to explain their purpose.

Description text should be written in sentences and be punctuated with a period. Variable and `@param`/`@return` value descriptions, however, should be kept brief and not written as full sentences, unless absolutely necessary. (The first letter of these descriptions, though, should still be capitalized.)

A blank line should be placed after each paragraph of explanation; between the explanation and author, date, and version tags; and between those tags and the param and return tags.

Example

```
/**
 * This is an explanation of a function.
 *
 * @author Ethan Ruffing <ruffinge@gmail.com>
 * @since 2014-08-06
 *
 * @param varName A brief description of the parameter varName
 * @return A brief description of the function's return value
 */
```

Naming

In general, names should be concise, but long enough to understand immediately.

Classes

Classes and interfaces should be titled in CamelCase, with a name that reflects the real-world object which the class is intended to represent.

Variables

All variables should be named in headlessCamelCase.

Constants

Constants (including enum entries) should be named in UPPER_CASE.

Whitespace

Whitespace helps tremendously to improve readability. The following are basic guidelines, but can be expanded on to further improve readability.

Blank Lines

One blank line should always be used in the following locations:

- Between methods
- Between a local variable in a method and its first statement
- Before a block or single-line comment
- Between logically separate sections of code within a method

Blank Spaces

Blank spaces should be used in the following locations:

- Between a keyword and a parenthesis. Note that a blanks space should *not* be used between a method and its opening parenthesis. Also, there should not be any blank space following an opening parenthesis or preceding a closing parenthesis. Example:

```
while (true) {
System.out.println(testVar);
...
}
```

- Before an opening brace (see above example)

- Between binary and ternary operators and their operands
- *Not* between unary operators and their operands
- After each semicolon in a `for` loop (for example, `for (int i = 0; i < n; i++) {}`)

Braces

All loops, conditionals, and other such structures where braces can be "optional" *must* use braces, even if their contents is a single line.

Error Handling

There are multiple existing systems of error handling. For example, in traditional C, errors are passed via return values as integers. In some places, errors are passed as simple booleans (i.e., `false` means an error occurred).

Nearly all object-oriented programming languages, however, support the much more sophisticated system of throwing exceptions. This is the method that should be used whenever possible for error handling. It is a system that fits much more naturally into the paradigm of object-oriented programming and allows for much more detailed reports and, therefore, easier debugging.

If a library is using some other form of error handling, it would be prudent to incorporate a wrapper of some form that will allow your program to throw an the error as an exception.

Exceptions, when appropriate, should be thrown as high as possible. They should not be caught and ignored in a private function. Rather, they should, ideally, make it as high as a static method (for example, in Java, `main()`), or the primary operating method of your program.

Java

Unless otherwise specified here, follow the guidelines of Oracle's Java Coding Style.

Braces

- Opening braces should always be on the same line as their parent.
- Close braces should always be on their one line, aligned horizontally with their open statement.

File Contents

Each file should contain only one top-level class, interface, enum, etc. The file should be named identically to that top-level item, with the file extension `*.java`. For example, if a file contains `class HelloWorld`, it should be named `HelloWorld.java`.

PHP

Unless otherwise specified here, follow the guidelines of the PHP Pear Coding Standards.

Braces

- Opening braces for classes and functions should always be on their own line, aligned on the column of the item they are for.
- Opening braces for loops and conditionals should always be on the same line as their parent.
- Close braces should always be on their one line, aligned horizontally with their open statement.

JavaScript

Unless otherwise specified here, follow the guidelines of the Google JavaScript Style Guide.

Braces

- Opening braces should always be on the same line as their parent.
- Close braces should always be on their one line, aligned horizontally with their open statement.

C++

Unless otherwise specified here, follow the guidelines of the Google C++ Style Guide.

Braces

- Opening braces should always be on their own line, aligned on the column of the item they are for.
- Close braces should always be on their one line, aligned horizontally with their open statement.

Chapter 4

Functional Programming

All Languages

Commenting

Comments should be included that give detailed descriptions of all functions, classes, and instance variables.

Documentation

When possible, DocBlock-style comments should be provided for *every* class and method (public or private). DocBlock comments should be formatted using the `/** ... */` system, as illustrated in the example further down.

The most commonly used DocBlock interpreter for C documentation is Doxgen. It supports export into a multitude of formats and supports a large number of tags. For a reference of what tags are supported, see the Special Commands section of the Doxygen Manual.

Author and date (since) should *always* be given for a file DocBlock. If using version numbering for the project, the version tag should also *always* appear in the class DocBlock.

At a minimum, functions should *always* have the date (since) tag. If multiple authors have contributed to the same class, they should also include the author tag. (If not given, the author is to be assumed to be the same as the class author.)

Functions should also have `@param` and `@return` tags where applicable. Any method other than a simple getter/setter (one which does not operate on the variable, but merely sets or returns its value) must have a description of its

functionality. When applicable to the language, parameter tags should also specify the parameter direction (`[in]` or `[out]`). The description portion of these tags should be located on the next line (after the tag and variable name) and indented one level past the tag itself.

Global variables should have a DocBlock comment to explain their purpose, and macros should be described in a block somewhere near the beginning of the file.

Description text should be written in sentences and be punctuated with a period. Variable and `@param`/`@return` value descriptions, however, should be kept brief and not written as full sentences, unless absolutely necessary. (The first letter of these descriptions, though, should still be capitalized.)

A blank line should be placed after each paragraph of explanation; between the explanation and author, date, and version tags; and between those tags and the param and return tags.

Example

```
/**
 * This is an explanation of a function.
 *
 * @author Ethan Ruffing <ruffinge@gmail.com>
 * @since 2014-08-06
 *
 * @param[in] varName
 *     A brief description of the parameter varName
 * @param[out] outName
 *     A brief description of the output outName
 * @return
 *     A brief description of the function's return value
 */
```

C

Unless otherwise specified here, follow the guidelines of the Google C++ Style Guide, paying attention only to those portions applicable to plain C.

Braces

- Opening braces should always be on their own line, aligned on the column of the item they are for.
- Close braces should always be on their one line, aligned horizontally with their open statement.

Chapter 5

Markup Languages

All Languages

Markup languages, by their nature, are largely self-documenting. Therefore, when writing in a markup language, it is permissible to be more lax when commenting than in other languages.

HTML

Unless otherwise specified here, follow the guidelines of the W3C HTML5 Syntax Recommendations.

Paragraphing

In writing HTML, there is occasionally a tendency to write multiple tags on one line. This should be avoided, as it leads to difficulty in reading the source. Instead, place each open and close tag on its own line, with the contents of those tags indented between them.

The one exception to this rule is for trivial tag contents, such as the text in an `href`, in which case it is permissible (even advisable) to place the contents directly between the tags on the same line.

CSS

Braces

When programming in CSS, follow the "end-of-line" opening brace style.

- Opening braces should always be on the same line as their parent.
- Closing braces should alaways be on their own line, aligned horizontally with their open statement

Spaces

A space should be placed after each colon, but *not* before the colon.

Documentation

Document all styles using KSS.