Specification: 0001 Robin Bron
Ratus B.V.
August 2018

JavaScript Styling

This document describes a set of rules for JavaScript source code. To apply these rules, include the next paragraph near the beginning of the documentation of the project or near the beginning of the generic README file of the project:

> The JavaScript source code in this project must adhere to the rules as described in RATUS/SPEC0001

Table of contents

1. Introduction

This document serves as the complete definition of the coding standards for source code in the JavaScript programming language as followed by Ratus. A JavaScript source file is described as being in "Ratus Style" if, and only if, it adheres to the rules herein.

Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the hard-and-fast rules that we follow universally, and avoids giving advice that isn't clearly enforceable (whether by human or tool).

1.1. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119 when, and only when, they appear in all capitals, as shown here.

2. Source file basics

2.1. File name

File names MUST be all lowercase and may include underscores (_) or dashes (-), but no additional punctuation. The extension MUST always be ".js".

2.2. File encoding

Source files MUST always be encoded according to the UTF-8 standard (See RFC3629).

2.3. Special characters

2.3.1 Whitespace characters

Aside from the line-feed character, the ASCII (See RFC20) horizonal space character (0x20) is the only whitespace character that appears anywhere in a source files.

2.3.2. Special escape sequences

For any character that has a special escape sequence, that sequence SHOULD be used rather than the corresponding numeric escape sequence. Legacy octal escapes MUST NOT be used.

2.3.3. Non-ASCII characters

For the remaining non-ASCII characters, either the actual Unicode character or the equivalent hex or Unicode escape is used, depending only on which makes the code easier to read and understand.

3. Formatting

3.1. Braces

3.1.1. Control structures

Braces are REQUIRED for all control structures (i.e. if, else, for, do, while, as wel as any others). The first statement of a non-empty block MUST begin on its own line.

Control structures SHOULD omit braces and be written on a single line if the both the statement and the control structure can be kept on a single line without wrapping when it improves readability.

3.1.2. Non-empty blocks

Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for non-empty blocks and block-like structures.

- 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 if that brace terminates a statement or the body of a function or class statement, or a class method. Specifically, there is no line break after the brace if it is followed by "else", "catch", "while", or a comma, semicolon, or right-prarenthesis.

3.1.3. Empty blocks

An empty block or block-like construct SHOULD be closed immediately after it is opened, with no characters, space, or line break in between, unless it is part of a multi-block statement.

3.2. Indentation

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block.

3.3. String literals

Ordinary string literals SHOULD be delimited with single quotes (’) and MUST NOT span multiple lines.

To prevent complex string concatenation, template strings (delimited with ‘) SHOULD be used. Template strings MAY span multiple lines in which case they SHOULD adhere the indent level of the enclosing block if the whitespace does not affect functionality or complicates the code.

3.4. Number literals

Numbers may be specified in decimal, hexidecimal, octal or binary. Use exactly "0x", "0o" and "0b" prefixes, with lowercase characters, for hex, octal and binary respectively. Never include a leading zero unless it is immediately followed by "x", "o" or "b".

3.5. Array literals

Array literals SHOULD include a trailing comma whenever there is a line break between the final element and the closing bracket.

The variadic Array constructor MUST NOT be used for creating a new array, unless used for allocating an empty array of a given length.

Non-numeric properties on an array other than "length" or a Symbol MUST NOT be used. Use a Map or Object instead.

Array literals MAY be used on the left-hand side of an assignment to perform destructuring (such as when unpacking multiple values from a single array or iterable). A final "rest" element MAY be included (with no space between the "..." and the variable name).

Destructuring MAY also be used for function parameters (note that a parameter name is required but ignored). Always specify "[]" as the default value if a destructured array parameter is optional, and provide default values on the left hand side.

Array literals MAY include the spread operator (...) to flatten elements out of one or more other iterables. The spread operator SHOULD be used instead of more awkward constructs with "Array.prototype". There is no space after the "...".

3.6. Object literals

A trailing comma SHOULD be used whenever there is a line break between the final property and the closing brace.

While the Object constructor does not have the same problems as the Array constructor, the Object constructor MUST NOT be used to create a new object. Use an object literal instead.

When writing an object literal, unquoted keys and quoted keys MUST NOT be used.

Computed property names are allowed and are considered quoted keys (they MUST NOT be mixed with non-quoted keys) unless the computed property is a symbol. Enum values may also be used for computed keys, but should not be mixed with non-enum keys in the same literal.

Methods SHOULD be defined on object literals using the method shorthand in place of a colon immediately followed by a function or arrow function literal to be consistent with class literals.

3.7. Functions

3.7.1. Function literals

Exported top-level functions MAY be defined directly on the exports object or else declared locally and exported separately. Non-exported functions are encouraged and should not be declared private. Functions MAY contain nested function definitions. If it is useful to give the function a name, it should be assigned to a local const.

3.7.2. Arrow function literals

Arrow function literals SHOULD be used instead of "function" literals whenever applicable, unless the code is easier to read and understand when not.

The right-hand side of the arrow MUST be either a single expression or a block. Multiple expressions MAY NOT be concatenated into a single expression using commas when used as the only statement of an arrow function.

3.7.3. Generator functions

Generators enable a number of useful abstractions and MAY be used as needed. When defining generator functions, attach the "*" to the "function" keyword when present and separate it with a space from the name of the function. When using delegating yields, attach the "*" to the "yield" keyword.

3.7.4. Parameters

3.7.4.1. Default parameters

Function parameters MUST be typed with JSDoc annotations in the JSDoc preceding the function’s definition,

Parameter types MAY be specified inline, immediately before the parameter name. Inline and "@param" type annotations MUST NOT be mixed in the same function definition.

Optional parameters SHOULD be indicated by using the equals operator to set a default value for that parameter, even if the default value should be undefined. Optional parameters indicated by a default value MUST include spaces on both sides of the equals operator, be named exactly like required parameters (i.e. not prefixed), use the "=" suffix in their JSDoc type and not use initializers that produce observable side effects. Optional parameters SHOULD come after required parameters.

Use default parameter values sparingly. Prefer destructuring to create readable APIs when there are more than a small handful of optional parameters that do not have a natural order.

3.7.4.2. Rest parameters

Use a rest parameter instead of accessing the special arguments variable. Rest parameters are typed with a "..." prefix in their JSDoc. The rest parameter MUST be the last parameter in the list. There is no space between the "..." and the parameter name. The rest parameter MUST NOT be named "arguments" or any other word which confusingly shadows built-in names.

3.7.5. Returns

Function return types MUST be specified in the JSDoc directly above the function definition.

3.7.6. Spread operator

Function calls MAY use the spread operator. The spread operator SHOULD be used in preference over Function.prototype.apply when an array or iterable is unpacked into multiple parameters of a variadic function. There MUST NOT be a space between the spread operator and the array or iterable.

3.8. Classes

3.8.1. Constructors

Constructors are OPTIONAL for concrete classes. Subclass constructors MUST call "super()" before setting any fields or otherwise accessing "this", unless required to do so in order to acquite their goal.

3.8.2. Fields

All of a concrete object’s fields (i.e. all properties other than methods) MUST be set from within the constructor. Fields that are never reassigned SHOULD be annotated with "@const".

Private fields SHOULD either be annotated with "@private" or have a Symbol as key. Fields MUST NOT be set on a concrete class’ prototype.

3.8.3. ES5 class declarations

While ES6 classes are preferred, there are cases where ES6 classes may not be feasible.

Per-instance properties SHOULD be defined in the constructor after the call to the super class constructor, if a super class exists. Methods SHOULD be defined on the prototype of the constructor.

3.8.4. Prototype manipulation

In ES6 class definitions, the prototype of the class SHOULD NOT be manipulated directly. Ordinary implementation code has no business manipulating these objects.

Mixins and modifications of the prototypes of builtin objects SHALL NOT be used, unless part of framework code which otherwise would resort to even-worse workarounds to avoid doing so.

3.8.5. Getters and setters

The JavaScript getter and setter properties MUST NOT be used, unless part of data-binding frameworks where they MAY be used sparingly.

3.9. This

Only use the this builtin in class constructors and methods, or in arrow functions defined within class constructors and methods. Any other uses of this MUST have an explicit "@this" declared in the immediately-enclosing function’s JSDoc.

The this builtin SHOULD NOT be used to refer to the global object, the context of an eval or the target of an event.

3.10. Disallowed features

The "with" keyword
: The "with" keyword MUST NOT be used. It makes your code harder to understand and has been banned in strict mode since ES5.

Dynamic code evaluation
: The "eval" method and the "Function(...string)" constructor MUST NOT be used outside of code loaders. These features are potentially dangerous and simply do not work in CSP environments.

Automatic semicolon insertion
: Always terminate statements with semicolons, except for function and class declarations.

Non-standard features
: Non-standard features MUST NOT be used. This includes old features that have been removed, new features that are not yet standardized or proprietary that are only implemented in some JavaScript environments. These features are only allowed if the code being written is intended for only that environment.

Wrapper objects for primitive types
: Never use the "new" keyword on primitive object wrappers nor include them in type annotations. The wrappers MAY be called as functions for coercing (which is preferred over using "+" or concatenating the empty string) or creating Symbols.

4. Naming

4.1. Rules for all identifiers

Identifiers MUST use only ASCII letters, digits, underscores and the dollar sign.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project and do not abbreviate by deleting letters within a word.

4.2. Rules by identifier type

```
Package names ..................................... kebab-case
Class names ..................................... UpperCamelCase
Method names .................................... lowerCamelCase
Enum names ...................................... UpperCamelCase
Constant names ............................ SCREAMING_SNAKE_CASE
Non-constant field names ........................ lowerCamelCase
Parameter names ................................. lowerCamelCase
Local variable names ............................ lowerCamelCase
Template parameter names ................... SCREAMING_SNAKE_CASE
```

5. JSDoc

5.1. General form

JSDoc is a generic docblock (/**) with a body as defined here. JSDoc is either multi-line or single-line, where the single-line version MUST follow the parameter or field section of the multi-line version.

There are many tools which extract metadata from JSDoc comments to perform code validation and optimization. As such, these comments MUST be well-formed.

A JSDoc comment can contain the following sections, which are described in 5.2. through 5.4.:

- Summary
- Description
- Tags

5.2. Summary

The summary is a one-line string used to give an impression of the function of the documented element. This can be used in overviews to allow the user to skim the documentation in search of the required template.

5.3. Description

The description contains concise information about the function of the documented element. The description MUST be in Markdown markup to apply styling.

The following list has examples of types of information that can be contained in a description:

- Explanation of algorithms
- Code examples
- Array specification
- Relation to other elements
- License information (in the case of file documentation)

Descriptions can also contain inline tags. These are special annotations that can be substituted for a specialized type of information (such as {@link}). Inline tags MUST always be surrounded by braces.

5.4. Tags

Tags represent metadata with which IDEs, external tooling or even the application itself know how to interpret an element.

5.4.1. JSDoc tag reference

The following tags are common and well supported by various documentation generation tools (such as JsDossier) for purely documentation purposes.

Tag	Description
@author @owner	Document the author of a file or the owner of a test, generally only used in the @fileoverview comment. Not recommended.
@bug	Indicates what bugs the given test function regression tests. Multiple bugs should each have their own @bug line, to make searching for regression tests as easy as possible.
@see	Reference a lookup to another class function or method, global function, file or URL.
@param	Indicates the type of a function or method parameter, optionally adding a description to further explain what the described parameter does.
@return	Indicates the return type of a function or method, optionally adding a description to further explain what the return value contains.
@type	Indicate the documented element’s type.
@const	Describes the documented element is a constant variable, that it MUST NOT be reassigned later.
@private	Describes the documented element is private and care MUST be taken to not expose the element to scopes other than the one it is declared in.
@this	Indicates the documented element uses the "this" keyword and SHOULD be handled with care in relation to it’s context.
@override	Indicates the documented method overrides the equally-named super class method.
@deprecated	Indicates the documented element is deprecated and this SHOULD not be used in new code.

5.5. Line wrapping

Line-wrapped block texts MUST be indented four spaces or be aligned with the start of the text when it’s a comment on a tag.

Wrapped description text SHOULD be lined up with the description on previous lines.

5.6. Top/file-level comments

A file MAY have a top-level overview. A copyright notice and author information are optional. File overviews are recommended whenever a file consists of more than a single class definition. The top level comment is designed to orient readers unfamiliar with the code to what is in this file. If present, it MAY provide a description of the file’s contents and any dependencies or compatibility information. Line wrapping MUST follow the rules defined in section 5.5.

5.7. Class comments

Classes, interfaces and records MUST be documented with a description and any template parameters, implemented interfaces and other appropriate tags. The class description SHOULD provide the reader with enough information to know how and when to use the class, as wel as any additional considerations necessary to correctly use the class. Textual descriptions MAY be omitted on the constructor.

5.8. Enum and typedef comments

Enums and typedefs MUST be documented. Public enums and typedefs MUST have a non-empty description. Individual enum items may be documented with a JSDoc comment on the preceding line.

5.9. Method and function comments

Parameter and return types MUST be documented. The "this" type should be documented when necessary. Method, parameter and return descriptions (but not types) MAY be omitted if they are obvious from the rest of the method’s JSDoc or from it’s signature. Method descriptions SHOULD start with a sentence written in the third person declarative voice (a.k.a. the summary).

If a method overrides a superclass method, it must include an @override annotation. Overridden methods must include all @param and @return annotations if any types are refined, but SHOULD emit them if the types are all the same.

Anonymous functions do not require JSDoc, though parameters may be specified inline if the automatic type inference is insufficient.

5.10. Property comments

Property types must be documented. The description may be omitted for private properties, if name and type provide enough documentation for understanding the code.

Publicly exported constants are commented the same way as properties. Explicit types may be omitted for @const properties initialized from an expression with an obviously known type.

5.11. Nullability

When defining the type of a parameter or other element, nullability SHOULD be indicated by either "!" or "?" as a prefix of the type for non-null and nullable, respectively. Primitive types are nullable by default but cannot be immediately distinguished from a name that is typed to a non-null-by-default type. As such, all types except primitives and record literals SHOULD be annotated explicitly with either "?" or "!" to indicate whether they are nullable or not.

5.12. Template parameter types

Whenever possible, one SHOULD specify template parameters when dealing with elements which by default contain other elements, such as Objects, Arrays or a Promise.

Objects MUST NOT specify template parameters when used as a hierarchy instead of a map-like structure.

6. Policies

6.1. Unspecified styling

For any style question that isn’t settled definitively by this specification, one SHOULD follow the code style of the rest of the file. If that doesn’t resolve the question, consider emulating the other files in the same package. If that still does not resolve the question, follow the rules set by standardjs.

As a rule of thumb: be consistent throughout the package.

6.2. Deprecation

Mark deprecated methods, classes, interfaces or functions with @deprecated annotations. A deprecation comment MUST include simple, clear directions for people to fix their call sites.

6.3. Code not in Ratus Style

You will occasionally encounter files in your codebase that are not in proper Ratus Style. These may have come from an acquisition, or may have been written before Ratus Style took a position on some issue, or may be in non-Ratus Style for any other reason.

6.3.1. Reformatting existing code

When working on the file, only reformat the functions and/or methods you change instead of the whole file. If significant changes are being made to a file, it is expected that the file will be in Ratus Style.

6.3.2. Newly added code

Brand new files MUST use Ratus style, regardless of style choices of other files in the same package. When adding new code to a file that is not in Ratus Style, reformatting the existing code first is recommended, subject to the advice in section 8.3.1.

If this reformatting is not done, the new code should be as consistent as possible with existing code in the same file, but MUST not break any rules of this specification.

6.4. Local style rules

Teams and projects may adopt additional style rules beyond those in this document, but must accept that cleanup changes may not abide by these additional rules, and must not block such cleanup changes due to violating any additional rules. Beware of excessive rules which serve no purpose. The style guide does not seek to define style in every possible scenario and neither should you.

6.5. Generated code

Source code generate by any build process is not required to be in Ratus Style. However, any generated identifiers that will by referenced from hand-written code must follow the naming requirements. As a special exception, such identifiers are allowed to contain underscores, which may help to avoid conflicts with hand-written identifiers.

6.6. Third-party code

This style specification does not apply to third-party code used within the package. When working on third-party code embedded in the package, section 6.3 applies.

When working on third-party code which is not embedded in the package, you MUST follow the style guide supplied by that project if available.

7. Informative resources

[JSGUIDE] Google JavaScript Style Guide
https://google.github.io/styleguide/jsguide

[STANDARDJS] StandardJS standard style
https://standardjs.com/rules

[kebab-case] Special case styles
https://en.wikipedia.org/wiki/Kebab_case

[camel-case] Camel case
https://en.wikipedia.org/wiki/Camel_case

[SCREAMING_SNAKE_CASE] Snake case
https://en.wikipedia.org/wiki/Snake_case

[RFC20] ASCII format for Network Interchange
Vint Cerf
https://tools.ietf.org/html/rfc20

[RFC2119] RFC Key Words
S. Bradner
https://tools.ietf.org/html/rfc2119

[RFC3629] UTF-8
F. Yergeau
https://tools.ietf.org/html/rfc3629

8. Author information

Name ..... Robin Bron
Nickname ... Finwo
EMail ..... robin@finwo.nl