DUY TÂN UNIVERSITY

International School

Capstone Project 2

CMU-SE 451 – C2SE.12

Code Standard

Version 1.0

Date: April 18th, 2021

Learn English Together

Submitted by

Ha, Le Thanh

Hieu, Le Xuan

My, Ngo Ngoc

Thong, Doan Trung

Approved by

MSc Huy, Truong Dinh

Proposal Review Panel Representative:

____

Name Signature Date

Capstone Project 2- Mentor:

____

Name Signature Date

PROJECT INFORMATION

Project acronym	LET		
Project Title	Learn English Together		
Start Date	26 Feb 2021	**End Date**	08 Jun 2021
Lead Institution	International School, Duy Tan University		
Project Mentor	MSc Huy, Truong Dinh		
Scrum master / Project Leader & contact details	Ha, Le Thanh Email: lethanhhadtu@gmail.com Tel: 0334002818		
Partner Organization	Duy Tan University		
Project Web URL			
Team members	Name	Email	Tel
	Ha, Le Thanh	lethanhhadtu@gmail.com	0334002818
	Hieu, Le Xuan	xuanhieu.le.1999@gmail.com	0399706614
	My, Ngo Ngoc	ngongocmy851999@gmail.com	0764497391
	Thong, Doan Trung	doanthong002@gmail.com	0886428208

CODE STANDARD DOCUMENT

Document Title	Code Standard Document		
Author(s)	H2MT Team		
Role	Product Owner, Team Member, Scrum Master		
Date	April 18th, 2021	**File name:**	C2SE.22_LET_CodeStandard_ver1.0.doc
URL			
Access	Project and CMU Program		

REVISION HISTORY

Version	**Person(s)**	**Date**	**Description**
1.0	**Le Thanh Ha**	April 18th, 2021	Draft for comment

DOCUMENT APPROVALS

The following signatures are required for approval of this document.

Ha, Le Thanh Student ID: 2321122516 *Scrum Master*	Signature	Date
Hieu, Le Xuan Student ID: 2321124665 *Team Member*	Signature	Date
My, Ngo Ngoc Student ID: 2321124970 *Team Member*	Signature	Date
Thong, Doan Trung Student ID: 2321124144 *Team Member*	Signature	Date

TABLE OF CONTENTS

1. Introduction

This document aims to set out a standard to programming languages to Prevent COVID-19 Website project that includes JavaScript should conform. It is expected that were appropriate changes be made to these standards to adapt them for the language in question.

2. Code Conventions for the Java Script Programming Language

2.1 Imperative and structured

JavaScript supports much of the structured programming syntax from C (e.g., if statements, while loops, switch statements, do while loops, etc.). One partial exception is scoping: JavaScript originally had only function scoping with var. ECMAScript 2015 added keywords let and const for block scoping, meaning JavaScript now has both function and block scoping. Like C, JavaScript makes a distinction between expressions and statements. One syntactic difference from C is automatic semicolon insertion, which allows the semicolons that would normally terminate statements to be omitted.

2.2 Weakly typed

JavaScript is weakly typed, which means certain types are implicitly cast depending on the operation used.

- The binary `+` operator casts both operands to a string unless both operands are numbers. This is because the addition operator doubles as a concatenation operator
- The binary `-` operator always casts both operands to a number
- Both unary operators (`+`, `-`) always cast the operand to a number

Values are cast to strings like the following:

- Strings are left as-is
- Numbers are converted to their string representation
- Arrays have their elements cast to strings after which they are joined by commas (`,`)
- Other objects are converted to the string `[object Object]` where `Object` is the name of the constructor of the object

Values are cast to numbers by casting to strings and then casting the strings to numbers. These processes can be modified by defining `toString` and `valueOf` functions on the prototype for string and number casting respectively.

JavaScript has received criticism for the way it implements these conversions as the complexity of the rules can be mistaken for inconsistency. For example, when adding a number to a string, the number will be cast to a string before performing concatenation, but when subtracting a number from a string, the string is cast to a number before performing subtraction.

Left operand	Operator	Right operand	Result
[] (empty array)	+	[] (empty array)	"" (empty string)
[] (empty array)	+	{} (empty object)	"[object Object]" (string)
false (Boolean)	+	[] (empty array)	"false" (string)
"123"(string)	+	1 (number)	"1231" (string)
"123" (string)	-	1 (number)	122 (number)

Often also mentioned is {} + [] resulting in 0 (number). This is misleading: the {} is interpreted as an empty code block instead of an empty object, and the empty array is cast to a number by the remaining unary + operator. If you wrap the expression in parentheses ({} + []) the curly brackets are interpreted as an empty object and the result of the expression is "[object Object]" as expected.

2.3 Dynamic

2.3.1 Typing

JavaScript is dynamically typed like most other scripting languages. A type is associated with a value rather than an expression. For example, a variable initially bound to a number may be reassigned to a string. JavaScript supports various ways to test the type of objects, including duck typing.

2.3.2 Run-time evaluation

JavaScript includes an eval function that can execute statements provided as strings at run-time.

2.4 Object-orientation (prototype-based)

Prototypal inheritance in JavaScript is described by Douglas Crockford as:

You make prototype objects, and then … make new instances. Objects are mutable in JavaScript, so we can augment the new instances, giving them new fields and methods. These can then act as prototypes for even newer objects. We don't need classes to make lots of similar objects…

In JavaScript, an object is an associative array, augmented with a prototype (see below); each string key provides the name for an object property, and there are two syntactical ways to specify such a name: dot notation (obj. x = 10) and bracket notation (obj['x'] = 10). A property may be added, rebound, or deleted at run-time. Most properties of an object (and any property that belongs to an object's prototype inheritance chain) can be enumerated using a for...in loop.

2.4.1 Prototypes

JavaScript uses prototypes where many other object-oriented languages use classes for inheritance. It is possible to simulate many class-based features with prototypes in JavaScript.

2.4.2 Functions as object constructors

Functions double as object constructors, along with their typical role. Prefixing a function call with new will create an instance of a prototype, inheriting properties and methods from the constructor (including properties from the Object prototype). ECMAScript 5 offers the Object. Create method, allowing explicit creation of an instance without automatically inheriting from the Object prototype (older environments can assign the prototype to null). The constructor's prototype property determines the object used for the new object's internal prototype. New methods can be added by modifying the prototype of the function used as a constructor. JavaScript's built-in constructors, such as Array or Object, also have prototypes that can be modified. While it is possible to modify the Object prototype, it is generally considered bad practice because most objects in JavaScript will inherit methods and properties from the Object prototype, and they may not expect the prototype to be modified.

2.4.3 Functions as methods

Unlike many object-oriented languages, there is no distinction between a function definition and a method definition. Rather, the distinction occurs during function calling; when a function is called as a method of an object, the function's local this keyword is bound to that object for that invocation.

2.5 Functional

A function is first-class; a function is an object. As such, a function may have properties and methods, such as. call() and .bind(). A nested function is a function defined within another function. It is created each time the outer function is invoked. In addition, each nested function forms a lexical closure: The lexical scope of the outer function (including any constant, local variable, or argument value) becomes part of the internal state of each inner function object, even after execution of the outer function concludes. JavaScript also supports anonymous functions.

2.6 Delegative

JavaScript supports implicit and explicit delegation.

2.6.1 Functions as roles (Traits and Mixins)

JavaScript natively supports various function-based implementations of Role patterns like Traits and Mixins. Such a function defines additional behavior by at least one method bound to this keyword within its function body. A Role then has to be delegated explicitly via call or apply to objects that need to feature additional behavior that is not shared via the prototype chain.

2.6.2 Object composition and inheritance

Whereas explicit function-based delegation does cover composition in JavaScript, implicit delegation already happens every time the prototype chain is walked in order to, e.g., find a method that might be related to but is not directly owned by an object. Once the method is found it gets called within this object's context. Thus, inheritance in JavaScript is covered by a delegation automatism that is bound to the prototype property of constructor functions.

2.7 Miscellaneous

2.7.1 Run-time environment

JavaScript typically relies on a run-time environment (e.g., a web browser) to provide objects and methods by which scripts can interact with the environment (e.g., a web page DOM). These environments are single-threaded. JavaScript also relies on the run-time environment to provide the ability to include/import scripts (e.g., HTML <script> elements). This is not a language feature per se, but it is common in most JavaScript implementations. JavaScript processes messages from a queue one at a time. JavaScript calls a function associated with each new message, creating a call stack frame with the function's arguments and local variables. The call stack shrinks and grows based on the function's needs. When the call stack is empty upon function completion, JavaScript proceeds to the next message in the queue. This is called the event loop, described as "run to completion" because each message is fully processed before the next message is considered. However, the language's concurrency model describes the event loop as non-blocking: program input/output is performed using events and callback functions. This means, for instance, that JavaScript can process a mouse click while waiting for a database query to return information.

2.7.2 Array and object literals

Like many scripting languages, arrays and objects (associative arrays in other languages) can each be created with a succinct shortcut syntax. In fact, these literals form the basis of the JSON data format.

2.7.3 Promises

JavaScript also supports promises, which are a way of handling asynchronous operations. There is a built-in Promise object that gives access to a lot of functionalities for handling promises, and defines how they should be handled. It allows one to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future. Recently, combinator methods were introduced in the JavaScript specification, which allows developers to combine multiple JavaScript promises and do operations on the basis of different scenarios. The methods introduced are: Promise.race, Promise.all, Promise.allSettled and Promise.any.

2.8 Syntax

2.8.1 Case sensitivity

JavaScript is case sensitive. It is common to start the name of a constructor with a capitalized letter, and the name of a function or variable with a lower-case letter.

```
var a=5;
console.log(a); // 5
console.log(A); // throws a
Reference Error: A is not defined
```

2.8.2 Whitespace and semicolons

The five problematic tokens are the open parenthesis "(", open bracket "[", slash "/", plus "+", and minus "-". Of these, the open parenthesis is common in the immediately-invoked function expression pattern, and open bracket occurs sometimes, while others are quite rare. The example given in the spec is:

```
a = b + c
(d + e).foo()

// Treated as:
// a = b + c (d + e).foo();
```

with the suggestion that the preceding statement be terminated with a semicolon.

Some suggest instead the use of leading semicolons on lines starting with '(' or '[', so the line is not accidentally joined with the previous one. This is known as a defensive semicolon, and is particularly recommended, because code may otherwise become ambiguous when it is rearranged. For example:

```
;(d + e).foo()

// Treated as:
//  a = b + c;
//  (d + e).foo();
```

Initial semicolons are also sometimes used at the start of JavaScript libraries, in case they are appended to another library that omits a trailing semicolon, as this can result in ambiguity of the initial statement.

The five restricted productions are return, throw, break, continue, and post-increment/decrement. In all cases, inserting semicolons does not fix the problem, but makes the parsed syntax clear, making the error easier to detect. return and throw take an optional value, while break and continue take an optional label. In all cases, the advice is to keep the value or label on the same line as the statement. This most often shows up in the return statement, where one might return a large object literal, which might be accidentally placed starting on a new line. For post-increment/decrement, there is potential ambiguity with pre-increment/decrement, and again it is recommended to simply keep these on the same line.

```
return
a + b;

// Returns undefined. Treated as:
//   return;
//   a + b;
// Should be written as:
//   return a + b;
```

2.8.3 Comments

Comment syntax is the same as in C++, Swift and many other languages.

```
/* this is a long, multi-line comment
  about my script. May it one day
  be great. */

/* Comments /* may not be nested */ Syntax error */
```

2.9 Variables

Variables in standard JavaScript have no type attached, and any value can be stored in any variable. Before ES6, variables were declared only with a var statement. Starting with ES6, the version of the language finalized in 2015, variables can also be declared with let or const which are for block level variables. The value assigned to a const cannot be changed, but its properties can. An identifier must start with a letter, underscore (_), or dollar sign ($); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, the uppercase characters "A" through "Z" are different from the lowercase characters "a" through "z".

Starting with JavaScript 1.5, ISO 8859-1 or Unicode letters (or \uXXXX Unicode escape sequences) can be used in identifiers. In certain JavaScript implementations, the at sign (@) can be used in an identifier, but this is contrary to the specifications and not supported in newer implementations.

2.9.1 Scoping and hoisting

Variables declared with `var` are lexically scoped at a function level, while ones with `let` or `const` have a block level scope. Declarations are processed before any code is executed. This is equivalent to variables being forward declared at the top of the function or block, and is referred to as *hoisting*.

With `var` the variable value is `undefined` until it is initialized, and forward reference is not possible. Thus, a `var x = 1` statement in the middle of the function is equivalent to a `var x` declaration statement at the top of the function, and an `x = 1` assignment statement at that point in the middle of the function – only the declaration is hoisted, not the assignment. Variables declared with `let` or `const` do not set the value to `undefined`, so until it is initialized, referencing the variable will cause an error.

Function statements, whose effect is to declare a variable of type `Function` and assign a value to it, are similar to variable statements, but in addition to hoisting the declaration, they also hoist the assignment – as if the entire statement appeared at the top of the containing function – and thus forward reference is also possible: the location of a function statement within an enclosing function is irrelevant.

Be sure to understand that

```
var func = function() { .. } // will NOT be hoisted
function func() { .. } // will be hoisted
```

Block scoping can be produced by wrapping the entire block in a function and then executing it – this is known as the immediately-invoked function expression pattern – or by declaring the variable using the `let` keyword.

2.10 Primitive data types

The JavaScript language provides six primitive data types:

- Undefined
- Null
- Number
- String
- Boolean
- Symbol

Some of the primitive data types also provide a set of named values that represent the extents of the type boundaries. These named values are described within the appropriate sections below.

```
// Declares a function-scoped variable named `x`, and implicitly assigns the
// special value `undefined` to it. Variables without value are automatically
// set to undefined.
var x;

// Variables can be manually set to `undefined` like so
var x2 = undefined;

// Declares a block-scoped variable named `y`, and implicitly sets it to
// `undefined`. The `let` keyword was introduced in ECMAScript 2015.
let y;

// Declares a block-scoped, un-reassign-able variable named `z`, and sets it to
// a string literal. The `const` keyword was also introduced in ECMAScript 2015,
// and must be explicitly assigned to.

// The keyword `const` means constant, hence the variable cannot be reassigned
// as the value is `constant`.
const z = "this value cannot be reassigned!";

// Declares a variable named `myNumber`, and assigns a number literal (the value
// `2`) to it.
let myNumber = 2;

// Reassigns `myNumber`, setting it to a string literal (the value `"foo"`).
// JavaScript is a dynamically-typed language, so this is legal.
myNumber = "foo";
```

2.11 Native Objects

2.11.1 Array

An Array is a JavaScript object prototyped from the Array constructor specifically designed to store data values indexed by integer keys. Arrays, unlike the basic Object type, are prototyped with methods and properties to aid the programmer in routine tasks (for example, join, slice, and push).

As in the C family, arrays use a zero-based indexing scheme: A value that is inserted into an empty array by means of the push method occupies the 0th index of the array.

```
var myArray = [];            // Point the variable myArray to a newly ...
                        // ... created, empty Array
myArray.push("hello World"); // Fill the next empty index, in this case 0
console.log(myArray[0]);          // Equivalent to console.log("hello World");
```

2.11.2 Date

A Date object stores a signed millisecond count with zero representing 1970-01-01 00:00:00 UT and a range of ±108 days. There are several ways of providing arguments to the Date constructor. Note that months are zero-based.

```
new Date();                    // create a new Date instance representing the current
time/date.
new Date(2010, 2, 1);            // create a new Date instance representing 2010-
Mar-01 00:00:00
new Date(2010, 2, 1, 14, 25, 30); // create a new Date instance representing
2010-Mar-01 14:25:30
new Date("2010-3-1 14:25:30");    // create a new Date instance from a String.
```

2.11.3 Error

Custom error messages can be created using the Error class:

```
throw new Error("Something went wrong.");
```

2.12 Related technologies

2.12.1 JSON

JSON, or JavaScript Object Notation, is a general-purpose data interchange format that is defined as a subset of JavaScript's object literal syntax.

3. References

WIKIPEDIA
[https://en.wikipedia.org/wiki/JavaScript#Growth_and_standardization]
[https://en.wikipedia.org/wiki/JavaScript#Imperative_and_structured]
[https://en.wikipedia.org/wiki/JavaScript_syntax]