AlarmBuddy Style Guide

Team Members:
Joshua Ringling: ring7885@stthomas.edu
Hannah Balut: balu2673@stthomas.edu
Alex Notch: notc6377@stthomas.edu
Mitzi Bustamante Chicaiza: bust6340@stthomas.edu

Background:

This style guide is to be used as a reference for all work being done related to the AlarmBuddy project. The attached guides were heavily inspired by style guidelines provided by Google under the CC-By 3.0 License, which encourages the sharing of these and the original source documents. The original style guides provided by Google can be found at https://google.github.io/styleguide/

Documentation Guidelines:

- Any services requiring accounts being used should have detailed instructions on how to create and login to an account.
- Any software or tools used in production such as IDEs should have detailed instructions on downloading and installing the software such that settings are the same used while making the project.

General Rules:

- Variables start with lower case letters and use context appropriate names. Use camel-case for variables that use multiple words.

```
int licenseNumber = 45301;
```

- Indent using tab to properly show scope. Examples: if statements, for loops, while loops, and functions.
- Use curly braces on all if-else statements, loops, and functions even when they are empty or only contain one line.
- Curly braces follow the Kernighan and Ritchie style (or “Egyptian brackets”) for code blocks:
    - 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 the current body of a method, constructor, or named class. For example, there is no line break after a brace if an else is being used.

```
if (x == 3) {
    for (int i = 0; i < 20; i++) {
        x = x + 1;
    }
} else {
    try {
        something();
    } catch (ProblemException e) {
        recover();
    }
}
```

- In mathematical operations, have spaces between all operators and operands *except* for parenthesis and use parenthesis to show clear mathematical steps.

```
int x = 2 * (1 + 1) ^ 2;
```

- Comment the use of functions as well as any special exceptions that may happen.
- Avoid making comments longer than 120 characters long. Try to wrap excessively long sections onto the next line.

```
//Returns an array of the same length as the input array
//but the output array has each data value square-rooted.
//If the input array has negative numbers in it, then an
//error is thrown.
```

- Try to initialize variables when declaring them. However, this is not always possible.

```
int x = 0;
String school = "St. Thomas";
```

JavaScript Style Guide

Joshua Ringling: ring7885@stthomas.edu

Background:

This document defines the formatting and style rules for JavaScript code. It applies to raw, working files that use JavaScript coding.

General Rules:

File Names:

File names must be all lowercase and may include underscores (_) or dashes (-) , but not additional punctuation.

The file's extension must be .js.

File Encoding:

All JavaScript files should be encoded using UTF-8.

Formatting Rules:

Braces:

Braces are required for all control structures (if, else, for, while, etc.), even if the body contains only a single statement.

All braces should follow the "Egyptian" brackets style mentioned above.

Empty Code Blocks:

An empty code block or block-like construct should be closed immediately if it contains nothing.

```
// Recommended
function doNothing() {}
```

Indentation:

Code blocks such as functions, if statements, or for loops should be indented using a single tab to properly show their scope.

Switch Statements:

Switch statements should be indented like any other block code.

The indentation for a switch statement example can be viewed down below.

All switch statements should have a default case.

```
// Recommended
switch (animal) {
    case Animal.REDPANDA:
      beAdorable();
      break;

    case Animal.PANDA:
      beADerp();
      break;

    default:
      throw new Error('Unknown animal');
  }
```

One Statement Per Line:

There should be only one statement on a line of code that is followed by a line-break.

Each statement must be terminated with a semicolon.

Line Wrapping:

JavaScript code should be limited to 80 characters per line. Any code that would exceed this limit must be line-wrapped.

There are no clear-cut rules for line wrapping other than to indent the wrapped code on its next line.

Try to wrap code to produce the best readability possible.

Comments:

Comments should be indented to the same degree as the code they are referring to.

There are many different styles of commenting. All are allowed, but be consistent.

```
/*
 * This is
 * okay.
 */

// And so
// is this.

/* This is fine, too. */
```

Language Features:

Const and Let:

Declare all local variables with const or let. Use const by default, unless a variable needs to be reassigned.

Do not use var to declare variables.

One Declaration At a Time:

Every local variable declaration should declare only one variable.

Do not use declarations such as let a = 0, b = 3;.

Array Literals:

Include a trailing comma whenever there is a line break between the final element and the closing bracket when declaring array values.

```
// Recommended
const values = [
    'some value',
    'another value',
  ];
```

Enums:

Enumerations are defined by adding the @enum annotation to an object literal. Additional properties should not be added to an enum after it is defined. Enums must be constant, and all enum values must be deeply immutable.

```
/**
 * Recommended
 * Supported temperature scales.
 * @enum {string}
 */
const TemperatureScale = {
    CELSIUS: 'celsius',
    FAHRENHEIT: 'fahrenheit',
  };
```

Overriding toString:

The toString method may be overridden, but must always succeed and never have visible side effects.

Arrow Functions:

Arrow functions are allowed anywhere an arrow function would make reasonable sense.

```
getJSON('/codes').then((result) => {
    app.codes = result;
    let code_dictionary = {};
    let i;
    for(i = 0; i<app.codes.length; i++) {
        code_dictionary[app.codes[i].code] = app.codes[i].type;
    }
    app.code_dictionary = code_dictionary;
}).catch((error) => {
    console.log('Error:', error);
});
```

Functions with no params and no return value should not use arrow functions.

```
// Not recommended
const someFunction = () => someOtherFunction();
```

String Literals:

Ordinary String literals are marked with single quotes (‘), rather than double quotes.

String Line Continuations:

Use the + operator to line wrap long String literals.

```
// Recommended
const longString = 'This is a very long string that far exceeds the 80 ' +
    'column limit. It does not contain long stretches of spaces since ' +
    'the concatenated strings are cleaner.';
```

This:

Only use this in class constructors and methods or in arrow functions defined within class constructors or methods.

Never use this to refer to a global object.

Equality Checks:

Use identity operators (=== / !==) except when trying to catch either null or undefined.

Constructor Omissions:

Never define a constructor with a new statement without using parentheses ( ) .

Naming Rules:

Camel-Case:

Methods and variables that use multiple words should use camel-case as described above.

Classes and Enums should also use camel-case, but start with a capital letter as well.

	Prose Form	Correct Camel-Case
Variable	"XML HTTP request"	XmlHttpRequest
Variable	"new customer ID"	newCustomerId
Class	"Online Timer App"	OnlineTimerApp

JSDoc:

General:

JSDoc should be used on all classes, fields, and methods.

General Form:

The basic formatting of JSDoc blocks can be seen in the example below.

```
/**
 * Multiple lines of JSDoc text are written here,
 * wrapped normally.
 * @param {number} x A number to do something to.
 */
function coolFunction(x) {
    doSomething(x);
}
```

Line Wrapping:

JSDoc lines that go past the 80 character should be wrapped onto the next line and indented like any other code.

File Overview:

A file should have a top-level file overview which should include a description of the file, but can also include a copyright notice, author information, or any other relevant document related information.

It is also recommended to include authors of files in the style of the below example.

```
/**
 * @author josh@someemail.com (Josh Ringling)
 * @fileoverview Description of file, its uses and information
 * about its dependencies.
 * @package
 */
```

Class Comments:

Classes and interfaces must be documented with a description and any template parameters and implemented interfaces. The class description should be detailed enough that the reader knows where this class could be used.

```
/**
 * A fancier event target that does cool things.
 * @implements {Iterable<string>}
 */
class MyFancyTarget extends EventTarget {
    /**
     * @param {string} arg1 An argument that makes this more interesting.
     * @param {!Array<number>} arg2 List of numbers to be processed.
     */
    constructor(arg1, arg2) {
        super();
        // ...
    }
};
```

Method and Function Comments:

In methods and named functions, parameter and return types must be documented.

Method descriptions begin with a verb phrase that describes what the method does.

If a method overrides a superclass method, it must include an @override annotation.

```
/**
 * Takes seconds since epoch and returns the current date and time in an
 * array with the month in array[0], the day in array[1], the year in
 * array[2], and the time in array[3].
 * @param {number} seconds
 * @return {Array.<string>}
 */
function makeArray(seconds) {
    blah blah blah
    return dateTime;
}
```