

MicroCODE Software Engineering Services

Project Coding Standards

MCX-S02 (Internal JS Style Guide) v006.docx

Development Environment: Microsoft Visual Studio Code

OS: HTML5 Browser Environment

Platform: Open Source ES6, HTML5, CSS3

Language: JavaScript (ECMAScript 6 – ES6)

Tools: JSLint, JSHint

MicroCODE JavaScript Style Guide

This was adopted from the **MIT xPRO JavaScript Certificate Style Guide**, differences are noted. The MIT guide was in turn adapted from the [Airbnb style guide](#) and [the Turing School style guide](#).

Everything we believe is based on this statement:

Code like a Machine: Consistently, Explicitly, Simply, and for Readability (CAESAR).

All our rules are marked with one or more of these to relate back to this statement:

(Code Consistently) (Code Explicitly) (Code Simply) (Code for Readability)

Table of Contents

Standard Terminology – MicroCODE additions

Font Selection and Readability – MicroCODE addition

Indents, Braces, Blocks, and Aligned-Code – MicroCODE addition

Whitespace

Types

Objects

Arrays

Strings

Functions

Arrow Functions – MicroCODE addition

Properties

Variables

Comparison Operators & Equality

Blocks

Commas

Semicolons

Naming Conventions

Common Patterns – MicroCODE addition

Appendix A: JS Class Structure – MicroCODE addition

Appendix B: VS Code Settings – MicroCODE addition



Standard Terminology (MicroCODE)

Any Style Guide should start with **coding terminology** because there is a general lack of standardized terms.

`()` – these are **PARENS**, short for parentheses, they are not ‘round brackets’ or ‘round braces’, parens are, by our definition, a pair of round delimiters.

`[]` – these are **BRACKETS**. They are not braces, they are not ‘square brackets’, brackets are, by our definition, a pair of square delimiters.

`{ }` – these are **BRACES**. They are not brackets, they are not ‘curly brackets’ nor ‘curly braces’, braces are by our definition a pair of a ‘curly’ delimiters. Saying ‘curly braces’ is redundant, saying ‘curly brackets’ is just wrong.

`< >` – these are **ANGLES**. They are not brackets, they are not ‘angled brackets’ nor ‘angled braces’, angles are, by our definition, a pair of angular delimiters.

In typography all of these are referred to generically as ‘brackets’. A nice article explaining this: <https://type.today/en/journal/brackets>. But we are not talking about typesetting, we are talking about coding, and precise language eliminates confusion and mistakes; it produces consistent results and saves time & money.

And so, for all our code and documentation... ([Code Consistently](#))

Code Delimiters

```
( ) = PARENS:   parameter grouping, operand precedence, quantities
[ ] = BRACKETS: indexing, array formation
{ } = BRACES:   code blocks, initialization values (compiler)
< > = ANGLES:  substitution identifier, option grouping
```

Boolean Expressions

```
& = AND: inclusive
| = OR:  optional
! = NOT: exclusive

() = AND Group, everything within is AND'ed
<> = OR Group, everything within is OR'ed
```

Boolean Equations

```
RPO ABC <XYZ ABO YUT> !ZZY Z71
RPO ABC <XYZ (ABO BBC !CDC) YUT> !ZZY Z71
```

Implies, in more detail...

```
&RPO &ABC <XYZ ABO YUT> !ZZY &Z71
&RPO &ABC <XYZ (ABO BBC !CDC) YUT> !ZZY &Z71
```

Implies, in complete notation...

```
&RPO &ABC &<XYZ|ABO|YUT> &!ZZY &Z71
&RPO &ABC &<XYZ|(&ABO &BBC &!CDC)|YUT> &!ZZY &Z71
```



Standard Terminology (continued)

Interlocks, Signals, and Memories vary in duration, persistence, and interactions. This should be understood immediately by the 'Parameter Persistence Type'. ([Code Consistently](#))

ONE-SHOT – an application variable—usually a BOOLEAN—that goes from FALSE to TRUE for a single program scan or routine execution.

RISING-EDGE – a signal condition—regarding a BOOLEAN—that indicates the variable has gone from FALSE to TRUE during this program scan or routine execution.

FALLING-EDGE – a signal condition—regarding a BOOLEAN—that indicates the variable has gone from TRUE to FALSE during this program scan or routine execution.

PULSE – an application variable—usually a BOOLEAN—that goes from FALSE to TRUE for specified (or configured) duration, normally specified in milliseconds.

INTERLOCK – an application variable—any PRIMITIVE or OBJECT—that is presented to external equipment or a remote application until it is Acknowledged (ACK/NAK). This implies retries and failure annunciation if not acknowledged.

MEMORY – an application variable—any PRIMITIVE or OBJECT—that changes state based on program scan or routine execution. These can be *temporary* or *persistent*.

CONTEXT – the sum total of all the application's **Persistent Memory** that control its behavior. Persistent Memory should be stored permanently from one session to another for a consistent User Experience (UX). The quality of this persistence is what gives a user confidence in a system or application, providing the feeling that 'it remembers what I was doing, and I did not have to configure anything for that to happen'.

CONFIGURATION – a special subset of the application's **Persistent Memory** that represents User Preferences or User Settings, usually configured by them thru the User Interface (UI) thru the 'Gear' or 'Cog' Icon. These settings must be protected thru all Application Upgrades as they represent 'User Work Products'. These settings must be written to permanent data storage, i.e.: a configuration file or database.

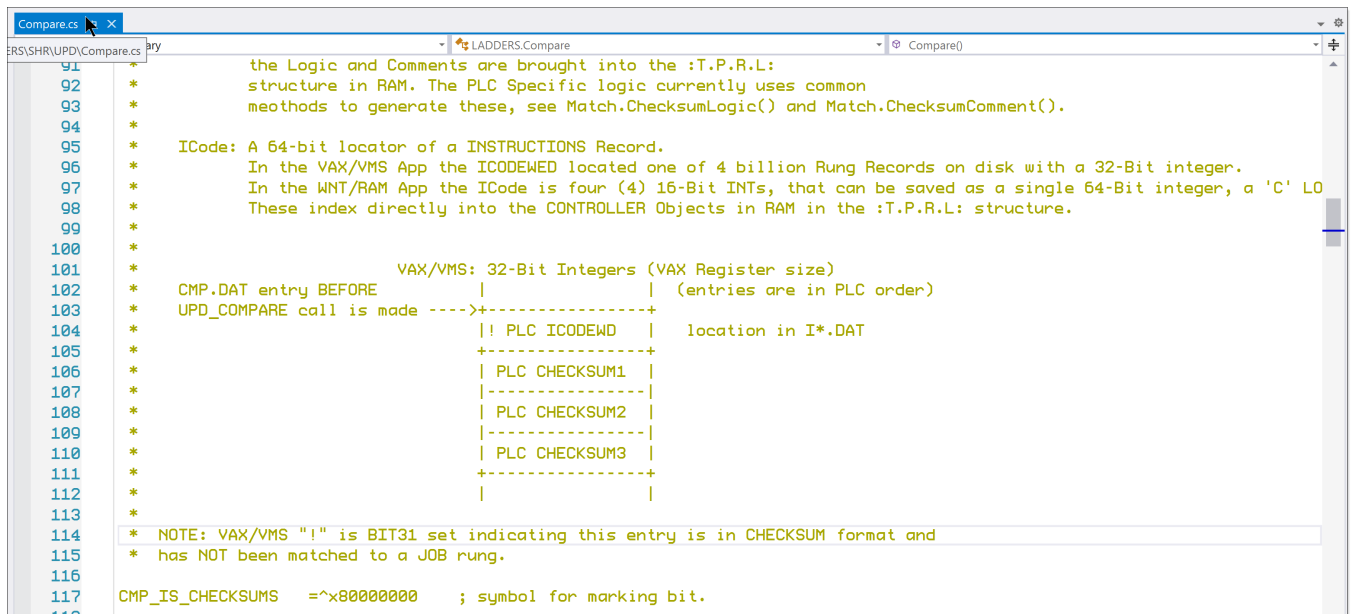


Font Selection and Readability (MicroCODE)

We believe you should always code in a mono-spaced font. (Code Consistently)

The only explanation required is a couple simple examples:

Monospace – enables the creation of text tables in documentation.

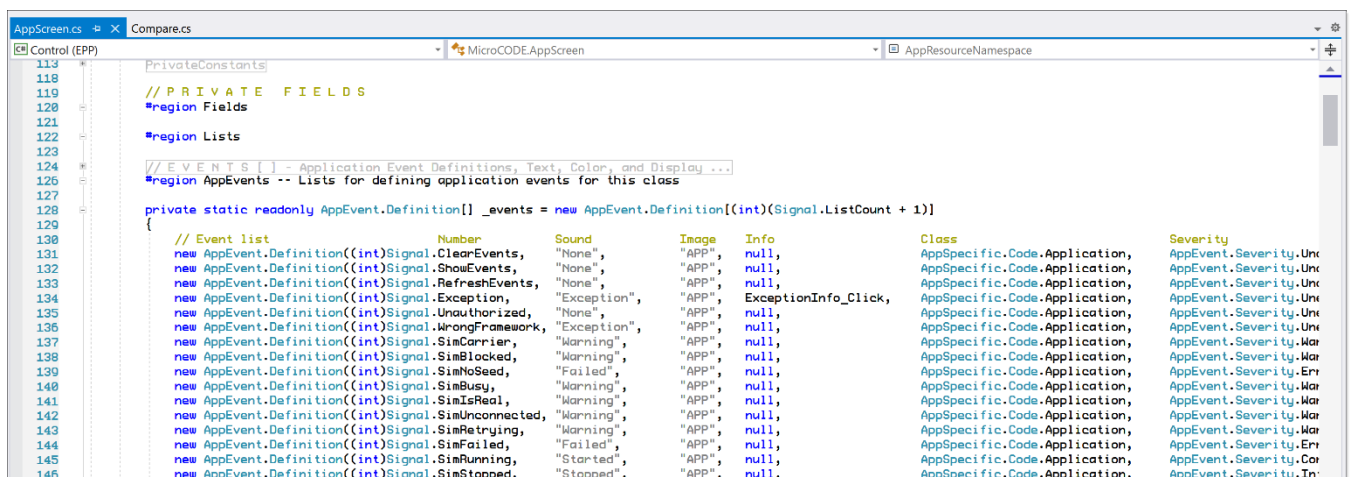


```

Compare.cs
ERS\SHR\UPD\Compare.cs
LADDERS.Compare
Compare()

91 * the Logic and Comments are brought into the :T.P.R.L:
92 * structure in RAM. The PLC Specific logic currently uses common
93 * methods to generate these, see Match.ChecksumLogic() and Match.ChecksumComment().
94 *
95 * ICode: A 64-bit locator of a INSTRUCTIONS Record.
96 * In the VAX/VMS App the ICODEWD located one of 4 billion Rung Records on disk with a 32-Bit integer.
97 * In the WNT/AM App the ICode is four (4) 16-Bit INTs, that can be saved as a single 64-Bit integer, a 'C' LO
98 * These index directly into the CONTROLLER Objects in RAM in the :T.P.R.L: structure.
99 *
100 *
101 * VAX/VMS: 32-Bit Integers (VAX Register size)
102 * CMP.DAT entry BEFORE |-----+ (entries are in PLC order)
103 * UPD_COMPARE call is made ---->+-----+
104 * |! PLC ICODEWD | location in I*.DAT
105 * +-----+
106 * | PLC CHECKSUM1 |
107 * |-----+
108 * | PLC CHECKSUM2 |
109 * |-----+
110 * | PLC CHECKSUM3 |
111 * +-----+
112 *
113 *
114 * NOTE: VAX/VMS "I" is BIT31 set indicating this entry is in CHECKSUM format and
115 * has NOT been matched to a JOB rung.
116
117 CMP_IS_CHECKSUMS = ^x80000000 ; symbol for marking bit.
118
  
```

Monospace – makes coded tables instantly readable and exceptions pop-out.



```

AppScreen.cs
Compare.cs
Control (EPP)
MicroCODE.AppScreen
AppResourceNamespace

118 PrivateConstants
119 // PRIVATE FIELDS
120 #region Fields
121
122 #region Lists
123
124 // EVENTS [ ] - Application Event Definitions, Text, Color, and Display ...
125 #region AppEvents -- Lists for defining application events for this class
126
127 private static readonly AppEvent.Definition[] _events = new AppEvent.Definition[(int)(Signal.ListCount + 1)]
128 {
129     // Event list
130     new AppEvent.Definition((int)Signal.ClearEvents, "None", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Unk),
131     new AppEvent.Definition((int)Signal.ShowEvents, "None", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Unk),
132     new AppEvent.Definition((int)Signal.RefreshEvents, "None", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Unk),
133     new AppEvent.Definition((int)Signal.Exception, "Exception", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Unk),
134     new AppEvent.Definition((int)Signal.Unauthorized, "None", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Unk),
135     new AppEvent.Definition((int)Signal.WrongFramework, "Exception", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Unk),
136     new AppEvent.Definition((int)Signal.SimCarrier, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Unk),
137     new AppEvent.Definition((int)Signal.SimBlocked, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.War),
138     new AppEvent.Definition((int)Signal.SimNoSeed, "Failed", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Err),
139     new AppEvent.Definition((int)Signal.SimBusy, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.War),
140     new AppEvent.Definition((int)Signal.SimIsReal, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.War),
141     new AppEvent.Definition((int)Signal.SimUnconnected, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.War),
142     new AppEvent.Definition((int)Signal.SimRetrying, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.War),
143     new AppEvent.Definition((int)Signal.SimFailed, "Failed", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Err),
144     new AppEvent.Definition((int)Signal.SimRunning, "Started", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Cor),
145     new AppEvent.Definition((int)Signal.SimStopped, "Stopped", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.In)
146 }
  
```



Proportional – creates a mess. A must for User documentation, bad for code.

```

191 * the Logic and Comments are brought into the :T.P.R.L:
192 * structure in RAM. The PLC Specific logic currently uses common
193 * methods to generate these, see Match.ChecksumLogic() and Match.ChecksumComment().
194 *
195 * ICode: A 64-bit locator of a INSTRUCTIONS Record.
196 * In the VAX/VMS App the ICODEWED located one of 4 billion Rung Records on disk with a 32-Bit integer.
197 * In the WNT/NT App the ICode is four (4) 16-Bit INTs, that can be saved as a single 64-Bit integer, a 'C' LONG.
198 * These index directly into the CONTROLLER Objects in RAM in the :T.P.R.L: structure.
199 *
200 *
201 * VAX/VMS: 32-Bit Integers (VAX Register size)
202 * CMP.DAT entry BEFORE | (entries are in PLC order)
203 * UPD_COMPARE call is made --> +-----+
204 * |! PLC ICODEWD | location in I*.DAT
205 * +-----+
206 * | PLC CHECKSUM1 |
207 * |-----|
208 * | PLC CHECKSUM2 |
209 * |-----|
210 * | PLC CHECKSUM3 |
211 * +-----+
212 * | |
213 *
214 * NOTE: VAX/VMS "I" is BIT31 set indicating this entry is in CHECKSUM format and
215 * has NOT been matched to a JOB rung.
216
217 CMP_IS_CHECKSUMS =^x80000000 : symbol for marking bit.
218

```

Proportional – creates a mess, coded tables are much harder to read.

```

120 #region Fields
121
122 #region Lists
123
124 // EVENTS[] - Application Event Definitions, Text, Color, and Display...
125 #region AppEvents - Lists for defining application events for this class
126
127 private static readonly AppEvent.Definition[] _events = new AppEvent.Definition[(int)Signal.ListCount + 1]
128 {
129 // Event list
130 // Number Sound Image Info Class Severity Audience Repeatable Persistent Targets
131 new AppEvent.Definition((int)Signal.ClearEvents, "None", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Undefined, AppEvent.Audience.Undefined, false, false, false),
132 new AppEvent.Definition((int)Signal.ShowEvents, "None", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Undefined, AppEvent.Audience.Operator, false, false, false),
133 new AppEvent.Definition((int)Signal.RefreshEvents, "None", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Undefined, AppEvent.Audience.Operator, false, false, false),
134 new AppEvent.Definition((int)Signal.Exception, "Exception", "APP", ExceptionInfo_Click, AppSpecific.Code.Application, AppEvent.Severity.Unexpected, AppEvent.Audience.Developer, true, true, true),
135 new AppEvent.Definition((int)Signal.Unauthorized, "None", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Unexpected, AppEvent.Audience.Developer, true, true, true),
136 new AppEvent.Definition((int)Signal.WrongFramework, "Exception", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Unexpected, AppEvent.Audience.Developer, true, true, true),
137 new AppEvent.Definition((int)Signal.SimCarrier, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Warning, AppEvent.Audience.Operator, true, true, true),
138 new AppEvent.Definition((int)Signal.SimBlocked, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Warning, AppEvent.Audience.Operator, true, true, true),
139 new AppEvent.Definition((int)Signal.SimNoSeed, "Failed", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Error, AppEvent.Audience.Operator, true, true, true),
140 new AppEvent.Definition((int)Signal.SimBusy, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Warning, AppEvent.Audience.Operator, true, true, true),
141 new AppEvent.Definition((int)Signal.SimIsReal, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Warning, AppEvent.Audience.Operator, true, true, true),
142 new AppEvent.Definition((int)Signal.SimUnconnected, "Warning", "APP", null, AppSpecific.Code.Application, AppEvent.Severity.Warning, AppEvent.Audience.Operator, true, true, true),

```



```

0 1 2 3 4 5 6 7 8 9 0
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
@ # $ % ^ & * ( ) [ ] { } < > _ + - = \ / |
; : ' " , . ? ! ` ~

```

Note: [11:16:55 AM 24-Oct-2023] Update. Google has released a great coding font named: “Fira Code”



Indents, Braces, Blocks, and Aligned-Code (MicroCODE)

Common JavaScript Style Guides like to use Kernighan & Richie (K&R) 'Egyptian Brackets'.

Right off the bat, they are really not talking about "brackets" they are talking about "braces".

Our opinion is that this style 'saves' one line break per block of code at the expense of readability.

Why? Because the key word expressions like 'if' and 'else' overlap and offset the code of the clause.

This style saved screen space when people coded on 24-Line CRTs (like the VT52s, VT100s, VT220s, etc.) and saved paper when people printed code. We have neither of these limitations in the 21st Century; white space is your friend, delimiter alignment is your friend, language keyword alignment is your friend.

Note: Aligned-Code is also known as **Allman Style** (after Eric Allman) or **BSD Style** – Berkeley Software Distribution.

```
if (condition) {  
    // true code  
} else {  
    // false code  
}  
  
try {  
    // protected code  
} catch {  
    // correction code  
} finally {  
    // exit code  
}
```

K&R Style

The above is K&R, below is BSD which is our preferred JavaScript Style (and C/C++/C# Style as well).

The braces are aligned.

The key words are aligned.

All code blocks are held within aligned braces.

All conditional code has natural white space around it via the braces.

People will argue that K&R is easy to read, but they are relying on colorized text editors that are hiding the readability issue in the bare text. **There is no world in which K&R is easier to read and maintain than BSD.**

```
if (condition)  
{  
    // true code  
}  
else  
{  
    // false code  
}  
  
try  
{  
    // protected code  
}  
catch  
{  
    // correction code  
}  
finally  
{  
    // exit code  
}
```

BSD Style



Notice in the second example the conditional code is naturally separated by white space created by the balanced braces, this makes the code far more readable than the 'Egyptian Brackets'. And there are no excuses based on typing speed, as these preferences can all be enforced automatically by VS CODE Settings.

White space is your friend and aids in readability; half the effort of maintaining code is readability and consistency.

As an extension, when building conditional execution, braces should always be used—and all code blocks should be placed on the lines following the conditional expression.

```
if (condition) ... // true code - DO NOT DO THIS

if (condition)
  ... // true code - DO NOT DO THIS

if (condition)
{
  ... // true code - ALWAYS DO THIS, conditional code in a BLOCK of BRACES, 'ALIGNED'
  ... // here is example a line added in the future, no reformatting required
}
```

This first reason for this is consistency and long-term maintenance. If the first two examples ever need editing—where an additional line of code needs to be added to the execution clause—the first thing that must be done is the addition of braces and the reformatting of the lines... a burden placed on the future coder by the original author. If line(s) of code are added in a hurry—and no braces are added—you just cost the next coder (or your future self) a good :10 minutes figuring out why their condition code is not working. It just not worth it.

The second reason is readability, it's just far easier to read all conditional code the same way, everywhere in a project, if it is all formatted exactly the same way.

It is often said: "Cleanliness is Next to Godliness", in coding we believe "Consistency is Next to Godliness".

The universe works on a set of consistent rules for a reason. If you code consistently—always following the same rules, always following the same patterns, where all code looks like it came from the same author, where all Classes follow the same format—you and your Team will reap the benefits now and forever. If your code is utterly consistent your Team members (or an A.I. Assistant like ChatGPT or GitHub Copilot) can code with predictions that will always be true... **that** will speed up development more than any style can by saving typing white space or braces.

See **Appendix A: MicroCODE JS Class Structure** as an example of using a Template to make following these rules simple.

Our Mantra: "**Code like a Machine: Consistently and Explicitly, Simply and for Readability** (Hail CEASAR)."SM

The combination of BSD Style, with the MCODE Rules, like always starting new modules from approved templates—no matter how simple the new Class or Module may be—we call...

MCODE Style

We've marked all AirBnB Rules that are eliminated as a side-effect of MCODE Style with...

MCODE Style eliminates



Whitespace

- Use soft tabs set to ~~2 spaces~~ 4 spaces. (Code for Readability).
- **Note:** Use 8 spaces for Assembly Language files from 1980s and 1990s or you will destroy diagrams, tabular data, etc. as these were created in CRT / VT100 environments.

```
// bad
function foo() {
var name;
}

// bad
function bar() {
var name;
}

// good
function baz()
{
var name;
}
```

MCODE Style eliminates

- ~~Place 1 space before the leading curly brace.~~
- Start all code blocks on new lines with aligned braces. (Code Consistently, Code for Readability).

```
// bad
function test(){
console.log('test');
}

// good
function test()
{
console.log('test');
}
```

MCODE Style eliminates

- ~~Place 1 space before the opening parenthesis in control statements (if, else if, while etc.).~~
- Start all code blocks on new lines with aligned braces. (Code Consistently, Code for Readability).

```
// bad
if(isSithLord) {
fight();
}

// good
if (isSithLord)
{
fight();
}
```

MCODE Style eliminates



- Place no space before the argument list in function calls and declarations. ([Code Consistently](#)).

```
// bad
function fight () {
  console.log ('Swoosh!');
}

// good
function fight()
{
  console.log('Swoosh!');
}
```

MCODE Style eliminates

- Set off operators with spaces.

```
// bad
var x=y+5;

// good
var x = y + 5;
```

- Leave a blank line after blocks and before the next statement. ([Code Consistently](#), [Code for Readability](#)).
- **Note:** This rule makes no sense when 'Egyptian Brackets' are used.
- **Never place more than one blank line between code lines of any type.**

```
// bad
if (foo) {
  return bar;
}
return baz;

// good
if (foo)
{
  return bar;
}

return baz;

// bad
var obj = {
  foo: function () {},
  bar: function () {}
};
return obj;

// good
var obj =
{
  foo: function () {},
  bar: function () {}
};

return obj;
```

MCODE Style eliminates



Types

- Use native data types for any variables that are not application specific or persistent. ([Code Simply](#)).

```
// bad
var x = {};
x.counter = 0;
for (x.counter = 0; x.counter < 99; x.counter++)
{
    // work
}

// good
for (let counter = 0; counter < 99; counter++)
{
    // work
}
```

- Use the simplest data structures possible. ([Code Simply](#)).



Objects

- Use the literal syntax for generic object creation. ([Code Simply](#)).

```
// bad
var item = new Object();

// good
var item = {};
```

- Don't use [reserved words](#) as keys. It won't work in IE8. [More info](#). ([Code Explicitly](#)).

```
// bad
var superman = {
  default: { clark: 'kent' },
  private: true
};

// good
var superman =
{
  alterEgo: { clark: 'kent' },
  hidden: true
};
```

- Use readable synonyms in place of reserved words. ([Code Explicitly](#)).

```
// bad
var superman = {
  class: 'alien'
};

// bad
var superman = {
  klass: 'alien'
};

// good
var superman =
{
  species : 'kryptonian'
};
```



Arrays

- Use the literal syntax for array creation. ([Code Simply](#)).

```
// bad
var items = new Array();

// good
var items = [];
```

- Use `Array.push` instead of direct assignment to add items to an array. ([Code Simply](#)).

```
var someStack = [];

// bad
someStack[someStack.length] = 'bohemianrhapsody';

// good
someStack.push('bohemianrhapsody');
```

- `.push` (add to end), `.pop` (remove from end) work on the **end** of the array
- `.unshift` (add to start), `.shift` (remove from start) work on the **start** of the array
- **Note:** `.shift` SHIFTS LEFT, `.unshift` SHIFTS RIGHT



Strings

- Use single quotes `'` for most strings, even though double-quotes `"` are allowed. ([Code Consistently](#)).
- Use double quotes `"` for string containing single quotes, or escape `'` in strings as `\'`.
- Use back-ticks (grave accents) ``` for template literals or template strings. ([JavaScript language](#)).

```
// preferred - no quotes
var name = 'Bob Barker';

// preferred - with apostrophes
var name = "Bob's Books";

// ok
var name = 'Bob\'s Books';

// bad
var fullName = "Bob " + this.lastName;

// preferred, and JS required
var fullName = `Bob ${this.lastName}`;
```

- Use back-ticks (grave accents) ``` for string holding double-quotes. ([Code Consistently](#)).
- Why? Because JavaScript functions are often building HTTP or CSS code that has embedded double-quoted values which must be double-quoted.

```
ui.result =
  <div class="card text-white bg-success mb-3" style="max-width: 40rem;">
    <div class="card-header" id="resultTitle">Result Display</div>
    <div class="card-body">
      <h5 class="card-title">Transaction Completed</h5>
      <label for="outputResult" class="col-sm-40 col-form-label">Result</label>
      <div class="col-sm-40">
        <input type="text" id="outputResult" value="<result>">
      </div>
    </div>
  </div>;
```



Functions

- Always use Named Function expressions. ([Code Consistently](#), [Code Explicitly](#)).
- **Note:** It helps when viewing stack-traces during debugging.

```
// anonymous function expression
var anonymous = function()
{
    return true;
};

// named function expression
var named = function named()
{
    return true;
};
```

- Never name a parameter “arguments”. This will take precedence over the arguments object that is given to every function scope. ([Code Explicitly](#)).

```
// bad
function dontDoThis(name, options, arguments) {
    // ...stuff...
}

// good
function thisIsOk(name, options, qualities)
{
    // ...stuff...
}
```

Arrow Functions

- Always code ‘fat arrow’ functions the same way, with Aligned Braces {} and a return statement. ([Code Consistently](#), [Code Explicitly](#)).

```
const x = (x, y) => x * y; // bad - removal of option return and braces
const x = (x, y) => { return x * y }; // bad - single line exception
const x = (x, y) =>
{
    return x * y // good - Code Consistently, Code for Readability
};
```



Properties

- Use **dot** (.) notation when accessing properties by name. ([Code Simply](#), [Code Consistently](#), [Code for Readability](#)).

```
var luke = {  
  jedi: true,  
  age: 28  
};  
  
// bad  
var isJedi = luke['jedi'];  
  
// good  
var isJedi = luke.jedi;
```

- Use **bracket notation** [] when accessing properties with a variable. ([Code Explicitly](#)).

```
var luke = {  
  jedi: true,  
  age: 28  
};  
  
function getProp(prop)  
{  
  return luke[prop];  
}  
  
var isJedi = getProp('jedi');
```



Variables

- Always use `const` or `let` to declare variables. Not doing so will result in global variables. We want to avoid polluting the global namespace. Globals should only be used for app 'context', or 'common memory', usually loaded from a preferences files, settings files, or app 'memory' files. ([Code Explicitly](#)).

```
// bad - automatic global created
superPower = new SuperPower();

// good - local and 'readonly'
const superPower = new SuperPower();
```

- Use one `const` or `let` declaration per variable or assignment. ([Code Simply](#), [Code Explicitly](#)).

Why? It's easier to add new variable declarations this way, and you never have to worry about swapping out a `;` for a `,` or introducing punctuation-only diffs. You can also step through each declaration with the debugger, instead of jumping through all of them at once.

```
// bad
const items = getItems(),
      goSportsTeam = true,
      dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
const items = getItems(),
      goSportsTeam = true;
      dragonball = 'z';

// good
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```



- Group all your **const** and then group all your **let**. ([Code for Readability, Code Consistently](#)).
- See **Appendix MicroCODE JS Code Ordering under JS Class Structure**

Why? This is helpful later on you might need to assign a variable depending on one of the previously assigned variables.

```
// bad
let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
let i;
const items = getItems();
let dragonball;
const goSportsTeam = true;
let len;

// CONSTANTS -- good
const goSportsTeam = true;
const items = getItems();

// FIELDS -- good
let dragonball;
let i;
let length;
```

MCODE Style eliminates



Comparison Operators & Equality

- Use `===` and `!==` over `==` and `!=`. ([Code Explicitly](#)).
- Conditional statements such as the `if` statement evaluate their expression using coercion with the `.ToBoolean` abstract method and always follow these simple rules:
 - **Objects** evaluate to **true**
 - **Undefined** evaluates to **false**
 - **Null** evaluates to **false**
 - **Booleans** evaluate to **the value of the boolean**
 - **Numbers** evaluate to **false** if **+0, -0, or NaN**, otherwise **true**
 - **Strings** evaluate to **false** if an empty string "", otherwise **true**

```
if ([0]) {  
  // true  
  // An array (even an empty one) is an object, objects evaluate to true  
}
```

- Use shortcuts for **Booleans**, but explicit comparisons for strings and numbers. For more information see [Truth Equality and JavaScript](#) by Angus Croll. ([Code Simply](#), [Code Explicitly](#)).

```
// bad - boolean  
if (isValid === true) {  
  // ...stuff...  
}  
  
//good - boolean  
if (isValid)  
{  
  //...stuff  
}  
  
// bad - string  
if (name) {  
  //...stuff  
}  
  
// good - string  
if (name !== '') or if (name.length > 0)  
{  
  //...stuff  
}  
  
// bad - array  
if (collection.length) {  
  //...stuff  
}  
  
// good - array  
if (collection.length > 0)  
{  
  //...stuff  
}
```



Blocks

- **Use** Always use code blocks in braces **with all multi-line blocks** for conditionals. (Code Consistently).
- See **MicroCODE: Indents, Braces, Blocks, and Aligned-Code**.

```
// bad
if (test)
    return false;

// bad
if (test) return false;

// good
if (test)
{
    return false;
}

// bad
function () { return false; }

// good
function ()
{
    return false;
}
```

MCODE Style eliminates

- **If you're using** When using **multi-line blocks with if** and **else**, put **else on the same line as your if block's closing brace** on a new line, aligned with the **if**. (Code Consistently).
- See **MicroCODE: Indents, Braces, Blocks, and Aligned-Code**.

```
// bad
if (test) {
    thing1();
    thing2();
} else {
    thing3();
}

// good - this will always be easier to read and maintain than the above
if (test)
{
    thing1();
    thing2();
}
else
{
    thing3();
}
```

MCODE Style eliminates



Commas

- Leading commas: **Nope.** ([Code for Readability](#)).
- What barbarian would ever do this?

```
// bad
var story = [
  once
  , upon
  , aTime
];

// good
var story =
[
  once,
  upon,
  aTime
];

// bad
var hero = {
  firstName: 'Bob'
  , lastName: 'Parr'
  , heroName: 'Mr. Incredible'
  , superPower: 'strength'
};

// good
var hero =
{
  firstName: 'Bob',
  lastName: 'Parr',
  heroName: 'Mr. Incredible',
  superPower: 'strength'
};
```



- Additional trailing comma: **Nope.** This can cause problems with IE6/7 and IE9 if it's in quirksmode. Also, in some implementations of ES3 would add length to an array if it had an additional trailing comma. This was clarified in ES5 ([source](#)):

Edition 5 clarifies the fact that a trailing comma at the end of an Array Initializer does not add to the length of the array. This is not a semantic change from Edition 3, but some implementations may have previously misinterpreted this.

- While it's tempting to leave the last comma to making adding future items easier the last trailing comma misleads the compiler/interpreter, and you never want to do that. ([Code Explicitly](#)).

```
// bad
var hero = {
  firstName: 'Kevin',
  lastName: 'Flynn',
};

var heroes = [
  'Batman',
  'Superman',
];

// good
var hero =
{
  firstName: 'Kevin',
  lastName: 'Flynn'
};

var heroes =
[
  'Batman',
  'Superman'
];
```



Semicolons

- **Yup.** ([Code Explicitly](#)).
- Just get used to semicolons, otherwise you are relying on the compiler/interpreter to figure out what you wanted across multiple lines of code. And sometimes it will make a different decision than you will.

```
// bad - raises exception
const luke = {}
const leia = {}
[luke, leia].forEach((jedi) => jedi.father = 'vader')

// bad - raises exception
const reaction = "No! That's impossible!"
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
})();

// bad - returns `undefined` instead of the value on the next line - always happens when
// `return` is on a line by itself because of ASI!
function foo() {
  return
  'search your feelings, you know it to be foo'
}

// good
const luke = {};
const leia = {};
[luke, leia].forEach((jedi) => {
  jedi.father = 'vader';
});

// good
const reaction = "No! That's impossible!";
(async function meanwhileOnTheFalcon() {
  // handle `leia`, `lando`, `chewie`, `r2`, `c3p0`
  // ...
})();

// good
function foo() {
  return 'search your feelings, you know it to be foo';
}
```



Naming Conventions

- Avoid single letter names. Be descriptive with your naming. ([Code Explicitly](#), [Code for Readability](#)).

```
// bad
function q() {
  // ...stuff...
}

// good
function sizeQuery()
{
  // ..stuff..
}
```

- Use **camelCase** when naming **objects**, **functions**, and **instances**. ([Code for Readability](#)).

```
// bad
var OBJEcttsssss = {};
var this_is_my_object = {};
var o = {};
function c() {}

// good
var thisIsMyObject = {};
function thisIsMyFunction() {}
```

- Use **PascalCase** when naming **classes** or **components**. ([Code for Readability](#)).

```
// bad
class user {
  constructor(options) {
    this.name = options.name;
  }
}

var bad = new user({
  name: 'nope'
});

// good
class User
{
  constructor(options)
  {
    this.name = options.name;
  }
}

var user = new User(
{
  name: 'yup'
});
```



- Name your functions. This is helpful for stack traces. (Code Explicitly, Code for Readability).
- **Note:** IE8 and below exhibit some quirks with named function expressions. See <http://kangax.github.io/nfe/> for more info.

```
// bad
var log = function (msg) {
  console.log(msg);
};

// good
var log = function log(msg)
{
  console.log(msg);
};
```

- Start PRIVATE METHOD names within a Class with an '_' (Underscore).
- And document the method as `@api private` in the method header.

```
// METHODS - PRIVATE
/**
 * _method1() - description of private method.
 *
 * @param1 {type} description of param1.
 * @returns {type} description of return value.
 * @api private
 */
_method1(param1)
{
  // ...

  return value;
}
```

- Start all PRIVATE FIELD names within a Class with a '#' (Pound Sign / Hash).

```
// PRIVATE FIELDS
#property1 = 0.00;
#property2 = '';

// CONSTRUCTOR
constructor(objectName)
{
  this.type = CLASS_TYPE;
```

- Make all CONSTANTS UPPERCASE and **static**

```
// CONSTANTS
static MIN_VALUE = 1;
static MAX_VALUE = 999;
static CLASS_TYPE = 'Example';
```



Common Patterns

Note: Value comparisons, test variable against a limit, not a limit against a variable...
(This is known as a 'Yoda Conditional' and is like saying "if blue is the sky"). ([Code for Readability](#)).

Use...

```
if (ObjectBeingTested == TestValue)
{
    ...
}
```

Not...

```
if (TestValue == ObjectBeingTested)
{
    ...
}
```

Example:

Use...

```
if (Object == null)
{
    ...
}
```

Not...

```
if (null == Object)
{
    ...
}
```

Reason:

The habit of placing the Object being tested first matches the way we speak, "If the object is null" and it makes ordered comparisons logically correct like this...

Use...

```
if (Object >= LowLimit)
{
    ...
}
```

Not...

```
if (LowLimit <= Object)
{
    ...
}
```




Code Maintenance


Follow a consistent procedure for editing released code. ([Code Consistently](#)).

1 – Follow a 'Quality Control Checklist' as you build and document a Release or Hot Fix.

Don't wait until you are done to 'pencil whip' the form, it is meant to be used as you go through the process.

 MicroCODE App Release Process Quality Control Checklist	
Background This defines the steps to successfully release MicroCODE Apps to Clients. App: Control (EPP) Build: v2.0.1 b (9)	STEP 6a: Document--with user requirements--the NEW FEATURES being added. These go into the RELEASE NOTES at the end under "New in Version vM.m.R c (B)" Document the Test Cases. STEP 6b: Document--with screen shots if helpful--the CORRECTED ISSUES being resolved. These go into the RELEASE NOTES at the end under "Correct in Version vM.m.R c (B)" Document the Test Cases. NOTE: Both 'New' and 'Corrected' go into the Release Notes regardless. If nothing changed note "No features were added" or "No issues were corrected".
PREPARATION STEP 1: Make a copy of the most recent INTERNAL source code tree. <u>Do not edit a stored copy of the 'Internal' source code tree.</u> STEP 2: Rename it 'D:\MicroCODE\Internal' -- all coding tools reference this path.	

The Checklist should include documentation templates for the README.md, Release Notes and Test Cases...

MicroCODE App Release Process Quality Control Checklist 	
New in Version vM.m.R c (B) The following features were added in this Release: 1) Requirement: ... Implementation: ... TEST CASES: These software tests were performed prior to release to ensure App functionality. Test Case N: ... – Passed. Test Case N: ... – Conditional. Test Case N: ... – Failed.	Corrected in Version VM.m.R c (B) The following defects were fixed this Release: 1) Issue: ... Correction: ... TEST CASES: These software tests were performed prior to release to ensure App functionality. Test Case N: ... – Passed. Test Case N: ... – Conditional. Test Case N: ... – Failed.



Appendix A: MicroCODE JS Class Structure

According to the [ES6 Rules Documentation](#)—and most JavaScript Style Guides—the ordering of the Class elements is shown below. This is the ordering enforced by the MicroCODE JS Class Template. ([Code Explicitly](#), [Code for Readability](#), [Code for Readability](#)).

Within a **class** (`ClassName`) group **datatypes** in this order:

- Constant Variables/Fields (`const`)
- Fields (`let`)
- Constructor (`constructor`)
- Enums (`const Object.freeze({})`)
- Properties (`get`, `set`)
- Iterators (`[Symbol.*]`)
- Methods (`methodName(param)`)
- Nested Classes

Within each of **type** group order by **access**:

- public
- private

Within each of the **access** groups, order by **static**, then **non-static**:

- static
- non-static

Within each of the **static/non-static** groups of fields, order by **readonly**, and then **non-readonly**:

- readonly
- non-readonly

e.g.: The methods section then would be unrolled in this order:

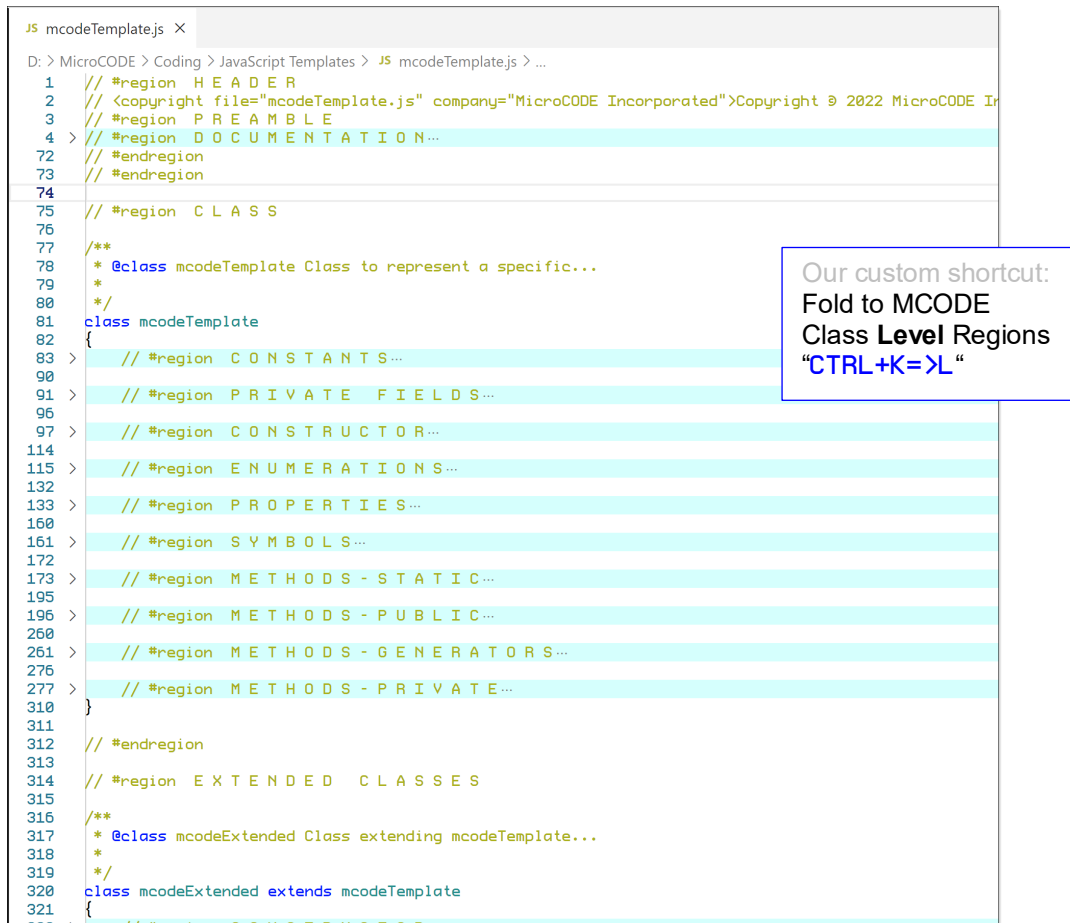
- public static methods
- public methods
- private static methods
- private methods



JS Class Structure (MicroCODE)

`mcodeTemplate` is technically a function (the one that we provide as constructor), while methods, getters and setters are written to `mcodeTemplate.prototype`. Below is our actual JavaScript Class Template file used to start all new JS Classes in our projects. ([Code Consistently](#)).

This template is built with **Code Folding** support through use of “`// #region`” and “`// #endregion`”. This will seem like a lot of unnecessary ‘syntactic sugar’—I hate that phrase BTW—reading through the following and understanding what it enables in VS CODE should make the value clear. Start with the entire file ‘folded’ to Class level. “`CTRL+K=>3`”.



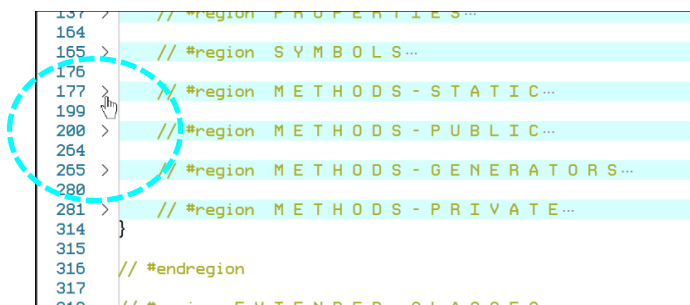
```

1 // #region H E A D E R
2 // <copyright file="mcodeTemplate.js" company="MicroCODE Incorporated">Copyright © 2022 MicroCODE Inc
3 // #region P R E A M B L E
4 // #region D O C U M E N T A T I O N...
72 // #endregion
73 // #endregion
74
75 // #region C L A S S
76
77 /**
78  * @class mcodeTemplate Class to represent a specific...
79  *
80  */
81 class mcodeTemplate
82 {
83 > // #region C O N S T A N T S...
84
85 > // #region P R I V A T E   F I E L D S...
86
87 > // #region C O N S T R U C T O R...
88
89 > // #region E N U M E R A T I O N S...
90
91 > // #region P R O P E R T I E S...
92
93 > // #region S Y M B O L S...
94
95 > // #region M E T H O D S - S T A T I C...
96
97 > // #region M E T H O D S - P U B L I C...
98
99 > // #region M E T H O D S - G E N E R A T O R S...
100
101 > // #region M E T H O D S - P R I V A T E...
102
103 }
104 // #endregion
105 // #region E X T E N D E D   C L A S S E S
106
107 /**
108  * @class mcodeExtended Class extending mcodeTemplate...
109  *
110  */
111 class mcodeExtended extends mcodeTemplate
112 {
113 > // #region C O N S T R U C T O R
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

Our custom shortcut:
Fold to MCODE
Class **Level** Regions
“`CTRL+K=>3`”

Then ‘unfold’ the area you are working my clicking the “`>`” for that Region.



```

157 > // #region P R O P E R T I E S...
164 > // #region S Y M B O L S...
176 > // #region M E T H O D S - S T A T I C...
177 > // #region M E T H O D S - P U B L I C...
199 > // #region M E T H O D S - G E N E R A T O R S...
200 > // #region M E T H O D S - P R I V A T E...
264 > // #region E X T E N D E D   C L A S S E S
265 >
280 >
281 >
314 >
315 >
316 > // #endregion
317 > // #region E X T E N D E D   C L A S S E S
318 >

```



This leaves everything you are not working on 'hidden' via the 'folded code' support in VS Code.

```

118 >
119 > // #region ENUMERATIONS...
136
137 > // #region PROPERTIES...
164
165 > // #region SYMBOLS...
176
177 > // #region METHODS - STATIC
178
179 > /**
180  * static1() - description of public static method, called by prototype not object.
181  * This does not operate on a specific copy of a Class object.
182  * @api public
183  *
184  * @param {type} param1 description of param1.
185  * @returns {type} description of return value.
186  *
187  * @example
188  *
189  *     static1('param1');
190  */
191 > static static1(param1)
192 {
193     // ...
194
195     return value;
196 }
197
198 // #endregion
199
200 > // #region METHODS - PUBLIC...
264
265 > // #region METHODS - GENERATORS...

```

All the 'code folding' can be opened with... "CTRL+K=>J".

```

JS mcodeTemplate.js x
D:\> MicroCODE > Coding > JavaScript Templates > JS mcodeTemplate.js > mcodeTemplate

57 *
58 *   MODIFICATIONS:
59 *   -----
60 *
61 *   Date:      By-Group:  Rev:      Description:
62 *
63 *   02-Feb-2022 TJM-MCODE {0001} New module for common reusable Javascript Classes for code
64 *   06-Feb-2022 TJM-MCODE {0002} Moved @api tag under method description and before @param
65 *   08-Feb-2022 TJM-MCODE {0003} Added @constructor, @property, @class, @enum,
66 *
67 *
68 */
69 "use strict";
70
71 // #endregion
72 // #endregion
73 // #endregion
74
75 // #region CLASS
76
77 /**
78  * @class mcodeTemplate Class to represent a specific...
79  *
80  */
81 class mcodeTemplate
82 {
83     // #region CONSTANTS
84
85     static MIN_VALUE = 1;
86     static MAX_VALUE = 999;
87     static CLASS_TYPE = 'Example';
88
89     // #endregion
90
91     // #region PRIVATE FIELDS
92     #property1 = 0.00;
93     #property2 = '';
94
95     // #endregion
96
97     // #region CONSTRUCTOR

```

Standard shortcut:
Unfold ALL Regions
"CTRL+K=>J"



Everything can be 'folded' to LEVEL 0 with... "CTRL+K=>0".

```
JS mcodeTemplate.js X
D: > MicroCODE > Coding > JavaScript Templates > JS mcodeTemplate.js > ...
1 > // #region H E A D E R|...
74
75 > // #region C L A S S ...
313
314 > // #region E X T E N D E D   C L A S S E S ...
346
```

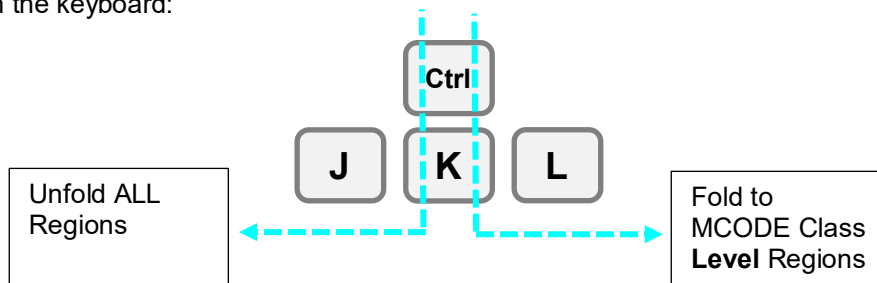
And you can restore your view of the overall Class structure (locked by **MCODE Style**) at any time with...

"CTRL+K=>J" "CTRL+K=>3".

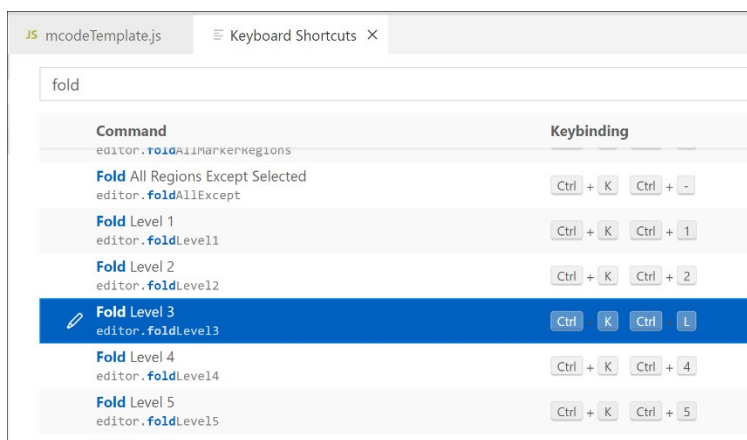
NOTE: You cannot get the same view with "CTRL+K=>3" without starting with "CTRL+K=>J". These commands are 'relative' to the current view, not 'absolute'... i.e.: "CTRL+K=>3" will not always give you the same result by itself.

I use this all day every day when navigating files, so I have remapped "CTRL+K=>L" to equal "CTRL+K=>3".

So, on the keyboard:



- File → Preferences → Keyboard Shortcuts



```
// #region H E A D E R
// <copyright file="mcodeTemplate.js" company="MicroCODE Incorporated">Copyright © 2022 MicroCODE Incorporated Troy,
MI</copyright><author>Timothy J. McGuire</author>
// #region P R E A M B L E
// #region D O C U M E N T A T I O N
/*
 * Title:      MicroCODE Common JavaScript Class Template
 * Module:     Modules (MicroCODE:mcodeTemplate.js)
 * Project:    MicroCODE Common Library
 * Customer:   Internal
 * Creator:    MicroCODE Incorporated
 * Date:       February 2022
 * Author:     Timothy J McGuire
 *
 * Designed and Coded: 2022 MicroCODE Incorporated
 *
 * This software and related materials are the property of
 * MicroCODE Incorporated and contain confidential and proprietary
 * information. This software and related materials shall not be
 * duplicated, disclosed to others, or used in any way without the
 * written of MicroCODE Incorporated.
 *
 * DESCRIPTION:
 * -----
 *
 * This module implements the MicroCODE's Common JavaScript Class Template.
 * This file is copied to start all MicroCODE JavaScript code files.
 *
 * REFERENCES:
 * -----
 *
 * 1. MIT xPRO Style Guide
 *    https://student.emeritus.org/courses/3291/files/2554233/download?wrap=1
 *
 * 2. AirBnB JavaScript Style Guide
 *    https://github.com/airbnb/javascript
 *
 * 3. Turing School Style Guide
 *    https://github.com/turingschool-examples/javascript/tree/main/es5
 *
 * 4. MDN Web Docs - JavaScript Classes
 *    https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
 *
 * 5. JSDoc - How to properly document JavaScript Code.
 *    https://
 *
 * 4. MicroCODE MCX-S02 (Internal JS Style Guide).docx
 *
 * DEMONSTRATION VIDEOS:
 * -----
 *
 * 1. ...
 *
 * MODIFICATIONS:
 * -----
 *
 * Date:      By-Group:  Rev:      Description:
 *
 * 02-Feb-2022  TJM-MCODE  {0001}  New module for common reusable Javascript Classes for code files.
 * 06-Feb-2022  TJM-MCODE  {0002}  Moved @api tag under method description and before @param and @return.
 * 08-Feb-2022  TJM-MCODE  {0003}  Aded @constructor, @property, @class, @enum,
 *
 */
"use strict";

// #endregion
// #endregion
// #endregion

// #region C L A S S

/**
 * @class mcodeTemplate Class to represent a specific...
 */
class mcodeTemplate
{
    // #region C O N S T A N T S

    static MIN_VALUE = 1;
    static MAX_VALUE = 999;
    static CLASS_TYPE = 'Example';

```



```

// #endregion

// #region PRIVATE FIELDS
@property1 = 0.00;
@property2 = '';
// #endregion

// #region CONSTRUCTOR
/**
 * @constructor mcodeTemplate class constructor.
 *
 * @param {string} objectName the name of the object.
 */
constructor (objectName)
{
    this.type = CLASS_TYPE;
    this.name = objectName;
    this.enums = [];

    // ...
}

// #endregion

// #region ENUMERATIONS
/**
 * @enum namedEnum1 - a description of this enum, its use, and meaning.
 */
static namedEnum1 = Object.freeze
(
{
    name1: 0,
    name2: 1,
    name3: 2,
    name4: 3,
    name5: 4,
    name6: 5,
    name7: 6
}
));

// #endregion

// #region PROPERTIES
/**
 * @property {type} property1 a description of this property, its use, and meaning.
 */
get property1()
{
    return this.#property1;
}
set property1(value)
{
    this.#property1 = value;
}

/**
 * @property {type} property2 a description of this property, its use, and meaning.
 */
get property2()
{
    return this.#property2;
}
set property2(value)
{
    this.#property2 = value;
}

// #endregion

// #region SYMBOLS
/**
 * iterator1 - a description of this iterator, its use, and meaning.
 */
[Symbol.iterator]()
{
    // method with computed name (symbol here)
}

// #endregion

// #region METHODS - STATIC
/**
 * static1() - description of public static method, called by prototype not object.
 * This does not operate on a specific copy of a Class object.

```



```

* @api public
* @param {type} param1 description of param1.
* @returns {type} description of return value.
* @example
*
*     static1('param1');
*/
static static1(param1)
{
    // ...

    return value;
}

// #endregion
// #region METHODS - PUBLIC
/**
* method1() - description of public method.
* @api public
* @param {type} param1 1st method parameter.
* @returns method result.
* @example
*
*     method1('param1');
*/
method1(param1)
{
    // ...

    return value;
}

/**
* method2() - description of public method.
* @api public
* @param {type} param1 1st method parameter.
* @param {type} param2 2nd method parameter.
* @returns method result.
* @example
*
*     method2('param1', 'param2');
*/
method2(param1, param2)
{
    // ...

    return value;
}

/**
* method3() - description of public method.
* @api public
* @param {type} param1 1st method parameter.
* @param {type} param2 2nd method parameter.
* @param {type} param3 3rd method parameter.
* @returns {type} method result.
* @example
*
*     method3('param1', 'param2');
*/
method3(param1, param2, param3)
{
    // ...

    return value;
}

// #endregion
// #region METHODS - GENERATORS
/**
* getValue() - returns all values in 'enums'.
*
*/

```



```

*getValue()
{
  for (const enumValue of this.enums)
  {
    yield value;
  }
}

// #endregion

// #region METHODS - PRIVATE

/**
 * _method1() - description of private method.  NOTE: Method headers are *optional* for private methods.
 * @api private
 * @param {type} param1 description of param1.
 * @returns {type} description of return value.
 */
_method1(param1)
{
  // ...

  return value;
}

/**
 * _method2() - description of private method.
 * @api private
 * @param {type} param1 description of param1.
 * @returns {type} description of return value.
 */
_method2(param1)
{
  // ...

  return value;
}

// #endregion
}

// #endregion

// #region EXTENDED CLASSES

/**
 * @class mcodeExtended Class extending mcodeTemplate...
 *
 */
class mcodeExtended extends mcodeTemplate
{
  // #region CONSTRUCTOR

  constructor (name)
  {
    super(name); // call the super class constructor and pass in the name parameter
  }

  // #endregion

  // #region METHODS - PUBLIC

  /**
   * method9() -- ouputs the extended name of the Class.
   *
   */
  method9()
  {
    console.log(`${ this.name } communicates.`);
  }

  // #endregion
}

// #endregion

```



Appendix B: Visual Studio Code Settings

These are the VS Code settings recommended to maintain **MCODE Style** automatically.

- Automatic Spaces

Editor: Insert Spaces

- ☒ Insert spaces when pressing **Tab**. This setting is overridden based on the file contents when [Editor: Detect Indentation](#) is on.

Editor > Comments: Insert Space

- ☒ Controls whether a space character is inserted when commenting.

- When to auto-format

Editor: Format On Paste

- ☒ Controls whether the editor should automatically format the pasted content. A formatter must be available and the formatter should be able to format a range in a document.

Editor: Format On Save

- ☒ Format a file on save. A formatter must be available, the file must not be saved after delay, and the editor must not be shutting down.

Editor: Format On Save Mode

Controls if format on save formats the whole file or only modifications. Only applies when [Editor: Format On Save](#) is enabled.

file



Editor: Format On Type

- ☒ Controls whether the editor should automatically format the line after typing.



- JavaScript auto-formatting options (same for TypeScript)

JavaScript › Format: Enable

☒ Enable/disable default JavaScript formatter.

JavaScript › Format: Insert Space After Comma Delimiter

☒ Defines space handling after a comma delimiter.

JavaScript › Format: Insert Space After Constructor

☒ Defines space handling after the constructor keyword.

JavaScript › Format: Insert Space After Function Keyword For Anonymous Functions

☒ Defines space handling after function keyword for anonymous functions.

JavaScript › Format: Insert Space After Keywords In Control Flow Statements

☒ Defines space handling after keywords in a control flow statement.

JavaScript › Format: Insert Space After Opening And Before Closing Empty Braces

☒ Defines space handling after opening and before closing empty braces.

JavaScript › Format: Insert Space After Opening And Before Closing Jsx Expression Braces

☐ Defines space handling after opening and before closing JSX expression braces.

JavaScript › Format: Insert Space After Opening And Before Closing Nonempty Braces

☒ Defines space handling after opening and before closing non-empty braces.

JavaScript › Format: Insert Space After Opening And Before Closing Nonempty Brackets

☐ Defines space handling after opening and before closing non-empty brackets.

JavaScript › Format: Insert Space After Opening And Before Closing Nonempty Parenthesis

- JavaScript auto-formatting options (continued) (same for TypeScript)

JavaScript › Format: Insert Space After Opening And Before Closing Nonempty Parenthesis

☐ Defines space handling after opening and before closing non-empty parenthesis.

JavaScript › Format: Insert Space After Opening And Before Closing Template String Braces

☒ Defines space handling after opening and before closing template string braces.

JavaScript › Format: Insert Space After Semicolon In For Statements

☒ Defines space handling after a semicolon in a for statement.

JavaScript › Format: Insert Space Before And After Binary Operators

☒ Defines space handling after a binary operator.

JavaScript › Format: Insert Space Before Function Parenthesis

☐ Defines space handling before function argument parentheses.

JavaScript › Format: Place Open Brace On New Line For Control Blocks

☒ Defines whether an open brace is put onto a new line for control blocks or not.

JavaScript › Format: Place Open Brace On New Line For Functions

☒ Defines whether an open brace is put onto a new line for functions or not.

JavaScript › Format: Semicolons

Defines handling of optional semicolons. Requires using TypeScript 3.7 or newer in the workspace.

insert 



Licenses

Original Airbnb CSS Style Guide

MIT License

Copyright © 2015 Airbnb

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

MicroCODE CSS Style Guide

MIT License

Copyright © 2022 MicroCODE, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

