# JS code standard

Original source: SPA book, hi_score project
External definitions are marked like so ↗

## 1 Purpose

This code standard is designed to help organizations:

1.1.1 Use the strengths of JavaScript.
1.1.2 Facilitate team communication.
1.1.3 Encourage code efficiency, effectiveness, and reuse.
1.1.4 Avoid common mistakes and technical debt.
1.1.5 Deliver a higher quality product sooner.

## 2 Background

Computer languages have two audiences: the machines that execute the instructions and the humans who write, maintain, or extend it. Code that is written for others to understand tends to be carefully constructed, well documented, and becomes an asset. Conversely, code that cannot be understood after it written is technical debt[1] that becomes a liability.

A well-defined standard for a loosely typed, dynamic language like JavaScript is arguably more valuable than with stricter languages. JavaScript's very flexibility can open a Pandora's Box of syntax and practice. Stricter languages have a good deal of structure and consistency built-in. Developers must employ convention and discipline to achieve the same with JavaScript.

This standard has been developed over many years and contains numerous discussions and examples. A condensed three-page quick-reference guide may be found in Appendix B.

1 Technical debt is the deferred cost of undocumented process and systems.

# 3 Name variables to mean something

Writers use a variable name convention so they won't have to insert a clumsy time-and-focus-sapping description every time a variable is presented. Remove this cognitive overhead frees the audience to focus on core concepts and valuable new features as shown in Listing A.1.

**Listing A.1 – Variables with meaning**

```
// == Avoid ===
var x = 'Fred';
function p () { console.log( 'Hello ' + x ); }
p();

// == Prefer ==
var person_name = 'Fred';
function sayHelloFn () { console.log( 'Hello ' + person_name ); }
sayHelloFn();
```

## 3.1 Abbreviate smartly

3.1.1 Do not abbreviate short words.
3.1.2 Remove most articles, adjectives, and prepositions from names.
3.1.3 Use standard abbreviations and acronyms where they exist.
3.1.4 Prefer truncated names to contractions.

**Listing A.2 – Abbreviations**

```
// == Avoid ===
// var dgClrStr = 'brown';
// function walkWithTheBrownDog () {…}
// for ( q = 0; q < 9; q++ ) …
// var denormalizationMap = {…};
// var dnmlztnMap = {…};

// == Prefer ==
// var dogColorStr = 'brown';  // Do not abbreviate short words
// function walkDog () {…}     // Discard articles and prepositions
// for ( i = 0; i < 9; i++ ) … // Use standard i, j, k for index
// var denormMap = {…};        // Truncate instead of contract
```

## 3.2 Replace comments with meaningful names

Name a variable to describe why it is needed and what type it is expected to contain. The first consideration is important in any language; the second is especially important for loosely-type languages like JavaScript. Listing A.3 shows variables named by purpose and type.

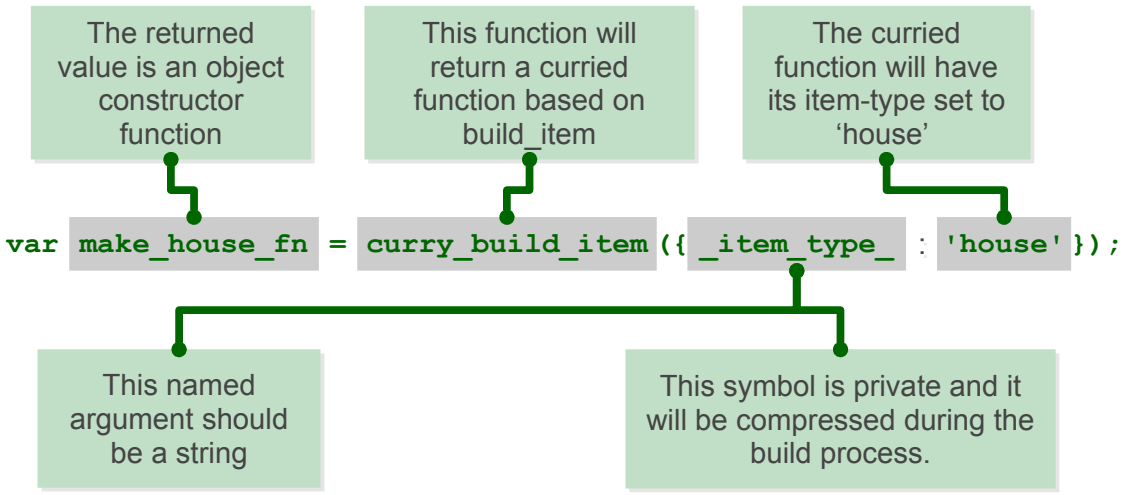
**Listing A.3 – Names with purpose**

```
// == Avoid ===
// 'creators' is an object constructor we get by
// calling 'makers'. The first positional argument
// of 'makers' must be a string, and it directs
// the type of object constructor to be returned.
// 'makers' uses a closure to remember the type
// of object the returned function is to
// meant to create.
//
var creators = makers( 'house' );

// == Prefer ==
var make_house_fn = curry_build_item({ _item_type_ : 'house' });
```

This approach discards comments and relies on convention to precisely identify the type and purpose of the variables as shown in Diagram A.1.

**Diagram A.1 – Variable name dissection**



Prefer the use of convention over copious comments to describe the meaning of statements.  Consider, for example, what can happen If a teammate updates a few names to for some reason. It is all to easy improperly update the comments making them misleading as shown in Listing A.4.

**Listing A.4 – Good intentions and bad comments**

```
// == Avoid ===
// 'creators' is an object constructor we get by
// calling 'makers'. The first positional argument
// of 'makers' must be a string, and it directs
// the type of object constructor to be returned.
// 'makers' uses a closure to remember the type
// of object the returned function is to
// meant to create.
//
var makers = builders( 'house' );

// == Prefer ==
var make_abode = curry_make_item({ _item_type_ : 'abode' });
```

Misleading comments can be worse than no comments at all. In contrast, the changes to the preferred code in are far shorter and guaranteed correct.

## 3.3  Use common characters

3.3.1 Use the characters `a-z`, `A-Z`, `0-9`, underscore, and `$`, for variable names.
3.3.2 Don't begin a name with a number.

**Listing A.5 – Keyboard characters**

```
// == Avoid ===
my_obj[ '00-x®' ] = 'hello';

// == Prefer ==
my_obj._greet_str_ = 'hello';
```

Limit variable names to characters available on most of the world's keyboards. Apply the same character limits to object property names since all variables are object properties of their functional scope.

## 3.4 Communicate variable scope

3.4.1 Place each module in its own file

3.4.2 Use `camelCase` when the variable is module scope.

3.4.3 Use `snake_case` when the variable is local to a function within a module.

3.4.4 Use two or more syllables for module-scope variables.

**Listing A.6 – Variable scope names**

```
// == Avoid ===
var stateMap = {};  // Module-scope
function initModule () {
  var
    localInt = 1,  // Local-scope
    localStr = 'Module initialized. Our number is ';
  console.log( localStr + localInt );
}
return { _initModule_ : initModule };

// == Prefer ==
var stateMap = {};  // Module-scope
function initModule () {
  // Local-scope
  var
    local_int = 1, // Local-scope
    local_str = 'Module initialized. Our number is ';
  console.log( local_str + local_int );
}
return { _initModule_ : initModule };
```

## 3.5 Communicate variable type

Add a suffix or prefix a variable name to indicate its intended type. Avoid changing a variable type after declaration because it causes confusion and rarely provides any benefit. When there is an exception, use an unknown type indicator.

**Listing A.7 – Type indicators**

```
// == Avoid ===
var x = 10, y = '02', z = x + y;
console.log ( z ); // '1002'

// == Prefer ===
var x_num = 10, y_str = '02', z_num = x_num + Number( y_str );
console.log ( z_num ); // 12
```

### 3.5.1 Booleans

Name boolean variables using **`noun-type`** or **`type-noun`**. Recommended **`type`** indicators are shown in Table A.1. Type indicators for booleans are often prefixes because they read better in English. Most other **`type`** indicators are suffixes.

**Table A.1 – Boolean indicators**

| Indicator | Local scope | Module scope |
|---|---|---|
| _bool [generic] | return_bool | returnBool |
| do_ (requests action) | do_retract | doRetract |
| has_ (inclusion) | has_whiskers | hasWhiskers |
| is_ (state) | is_retracted | IsRetracted |

### 3.5.2 Functions

Name functions and function variables using **`verb-noun-type`**. Recommended **`type`** indicators are shown in Table A.2. Recommended verbs for are shown in tables A.3-5.

**Table A.2 – Function indicators**

| Indicator | Local scope | Module scope |
|---|---|---|
| _fn [generic] | bound_fn<br>curry_get_list_fn<br>get_car_list_fn<br>fetch_car_list_fn<br>remove_car_list_fn<br>store_car_list_fn<br>send_car_list_fn | boundFn<br>curryGetListFn<br>getCarListFn<br>fetchCarListFn<br>removeCarListFn<br>storeCarListFn<br>sendCarListFn |
| <verb><noun><type> | (not recommended) | curryGetList<br>getCarList |

| Table A.3 – Function verbs for local data | | |
|---|---|---|
| **Verb** | **Example** | **Meaning** |
| fn | syncFn | Generic function indicator |
| bound | boundFn | A curried function that has a context bound to it. |
| curry | curryMakeUser | Return a function as specified by argument(s) |
| delete | deleteUserObj | Remove data structure from memory |
| destroy, remove | destroyUserObj | Same as delete, but implies references will be cleaned up as well |
| empty | emptyUserList | Remove all members of a data structure without removing the container |
| get | getUserObj | Get data structure from memory |
| make | makeUserObj | Create a new data structure using input parameters |
| store | storeUserList | Store data structure in memory |
| update | updateUserList | Change memory data structure in-place |

| Table A.4 – Function verbs for remote data | | |
|---|---|---|
| **Verb** | **Example** | **Meaning** |
| fetch | fetchUserList | Fetch data from external source like AJAX, local storage, or cookie |
| put | putUserChange | Send data to external source for update |
| send | sendUserList | Send data to external source |

| Table A.5 – Function verb for event handler | | |
|---|---|---|
| **Verb** | **Example** | **Meaning** |
| on | onMouseover onClickHeader | An event handler. Use <on><event-name><modifier> |

### 3.5.3 Integers

Name integer variables using `noun-type`. Recommended `type` indicators are shown in Table A.6.

| Table A.6– Integer indicators | | |
|---|---|---|
| **Indicator** | **Local scope** | **Module scope** |
| _int [generic] | size_int | sizeInt |
| _count | user_count | userCount |
| _idx | user_idx | userIdx |
| _ms (milliseconds) | click_delay_ms | clickDelayMs |
| i, j, k (convention) | i | - |
| _toid, _intid | show_popup_toid | showPopUpToid |

JavaScript requires an integer value for a number of purposes such as an index for an array, or as an argument for `indexOf`, or `subStr`. Consider, for example, what happens if we try to use a float for an array index as shown in Listing A.8.

**Listing A.8 – Array with a non-integer index**

```
// == Avoid ===
var color_list = [ 'red', 'green', 'blue' ];
color_list[1.5] = 'chartreuse';
console.log( color_list.pop() ); // 'blue'
console.log( color_list.pop() ); // 'green'
console.log( color_list.pop() ); // 'red'
console.log( color_list.pop() ); // undefined - where is
'chartreuse'?
console.log( color_list[1.5]  ); // oh, there it is
console.log( color_list       ); // '[1.5: "chartreuse"]'
```

## 3.5.4 Lists

Name array variables using `noun-type`. Recommended `type` indicators are shown in Table A.7. Please use only singular nouns as the suffixes indicate plurality.

| Table A.7 – List indicators | | |
|---|---|---|
| **Indicator** | **Local scope** | **Module scope** |
| _list [generic] | timestamp_list<br>color_list | timestampList<br>colorList |
| _table [list of pointers like list-of-lists ] | user_table | userTable |

### 3.5.5 Numbers

Name floating-point numbers using **`noun-type.`** Recommended **`type`** indicators are shown in Table A.8. Please use only singular nouns as the suffixes indicate plurality.

**Table A.8 – Number indicators**

| Indicator | Local scope | Module scope |
|---|---|---|
| `_num [generic]` | `size_num` | `SizeNum` |
| `_coord` | `x_coord` | `xCoord` |
| `_px (fractional unit)` | `x_px, y_px` | `xPx` |
| `_ratio` | `sale_ratio` | `saleRatio` |
| `x,y,z` | `x` | `-` |

### 3.5.6 Maps

Name maps using **`noun-type.`** Recommended **`type`** indicators are shown in Table A.9. Please use only singular nouns as the suffixes indicate plurality.

**Table A.9 – Map indicators**

| Indicator | Local scope | Module scope |
|---|---|---|
| `_map [generic]`<br>`_matrix [map of pointers like map-of-objects]` | `employee_map`<br>`receipt_map`<br>`user_matrix` | `employeeMap`<br>`receiptMap`<br>`userMatrix` |

Maps are simple objects used to store collections of data. Similar structures include a **`map`** in Java, a **`dict`** in Python, an **`associative array`** in PHP, or a **`hash`** in Perl. When we name a map, we want to emphasize the data intent. Use the **`_table`** or **`_matrix`** suffix to indicate more complex data structures such as a a list-of-lists or a map-of-objects.

### 3.5.7 Objects

Name full-featured objects using **`noun-type.`** Recommended **`type`** indicators are shown in Table A.10.

| Table A.10 – Object indicators | | |
|---|---|---|
| **Indicator** | **Local scope** | **Module scope** |
| _obj [generic] | employee_obj<br>receipt_obj<br>error_obj | employeeObj<br>receiptObj<br>errorObj |
| $ (jQuery object) | $header<br>$area_tabs | $Header<br>$areaTabs |
| _proto (prototype) | user_proto | userProto |

### 3.5.8 Regular expression objects

Name regular expression objects using **`noun-type`**. The recommended **`type`** indicator is shown in Table A.11.

| Table A.11 – Regex indicator | | |
|---|---|---|
| **Indicator** | **Local scope** | **Module scope** |
| _rx | match_rx | matchRx |

### 3.5.9 Strings

Name strings using **`noun-type`**. Recommended **`type`** indicators are shown in Table A.12.

| Table A.12 – String indicators | | |
|---|---|---|
| **Indicator** | **Local scope** | **Module scope** |
| _str [generic] | Direction_str | directionStr |
| _date | email_date | emailDate |
| _html | body_html | bodyHtml |
| _id, _code (identifier) | email_id | emailId |
| _msg (message) | employee_msg | employeeMsg |
| _name, _filename, _dirname | employee_name | employeeName |
| _txt | email_text | emailText |
| _type | item_type | itemType |

### 3.5.10 Unknown types

Name variable of an unknown type using `noun-type`. The recommended `type` indicator is shown in Table A.13.

**Table A.13 – Unknown type indicator**

| Indicator | Local scope | Module scope |
|---|---|---|
| _data | http_data<br>socket_data<br>arg_data<br>data | httpData,<br>socketData |

Variables with unknown types are encountered in polymorphic functions where an argument may have one of many types. One such function might concatenate strings, numbers, arrays, or maps. We also encounter unknown data types when receiving data from an external source such as an AJAX or web socket API.

## 3.6 Avoid plurals

Avoid plurals in any variable name. A plural noun implies an indeterminate group of data. Instead use a variable name that more precisely identifies the type of group data a variable contains.

**Listing A.9 – Collections of data**

```
// == Avoid ===
var
  cats     = [ 'callico', 'tabby' ],
  colors   = { blue : '#00f', green : '#0f0', red : '#00f' },
  persons  = [ { name : 'Fred' }, [ name : 'Wilma' } ],
  retracts = true,
  users    = 'Betty,Bamm-Bamm,Fred,Pebbles,Wilma'

// == Prefer ==
var
  cat_list      = [ 'callico', 'tabby' ],
  color_map     = { blue : '#00f', green : '#0f0', red : '#00f' },
  do_retract    = true,
  person_table  = [ { name : 'fred' }, [ name : 'wilma' } ],
  user_csv_list = 'Betty,Bamm-Bamm,Fred,Pebbles,Wilma'
  ;
```

## 3.7 Sort alphabetically

Sort list and map literals and variable declarations in alphabetical order unless there is a precedence requirement or other obvious reason for a different order. Sort lists, maps, and strings *within* declarations as well. Use an editor like Vim, Sublime, or WebStorm which support in-line sorting.

**Listing A.10 – Sorted declarations**

```
// == Avoid ===
var
  do_retract    = true,
  color_map     = { green : '#0f0', red : '#00f', blue : '#00f' },
  person_table  = [ { name : 'Wilma' }, { name : 'Fred' } ],
  user_csv_list = 'Pebbles,Wilma,Betty,Bamm-Bamm,Fred',
  cat_list      = [ 'tabby', 'callico' ]
  ;

// == Prefer ==
var
  cat_list      = [ 'callico', 'tabby', ],
  color_map     = { blue : '#00f', green : '#0f0', red : '#00f' },
  do_retract    = true,
  person_table  = [ { name : 'fred' }, [ name : 'wilma' } ],
  user_csv_list = 'Betty,Bamm-Bamm,Fred,Pebbles,Wilma';
  ;
```

## 3.8 Use similar conventions for properties

Object keys (or properties as they are sometimes called) should be named using the same convention as other variables. The only difference is that the keys should be wrapped in underscores so they may be discovered and compressed. Examples are shown below.

**Table A.14 – Example property names**

| Type | Local scope | Module scope |
|---|---|---|
| Array | local_map._person_list_ | spa._model_._personList_ |
| Boolean | local_map._is_enabled_ | spa._model_._isReady_ |
| Function | local_map._init_fn_ | spa._initModule_<br>spa._model_.initModule_<br>spa._shell_._initModule_ |

| | | |
|---|---|---|
| Integer | local_map._leg_count_ | spa._model_._callbackCount_ |
| Map | local_map._user_map_ | spa._slider_._instanceMap_ |
| Number | local_map._mix_ratio_ | spa._sound_._mixRatio_ |
| String | local_map._username_ | spa._model_._userMap_ |
| Regex | local_map._match_rx_ | spa._matchRx_ |

# 4 Variable declaration and assignment

Variables can contain array, functions, numbers, object, strings, **`null`**, or **`undefined`**. Some JavaScript implementations may make internal distinctions between integers, 32-bit signed, and 64-bit double-precision floating point numbers but there is no formal means to enforce this.

## 4.1 Use namespaces

Create a single global namespace map inside of which all our other variables are scoped. In the browser environment use an **IIFE** (Immediately Invoked Function Expression) as shown in Listing A.11. When declaring an **IIFE** always wrap the function in parenthesis so that it's clear that the value being produced is the result of the function and not the function itself. Always use the **`'use strict'`** pragma for module-scope **IIFEs**.

**Listing A.11 – A namespace map created using an IIFE**

```
// == Avoid ==
var greetStr = 'Hi there!';
console.log( window.greetStr ); // prints 'Hi there!'

// == Prefer ==
var spa = (function () {
  'use strict';
  var greetStr = 'Hi there!';
  function initModule () { console.log( greetStr ); }
  return { _initModule_ : initModule };
}());
spa._initModule_();
```

We can break a namespace into manageable subdivisions. For example, we can add **`spa._shell_`** and **`spa._slider_`** to our **`spa`** namespace as shown in Listing A.12.

**Listing A.12 – A namespace subdivided**

```
// == Prefer ==
// In the file spa.shell.js:
var spa._shell_ = (function () {
  'use strict';
  // … private variables and methods …
  return {
    _initModule_   : initModule,
    _resetDisplay_ : resetDisplay
  };
}());

// In the file spa.slider.js:
var spa._slider_ = (function () {
  'use strict';
  // … private variables and methods …
  return {
    _initModule_    : initModule,
    _extendSlider_  : extendSlider,
    _retractSlider_ : retractSlider
  };
}());

// Initialize the spa from another module
spa._initModule_();
```

Namespace CSS in parallel with the JavaScript. For example, any classes used by **`spa._shell_`** should have an **`spa-_shell_`** prefix.

## 4.2 Avoid module scope variables

Avoid too many module-scope variables. A good strategy is to place configuration parameters in a module-scoped **`configMap`** and dynamic state information **`stateMap`** as shown in Listing A.13.

**Listing A.13 – Module scope variables**

```
// == Avoid ===
var
  isSliderActive = true,
  isSliderOpen   = false
  ;
```

```
// == Prefer ==
stateMap = {
  _is_slider_active_ : true,
  _is_slider_open_   : false
};
```

## 4.3 Avoid the new keyword

Use `{}` or `[]` instead of `new Object()` or `new Array()` to create an object, map, or array. If you require object inheritance, use `Object.create()` and use the factory pattern for object constructors shown in Listing A.14.

**Listing A.14 – Factory object constructor**

```
// == Avoid ===
var dog = new Dog();

// == Prefer ==
var dog_obj = makeDogObjFn();
```

## 4.4 Copy carefully

Complex variables in JavaScript, such as arrays and objects, are not copied when they're assigned;instead the pointer to the data structure is copied. We highly recommend the use of well-tested utilities like those provide by jQuery to copy complex data structures.

## 4.5 Use a single var statement per function

Explicitly declare all variables first in the function scope using a single `var` keyword. JavaScript scopes variables by function and doesn't provide universal block scope. Therefore if you declare a variable anywhere within a function, it'll be initialized with a value of undefined immediately on invocation of the function. Placing all the variable declarations first recognizes this behavior. It also makes the code easier to read and to detect undeclared variables.

**Listing A.15 – Single var statement**

```
// == Avoid ===
function shallowCopyMap( arg_map ) {
```

```
  var solve_map = {};
  var key_list = Object.keys( arg_map );
  var key_count = key_list.length;

  for ( var idx = 0; idx < key_count; idx++ ) {
    var key_name = key_list[ idx ];
    var val_data = arg_map[ key_name ];
    if ( val_data !== undefined ) {
      solve_map[ key_name ] = val_data;
    }
  }
  return solve_map;
}

// == Prefer ==
function shallowCopyMap( arg_map ) {
  var
    solve_map  = {},
    key_list   = Object.keys( arg_map ),
    key_count  = key_list.length,
    key_name, val_data, idx;

  for ( idx = 0; idx < key_count; idx++ ) {
    key_name = key_list[ idx ];
    val_data = arg_map[ key_name ];
    if ( val_data !== undefined ) {
      solve_map[ key_name ] = val_data;
    }
  }
  return solve_map;
}
```

Declaring a variable is not the same as assigning a value to it. Declarations determine which variables exists within which functional scope and are processed at compile-time. Assignments occur only at run-time. Variable hoisting occurs due to this distinction. As a convenience we may combine declaration and assignment with the **`var`** statement but this is not required.

## 4.6 Name functions directly

Functions with names generally are easier to debug than anonymous functions. For most purposes the the declarations in Listing A.16 are usually equivalent. However, we

*will* see a difference when debugging.  When we declare functions with a canonical names, legible stack-traces can be computed prior to run-time.

**Listing A.16 – Canonical function names**

```
// == Avoid ===
getMapCopy = function ( arg_map ) { ... };

// == Prefer ==
function getMapCopy( arg_map ) { ... };
```

## 4.7 Use named arguments

Use named arguments whenever requiring three or more arguments in a function, as positional arguments are easy to forget and aren't self-documenting:

**Listing A.17 – Named arguments**

```
// == Avoid ===
coor_map = refactorCoords( 22, 28, 32, 48);

// == Prefer ==
coord_map = refactorCoords({
  x1 : 22, y1 : 28, x2 : 32, y2 : 48
});
```

## 4.8 Format consistently

4.8.1 Use a document width of 80 characters.
4.8.2 Indent two spaces per code level.
4.8.3 Don't use tab characters.
4.8.4 Place white space between operators and variables.
4.8.5 Place white space after every comma.
4.8.6 Use only one statement or variable assignment per line.
4.8.7 Place a semicolon at the end of every statement.

**Listing A.18 – Consistent format**

```
// == Avoid ===
function makePctStr(arg_ratio,arg_count,arg_sigil_str){
        var
```

```
ratio=castNum(arg_ratio,0),count=castNum(arg_count,0),sigil_str=cast
Str(arg_sigil_str,'%')
        count=count<0?0:Math.floor(count);
  return // ← Semicolon insertion bug: return value is undefined
    { pct_str:(ratio*100).toFixed(count)+sigil_str }
}

// == Prefer ==
function makePctStr ( arg_ratio, arg_count, arg_sigil_str ) {
  var
    count     = castNum( arg_count,     0   ),
    ratio     = castNum( arg_ratio,     0   ),
    sigil_str = castStr( arg_sigil_str, '%' );

  count = count < 0 ? 0 : Math.floor( count );
  return {
    pct_str : ( ratio * __100 ).toFixed( count ) + sigil_str
  };
}
```

Keep the document width at or below 80 characters so that it fits within a standard terminal window and reads well on constrained displays such as those found on mobile devices[2]. Indent two spaces per level to avoid exceeding the document width when code is nested three or four levels deep. Use spaces instead of tabs because there is not a standard display width for tabs. A tab-formatted document might display well on one viewer but appear completely jumbled on another display.

Place white space around operators, variables, and commas to assist with readability. This has no effect on product performance as our JavaScript will be concatenated, minified, and compressed before it reaches our users.

Place only one statement or assignment per line although we may *declare* multiple variable names on a single line.  Explicitly terminate every statement with a semicolon to avoid semicolon insertion bugs as shown in Listing A.18.

## 4.9 Distinguish function declaration and invocation

4.9.1 When declaring a function place a single space between the name and its opening left parenthesis.

2 Line widths of 66 characters are considered optimal for comprehension. See Binghust, R. (2004) *The Elements of Typographic Style (3rd edition),* New York

4.9.2 When invoking a function do not put a space between the the name and the opening left parenthesis.

**Listing A.19 – Function declaration and invocation**

```
function processMap( arg_map ){ … }      // == Avoid ===
function processMap ( arg_map ) { … }    // == Prefer ==

result_map = processMap ( example_map ); // == Avoid ===
result_map = ( example_map );  // == Prefer ==
```

## 4.10 Delimit string literals with single quotes

Prefer single quotes over double quotes for string delimiters. This communicates that literal strings do not expand variables. This is consistent with Bash shell and many other popular languages. Quoting HTML is also much easier with a single quote as shown in Listing A.20.

**Listing A.20 – Single quotes and literal strings**

```
// == Avoid ===
link_str = "<a href=\"/wiki/fish\" title=\"fish\">fish</a>";

// == Prefer ==
link_str = '<a href="/wiki/fish" title="fish">fish</a>';
```

## 4.11 Break lines on operators

4.11.1 Prefer single lines when possible.
4.11.2 Break lines before operators or after comma separators.
4.11.3 Indent subsequent lines of the statement one level.
4.11.4 Optionally place the terminating semicolon on its own line to multi-line statements.

**Listing A.21 – Lines broken on operators**

```
// == Avoid ===
var full_address_str = first_name + ' ' + last_name +
'\n' + street1_str + '\n' + street2_str + '\n' +
city_str + ',' + state_code + ' ' + zip_code;

// == Prefer ==
var full_address_str
  = first_name  + ' ' + last_name + '\n'
  + street1_str + '\n'
  + street2_str + '\n'
  + city_str    + ',' + state_code + ' ' + zip_code
  ;
```

Place all statements and declarations within our document width on a single line. Divide into multiple lines any statement or declaration that exceeds our document width. Break before operators so they line-up on the left column.  Indent all continuation lines one level. This highlights the action taking place on the data.

## 4.12 Use Stroustrup-style bracketing

4.12.1 Place the opening parenthesis, brace, or bracket at the end of the opening line.
4.12.2 Indent the code inside the delimiters (parenthesis, brace, or bracket) one level.
4.12.3 Place the closing parenthesis, brace or bracket on its own line with the same indentation as the opening line.
4.12.4 Do not omit braces on **any** single-line statement.

**Listing A.22 – Stroustrup-style bracketing**

```
// == Avoid ===
function getSign(arg_data)
{
  var
    arg_num   = arg_data + 0,
    solve_int = 0;

  if (arg_num < 0) solve_int = -1
  else if (arg_num === 0)
  {
    solve_int = 0;
  } else {
```

```
    solve_int = 1;
  }
  return solve_int;
}

// == Prefer ==
function getSign( arg_data ) {
  var
    arg_num   = arg_data + 0,
    solve_int = 0;

  if ( arg_num < 0 ) {
    solve_int = -1;
  }
  else if ( arg_num === 0 ) {
    solve_int = 0;
  }
  else {
    solve_int = 1;
  }
  return solve_int;
}
```

Stroustrup style is a *one-true-brace* variant of K&R-style that does not cuddle else clauses. Many feel it nicely balances compactness, clarity, and safety.

## 4.13 Organize by paragraphs

4.13.1 Group code in paragraphs and place blank lines between each.
4.13.2 Vertically align like operators within paragraphs.
4.13.3 Indent comments the same amount as the code they explain.
4.13.4 Comment per paragraph.

**Listing A.23 – Comments by line**

```
// == Avoid ===
function shuffle( items ) {
  // Items should be an array.
  // Return false if argument is not an array
  if ( ! Array.isArray( items ) ) { return false; }
  // Get the length of the items array.  Size is an integer.
  var size = items.length;
  // Decrement i from the size of the list to 1
```

```
  for ( var i = size; i > 0; i-- ) {
    // x is the int element index at the end of the section.
    var x = i - 1;
    // y is a random integer index within the section.
    var y = Math.floor( Math.random() * i );
    // Get random element value. Swap could be any data type.
    var swap = items[ y ];
    // Set random element value to same as end of section
    items[ y ] = items[ x ];
    // Set end of section value to random element value
    items[ x ] = swap;
  }
  return true;
}
```

**Listing A.24 – Comments by paragraph**

```
// == Prefer ==
function shuffleList ( arg_list ) {
  var
    list  = castList( arg_list ),
    count, idj, last_idx, pick_idx, swap_data;

  if ( ! list ) { return false; }
  count = list.length;

  for ( idj = count; idj > 0; idj-- ) {
    last_idx         = idj - 1;
    pick_idx         = Math.floor( Math.random() * idj );
    swap_data        = list[ last_idx ];
    list[ last_idx ] = list[ pick_idx ];
    list[ pick_idx ] = swap_data;
  }
  return true;
}
```

Comment by paragraph and avoid hard-to-maintain line comments. Rely on the name convention to explain variable content and purpose, paragraph comments to explain broader concepts, and function comments to describe APIs. Use an editor like Vim, Sublime, or WebStorm which support vertical selection and alignment. Webstorm provides an option to auto-align map values which is a great time-saver.

## 4.14 Document APIs in-line

Document any non-trivial API in-line. Provide its purpose, examples, arguments, return values, exceptions, and methods as listing A.25. Place architecture plans in documents separate from the code.

**Listing A.25 – In-line APIs**

```
// == Avoid ===
function shuffleList( arg_list ) { … }

// == Prefer == See hi_score/js/xhi/xhi-module-template.js.
// BEGIN public method /shuffleList/
// Purpose   : Shuffle elements in a list
// Example   : shuffleList( [1,2,3,4] ) returns [ 3,1,4,2 ]
// Arguments : ( positional )
//   0. arg_list - The list to shuffle
// Returns   : boolean true on success
// Throws    : none
// Method    :
//   1. Count down from end of array with last_idx
//   2. Randomly pick element from between 0 and last_idx
//   3. Swap pick element with last_idx element
//
function shuffleList( arg_list ) { … }
// . END public method /shuffleList/
```

## 4.15 Mark future tasks with TODOs

Create `TODO` comments for tasks that can't be complete immediately using the format shown in Listing A.26. These have become standard enough that JSLint and most IDEs recognize them.

**Listing A.26 – A TODO comment**

```
// == Prefer ==
// TODO <yyyy-mm-dd> <username> <syslog-level>: <Explanation>
```

The date conveys the freshness of the comment, the username conveys responsibility, and the syslog-level (`emerg`, `alert`, `crit`, `err`, `warn`, `notice`, `info`, or `debug`) expresses urgency. We can use Bash to report all outstanding `TODOs` as shown in Listing A.27. This check is included in the **hi_score** commit hook.

**Listing A.27 – Listing TODOs**

```
grep TODO $(find ./ -type f -name '*.js' \
  | grep -v node_modules |grep -v /vendor/) |sort -u
```

## 4.16 Comment disabled code

It is wise to disable a code block and only delete it when we are certain it will no longer be useful. This prevents team members from solving the same problem multiple times. Disabled code should be identified by a **`TODO`** comment as shown in Listing A.28.

**Listing A.28 – Disabled code with an explanation**

```
// == Avoid ===
// while ( k > 0 ) { … }

// == Prefer ==
// BEGIN TODO 2017-12-29 mikem warn:
//   Disabled while testing alternative
// while ( k > 0 ) { … }
// . END TODO 2019-12-29 mikem
```

Search and resolve **`TODOs`** regularly – once a week is good – by recording them in the organization's issue tracking database. Convert each comment as each issue is entered as shown in Listing A.29

**Listing A.29 – An issue comment**

```
// == Prefer == (issue ID used to explain disabled code)
// Issue #96785: Disabled while testing alternative
// while ( k > 0 ) { … }
```

## 4.17 Put it all together

Listings A.30 and A.31 compare the readability of an object prototype before and after applying the recommended formatting.

### Listing A.30 – Carefree format

```
// == Avoid ===
doggy = {
  temperature : 36.5,
  name : 'Guido',
  greeting : 'Grrrr',
  speech : 'I am a dog',
  height : 1.0,
  legs : 4,
  ok : check,
  remove : destroy,
  greet_people : greet_people,
  say_something : say_something,
  speak_to_us : speak,
  colorify : flash,
  show : render
};
```

### Listing A.31 – Purposeful format

```
// == Prefer ===
dogProto = {
  _greet_str      : 'Grrrr',
  _height_m_num_  : 1.0,
  _leg_count_     : 4,
  _name           : 'Guido',
  _speak_str_     : 'I am a dog',
  _temp_c_num_    : 36.5,

  _check_destroy_fn_ : checkDestroyFn,
  _destroy_dog_fn_   : destroyDogFn,
  _print_greet_fn_   : printGreetFn,
  _print_name_fn_    : printNameFn,
  _print_speak_fn_   : printSpeakFn,
  _redraw_dog_fn_    : redrawDogFn,
  _show_flash_fn_    : showFlashFn
};
```

# 5 Apply consistent syntax

## 5.1 Remove ambiguity with labels

Labels may be used with `while`, `do`, `for`, or `switch` blocks. They clarify the purpose of a `continue` or `break` statement and make the code more resistant to nesting errors.

**Listing A.32 – Labeled statements**

```
var
  horse_list  = [ 'Anglo-Arabian', 'Clydsedale' ],
  horse_count = horseList.length,
  breed_name, idx, idj
  ;

_HORSE_NAME_: for ( idx = 0; idx < horse_count; idx++ ) {
  breed_name = horse_list[ idx ];
  _LEG_IDX_ : for ( idj = 0; idj < 4; idj++ ) {
    if ( Math.random() < 0.5 ) { break _LEG_IDX_;
    if ( idj === 3 && breed_name === 'Clydesdale' ) {
      break _HORSE_NAME_;
    }
  }
}
```

## 5.2 Prefer C-style ‘for’ syntax

Use the C-style form of the `for` statement. Avoid the `for-in` form as this iterates over inherited properties which are unreliable. Instead use `Object.keys()` to get a list of property names and iterate over that as shown in Listing A.33.

**Listing A.33 – C-style ‘for’ syntax**

```
// == Avoid ===
for ( key in cat_obj ) {
  if ( cat_obj.hasOwnProperty( key ) ) {
    // process key
  }
}

// == Prefer ==
```

```
key_list  = Object.keys( cat_obj );
key_count = key_list.length;
for ( idx = 0; idx < key_count; idx++ ) {
  key = obj_list[ idx ];
  // process key
}
```

## 5.3 Compare with precision

It is always better to use the precise `===` and `!==` operators. The `==` and `!=` operators do type coercion which is inconsistent and confusing. In particular, don't use `==` to compare against falsey values. Our JSLint configuration doesn't allow type coercion. If you want to test if a value is truthy or falsey as shown in Listing A.34.

**Listing A.34 – Check for 'truthiness'**

```
if ( is_drag_mode ) { runReport(); }
```

## 5.4 Avoid 'return' mistakes

The `return` value must start on the same line as the `return` keyword in order to avoid semicolon insertion. Do not place parentheses around the return value.

**Listing A.35 – Return without errors**

```
// == Avoid ===
return
  ( { _make_house_fn_ : make_house_fn } );

// == Prefer ==
return { _make_house_fn_ : make_house_fn };
```

## 5.5 Avoid 'switch' fall-through

Each group of statements except the default should end with `break`, `return`, or `throw`; fall-through should only be used with great caution and accompanying comments, and even then you should rethink the need for it. Is the terseness really worth the trade-off in legibility? Probably not.  The preferred form is shown in Listing A.36.

**Listing A.36 – Avoid fall-through**

```
// == Prefer ==
switch ( expression ) {
  case expression:
    // statements
    break;
  case expression:
    // statements
    break;
  default:
    // statements
}
```

## 5.6 Trap exceptions

Exceptions should be trapped using try-catch blocks using Stroustrup-style bracketing as shown in Listing A.37.

**Listing A.37 – Try-catch block**

```
// == Prefer ==
try {
  // statements
}
catch ( error_data ) {
  // statements
}
try {
  // statements
}
catch ( error_data ) {
  // statements
}
finally {
  // statements
}
```

## 5.7 Avoid the comma operator

Avoid the use of the comma operator as found in some for loop constructs. This doesn't apply to the comma separator, which is used in object literals, array literals, variable declarations, and parameter lists.

## 5.8 Avoid assignment expressions

Avoid using assignments as test conditions as this is needlessly confusing.

**Listing A.38 Assignment expressions**

```
// == Avoid ===
var
  random_int = Math.floor( Math.random() * 2 ),
  set_int
  ;

if ( set_int = random_int ) {
  console.warn( 'random int is 0' );
}

// == Prefer ==
var
  random_int = Math.floor( Math.random() * 2 ),
  set_int    = random_int
  ;

if ( random_int === 0 ) {
  console.warn( 'random int is 0' );
}
```

## 5.9 Avoid the ‘do’, ‘while’, ‘setInterval’ statements

Avoid the `do`, `while` and `setInterval` statements as they result can result in endless loop conditions which will ultimately crash a browser. Use the `for` and `setTimeout` statements which are self-limiting instead.

## 5.10 Avoid the ‘with’ statement

The `with` statement should be avoided. Use the `object.call()` family of methods instead to adjust the value of `this` during function invocation.

## 5.11 Avoid confusing plus and minus operators

Be careful to not follow a `+` with a `+` or a `++`. This pattern can be confusing. Insert parentheses between them to make your intention clear.

**Listing A.39 – Confusing signs**

```
// == Avoid ===
total = total_count + +arg_map._cost_int_;

// == Prefer ==
total = total_count + ( +arg_map._cost_int_);
```

This prevents the `+ +` from being misread as `++`. The same guideline applies for the minus sign.

## 5.12 Avoid using 'eval'

JavaScript will attempt to `eval` (evaluate and execute) a string variable in numerous situations. Avoid these situations for better security and performance.

5.12.1 Don't use the `Function` constructor with a string as this performs an `eval`.
5.12.2 Don't pass strings to `setTimeout` or `setInterval` as this performs an `eval`.
5.12.3 Don't use `eval` to process JSON. Use `JSON.parse()` to convert JSON string into a data structure and `JSON.stringify()` to convert a data structure into a string.

# 6 Layout JS and CSS files by namespace

6.1 Place JavaScript files used by the web application under a directory called `js`.
6.2 Name JavaScript files according to the namespace they provide, one namespace per file. All files should have the `.js` suffix as shown in Listing A.40.
6.3 Use the template to start any JavaScript module file. This is found in the **hi_score** project at `js/xhi/xhi-module-template.js`.
6.4 Place all CSS files used by the web application in a directory called `css`.
6.5 Maintain a parallel structure between JavaScript and CSS files and class names. Create a CSS file for each JavaScript file that generates HTML. All files should have the `.css` suffix as shown below.
6.6 Prefix CSS selectors according to the name of the module they support. This practice helps greatly to avoid unintended interaction with classes from third-party modules as shown in Listing A.40.
6.7 Use <namespace>_x_<descriptor>_ for state-indicator and other shared class names. Examples include `spa-_x_select_` and `spa-_x_disabled_`. Place these in the root namespace stylesheet, for example `spa.css`.

6.8 When using PowerCSS, keep the same parallel structure, replacing CSS files with JS files as shown in Listing A.40.

6.9 Include third-party JavaScript files first in our HTML so their functions may be evaluated and made ready for our application.

6.10 Include custom JavaScript files next, in order of namespace. We can't load namespace `spa.shell`, for example, if the root namespace, `spa`, has not yet been loaded.

**Listing A.40 – Namespaced files**

```
// == Prefer == JS files
js/spa.js          // spa.* namespace
js/spa.shell.js    // spa._shell_.* namespace
js/spa.slider.js   // spa._slider_.* namespace

// == Acceptable == CSS files
css/spa.css        // spa-*         namespace
// defines #spa, .spa-_x_clearall_

css/spa.shell.css  // spa-_shell_*  namespace
// defines .spa-_shell_, spa-_shell_header_,
//   .spa-_shell_footer_, and .spa-_shell_main_

css/spa.slider.css // spa-_slider_* namespace
// defines .spa-slider-open, spa-slider-closed

// == Prefer == PowerCSS alternatives to CSS
js/spa.css.js        // spa-*         namespace
js/spa.css_shell.js  // spa-_shell_*  namespace
js/spa.css_slider.js // spa-_slider_* namespace
```

These conventions make the interplay between CSS and JavaScript easier to manage and debug.

# 7 Use JSLint

JSLint is a very popular JavaScript validation tool written and maintained by Douglas Crockford. Use it to identify errors and ensure best practice. JSLint is automatically installed with **hi_score**.

## 7.1 Install JSLint

We may install JSLint globally as shown in Listing A.41.

**Listing A.41 – Install JSLint**

```
sudo npm install -g jslint
```

## 7.2 Configure JSLint

Our module template includes the configuration for JSLint. These settings are used to match our code standard as shown in Listing A.42.

**Listing A.42 – JSLint settings**

```
/*jslint        browser : true, continue : true,
 devel   : true, indent  : 2,    maxerr   : 50,
 newcap  : true, nomen   : true, plusplus : true,
 regexp  : true, sloppy  : true, vars     : false,
 white   : true, todo    : true, unparam  : true
*/
```

The table below describes each setting.

**Table A.15 – Example property names**

| | |
|---|---|
| `browser : true` | Allow browser keywords like document, history, clearInterval, etc. |
| `continue : true` | Allow the continue statement |
| `devel : true` | Allow development keywords like alert, console, etc. |
| `indent : 2` | Expect two-space indentation |
| `maxerr : 50` | Abort linting after 50 errors |
| `newcap : true` | Tolerate leading underscores |
| `nomen : true` | Tolerate uncapitalized constructors |
| `plusplus : true` | Tolerate ++ and -- |
| `regexp : true` | Allow useful but potentially dangerous regular expression constructions |
| `sloppy : true` | Require the use strict pragma |

| | |
|---|---|
| `vars : false` | `Don't allow multiple var statements per functional scope` |
| `white : true` | `Disable JSLint's formatting checks` |

## 7.3 Verify JavaScript

We can use JSLint from the command line whenever we want to check code validity. The syntax is `jslint <filepath1> [<filepath2>, ... <filepathN>]`. When we install **hi_score** a commit hook is installed that runs JSLint, checks whitespace, and `'use strict'` declaration on all changed, non-vendor JavaScript. One can also use JSLint or the commit hook directly as shown in Listing A.43.

**Listing A.43 – Commit hook**

```
# == Prefer ==
# Use JSLint directly
cd hi_score
PATH=$PATH:$(pwd)/node_modules/.bin
jslint js/xhi/*.js

# Check all changed, non-vendor JS files in repo
bin/git-hook_pre-commit
```

## 7.4 Share IDE configurations

We highly recommend the WebStorm JavaScript IDE. Install the **hi_score** settings to support the above conventions. Do this by selecting `File → Import settings` and then picking .the file `hi_score/cfg/webstorm-<version>-settings.jar`.

# 8 Choose libraries and frameworks carefully

8.1 Investigate third-party code like jQuery plugins before building your own—balance the cost of integration and bloat versus the benefits of standardization and code consistency.
8.2 Avoid embedding JavaScript code in HTML; use external libraries instead.
8.3 Minify, obfuscate, and compress (gzip) JavaScript and CSS before distribution.

# 9 End

Original author: Michael Mikowski for SPA book, hi_score