Auto-Garçon Style Guide

Version 2.0 / May 21th, 2020

This document serves as a reference for the coding standards on the Auto-Garçon project. The purpose of this guide is to ensure that all members of the project can collaborate effectively by maintaining a consistent styling standard.

This guide is not comprehensive, so if you have a question about style, ask a member of the style guide team. For everything else, please use your best judgement. You may also want to look through Google's official style guide at **https://google.github.io/styleguide/**. Keep in mind that the style guide for this project will not be consistent with all of Google's styling rules.

1. Contents

2. Naming Conventions

Variables

All variable names are required to be in `camelCase`, regardless of language. No variable name should be greater than 20 characters. The name should clearly describe what it is used for. If you cannot describe its function within the character limit, a description should precede the variable definition as a comment.

Classes / Files

All class names are required to be in `TitleCase`. This extends to the file that contains the class, and any other files that use the same name as the class. For any other type of file, `snake_case` must be used.

(Note: If a file is generated for your project as a part of an IDE, you do not need to change it's name.)

3. Comments

Method Descriptions

Comments will be required at the beginning of each function that is greater than one line in length. They should include a high-level explanation of the function and its output. **The name of the writer of each function should be commented with this explanation as well** in case of questions in the future regarding the contents.

For more dense functions that have loops with multiple operations within them, include a brief explanation of the contents of the loop before the loop starts. Similarly, if there are any particularly dense operations, include a comment regarding the output/goal of the operation prior to its occurrence.

Delimiters

The type of comment delimiter is only dependent on the length on the comment; one-line comments may use one-line comment delimiters. You may use multiple single line comment delimiters for a multi-line comment. **Always use a space between the comment delimiter and the start of the comment for readability.**

```
// myFunction: this function returns the value of n after a
// set of operations
// Author: Author Name
function myFunction(int n) {
    ...
}
```

✓ ***Acceptable***

4. Readme

Each working group oversees their own working repository, which includes the Readme document that should be filled out as needed. The Readme should contain the contributors to the repository, who are the members of the specific group. Along with this, the Readme needs to include a brief description of the most significant directories and subdirectories, as well as their use and running instructions.

There should be a concise description of the significant folders within the Readme. These descriptions should give the folder name and a brief explanation of the files within it and their purpose.

5. Script Organization & White Space

Organization

Each script should be organized with the "starter" functions at the top of the script. These functions include any needed constructors and one-line get and set functions. **These get and set methods should be grouped together**. Beneath the starter functions, use your best judgment to organize functions top-down; functions used more often than others should be generally be listed towards the top.

White Space

Avoid unnecessary white space within and between each function; include one new line of white space between functions for ease of reading.

6. Indentation & Scope

Indentation

Since every code editor has its own conventions for displaying tabs, we will want to expand all tabs into spaces**.** You can enable this in most code editors by enabling "**soft tabs**". We also will be using a tab length of 4 as our standard.

Note: For large configuration files such as XML or JSON, or HTML documents that require a lot of nesting, you may set the tab length to 2.

```
def fibonnaci(i):
    if (I <= 1):
        return 1
    else:
        return fibonnaci(i-1) + fibonnaci(i-2)
```

✓ ***Acceptable***

Braces

Braces `{}` must be used for all loops and conditional statements in any language that supports them. This includes, but is not limited to **C/C++**, **Java**, and **JavaScript**. The opening brace should be on the same line as the statement that it is a part of. **There must be a space between the bracket and the statement**. The closing brace should be on its own line, unless it is followed by `else`, `catch`, or a similar statement.

```
if (i==1) {
    return;
}
```

✓ ***Acceptable***

```
if(i==2){
    return;
}
```

× ***Not enough spacing between braces and code.***

```
if (i==3)
{
    return;
}
```

× ***Bracket is not on the same line as the if statement.***

```
if (i==4) return;
```

× ***Missing brackets.***

Line Wrapping

Most code editors show a preferred line length of 80 characters. For this project, we can set our preferred width **up to 120 characters**. If you have a statement that cannot fit within this line width, you may need to wrap the statement across multiple lines. If this is the case, you should have all following lines set to an indentation of 8 spaces, or two soft tabs.

```
restaurant.getMenu().getItem("Hamburger").addToReciept(
        thisCustomer.getReciept());
```

✓ ***Acceptable***

7. Commit Messages

Commit messages must include a subject and a body. The subject should be related to what type of commit you are making (ex: add feature, fix bug, style, refactor, test, documentation, etc.). Write subjects in the imperative: "fix bug" and not "fixed bug". Wrap subject to 50 characters.

The body is separate from the subject with a blank line. The body should explain the changes you made, where and why you made them. Write to ensure that anyone can understand why the change was made (your code is not self-explanatory). Bullet points are fine.

```
Feature: Add save menu option

Users were unable to save drafts or final copies of
menus
- Added saveMenu() method to Menu class
```

✓ ***Acceptable***

8. JavaScript

Function Declaration

Functions should only be defined in one of two ways for this project. For all functions that are defined as a top-level member of a script, they must be defined with the `function` keyword. If the function needs to be exported, it should be added into the `exports` object after it has been defined. If the function is a part of a class, omit the function keyword but keep the same syntax.

```
function someFunction(parameter) {
    return parameter + 1;
}

exports = {someFunction};
```

✓ ***Acceptable***

```
class MenuItem {
    ...
    getCalories() {
        return this.calories;
    }
}
```

✓ ***Acceptable***

```
exports.someFunction = (parameter) => {
    return parameter + 1;
}
```

× ***Arrow notation should not be used here***

If you need to write an anonymous function, or an inline function that is not in the top level of a class, you must use arrow notation.

```
setInterval(() => {
    console.log("Spam");
}, 1);
```

✓ ***Acceptable***

Semicolon Usage

Although semicolons `;` are not required in JavaScript, **they will be required** after every statement in this project. This is mainly to stay consistent with our styling for Java.

9. Java

Javadoc

Javadoc is a convenient documentation generator for Java. It is required for all files written in Java. If a team does decide to use Javadoc, then every method and class must have thorough documentation. In this case, every method (excluding @Overrides) must have the `@param` and `@returns` tags. Any reference to another method or class should also come with a `@link` tag.

10. SQL

Naming Conventions

Table names must begin with a capital letter and be in camel case form.

Column names must begin with a lowercase letter and be in camel case form.

Stored procedure names must begin with a capital letter and be in camel case form.

Unique identifiers for a table should use the full/partial name of the table followed by "ID" in capital letters.

Foreign keys must use the same identifier as the column name they reference.

The schema and column name descriptions for our tables must be documented and available to the other teams.

Stored Procedures

Stored procedures must not use output parameters, the results must be selected (returned as rows) and handled by the recipient.

Stored procedures that use nested queries must use indentation before the nested query.

Stored procedures must all have a "definer" identified.

Stored procedure inputs and outputs must be documented and available to the other teams.

Stored procedures must have at least one comment for small stored procedures, along with other comments for procedures that involve multiple steps. These comments must describe the functionality of the stored procedure in a way that is simple to understand (as non-technical as possible).