Coding standards of the Project 'FindMeAPet'

1. Introduction

The present document reflects the coding standards of the Project *FindMeAPet*, which needs to be followed to maintain a clean, maintainable, and uniform code throughout the project.

This document is based on the *Code conventions for Java Programming Language* [1] and adapted for our needs and the coding languages of the project.

1.1. Why have coding conventions

As is mentioned in the *Code conventions for Java Programming Language* [1] Code conventions are important to programmers for several 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.

2. File Names

This section lists commonly used file suffixes and names.

2.1. File suffixes

This project should use the following suffixes:

File Type	Suffix
Javascript source	.js
Typescript source	.ts
HTML source	.html
CSS source	.css
Javascript testing file	.spec.js
Typescript testing file	.spec.ts

2.2. Common file names

These are common file names, avoid using files with these names with a different purpose.

File Name	Use
README.md	The preferred name for the file that summarizes the contents of a particular directory
package.json	The name for the file that contains the components dependencies
.gitignore	The name for the file containing the directories and files that git should ignore.

3. File organization

A file consists of sections that should be separated by a single black line and an optional identifying each section.

3.1. Max number of lines in a source code File

Files longer than 1000 lines of code are cumbersome and should be avoided in this project.

3.2. Require and import Statements

The firsts non-commented lines in a javascript/typescript should be the require/import statements.

3.3. Class and Interface Declarations

For the files containing classes or interfaces, the following table describes the parts of a class or interface declaration, in the order they should appear:

Order	Part of Class/Interface Declaration	Details
1	Class/Interface documentation comment (optional)	
2	Class or interface statement	
3	Static variables	First the public static variables, then the protected, then no access modifiers variables, and then the private.
4	Instance variables	First public, then protected, then no access modifiers, and then the private
5	Constructors (optional)	Constructors different from the default constructor
6	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

4. Indentation

Two spaces should be used as the unit of indentation. Tabs characters (\t) should be avoided.

4.1. Line Length

Avoid lines longer than 110 characters for this project to avoid scrolling horizontally on code editors.

4.2. Wrapping Lines

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.
- If defining a Javascript object, indent one level per object's hierarchy
- If listing elements (like a function's parameters, array's elements, etc.) just align the elements vertically adding one level of indentation to all elements and write the closing character in a new line indented to the beginning of the statement.

An example of the last principle could be:

```
private static horkingLongMethodName(
  anArg: number,
  anotherArg: Object,
  yetAnotherArg: string,
  andStillAnother: string
) {
  //Lines of code
}
```

Or defining an array:

```
let longArray = [
  'Hello everyone',
  'how are you?',
  'I am',
  'defining',
  'codign standards',
  'today',
];
```

5. Comments

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

In the Project it can be three types of comments:

5.1. Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code. For example:

Use it for documentation.

```
/**
 * Here is a block comment
 */
```

On the other hand, try to avoid block comments like this:

```
/*
  Here is a block comment
  It have multiple lines
*/
```

Also try to avoid letting block commented code.

5.2. Single line comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format (see section 5.1).

Example:

```
if (condition) {
  /* Handle the condition*/
  console.log("Some code here")
}
```

5.3. End of line Comments

The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code. Examples of all three styles follow

```
if (foo > 1) {
  // Do a double-flip.
} else {
  return false; // Explain why here.
}
//if (bar > 1) {
//
// Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

6. Declarations

6.1. Number per line

One declaration per line is recommended, since it is easier to read, and to find the declaration and type of that variable (if apply).

For example:

```
let variableA;
let variableB;
```

is preferred over:

```
let variableA, variableB;
```

6.2. Spaces

Only one space should separate the let/const keywords with the name of the variable.

6.3. Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

6.4. Class, Interfaces, and functions declarations

When coding Javascript/Typescript classes, interfaces and functions, the following formatting rules should be followed:

- No space between a method name and the parenthesis “(” starting its parameters list
- Open brace "{" appears at the end of the same line as the declaration statement
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

7. Statements

7.1. Simple statements

Each line should contain at most one statement. Example:

```
variableA++; //Correct
variableB++; //Correct
variableA++; variableB++; //Incorrect
```

7.2. Compound statements

Compound statements are statements that contain lists of statements enclosed in braces "{ statements }".

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

8. White space

8.1. Blank lines

Blank lines improve readability by setting off sections of code that are logically related.

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

- Between methods
- Before a block comment (see section 5.1).
- Between logical sections inside a method to improve readability.
- Between sections of a source file.
- Between class and interface definitions.

8.2. Blank spaces

A single blank space should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. (Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls).
- A blank space should appear after commas in argument lists.
- All binary operators should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.
- The expressions in a for statement should be separated by a blank space.

9. Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can clarify the code.

Identifier Type	Rules for Naming	Examples
Classes	Class names should be nouns, in pascal case with the first letter of each internal word capitalized. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	class Raster; class InageSprite;
Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate; interface Storing;

Methods	Methods should be verbs, in camel case with the first letter lowercase, with the first letter of each internal word capitalized.	run();
		runFast();
		getBackground();

Variables	Variables should be short yet meaningful in camel case with the first letter lowercase, with the first letter of each internal word capitalized. One-character variable names should be avoided except for temporary like i, j, k.	let i; let myWidth;
Constants and enums	The names of variables declared class constants and values of enums should be all uppercase with words separated by underscores ("_").	const MIN_WIDTH = 5; enum dias {MONDAY, TUESDAY, …}

10. Testing

10.1. Test Case Id

Test cases are an important part of the code, and we need to correctly identify every test case to the correctly identify them when some test cases fail. The identification of the test case should conform to the following convention:

2-3 Uppercase characters that uniquely identify the module to test + '-' + sequence number

Examples:

- For Storage Service: SS-01, SS-02, and so on.
- For Login Page: LP-01, LP-02, and so on.

References

[1] Oracle, 19 April 2018. [Online]. Available: https://www.oracle.com/java/technologies/javase/codeconventions-contents.html. [Último acceso: 18 June 2021].