JavaScript code standard

- Why a code standard is important
- Format and documenting code consistently
- Name variables consistently
- Isolate code using namespaces
- Organize files and consistent syntax
- Validate code using JSLint and commit hooks
- Use templates

1. Introduction

A good code standard improves product quality by minimizing common mistakes and miscommunication. It helps delivers a better product faster by facilitating team communication and encouraging code review and reuse. It also helps avoid technical debt by encouraging self-documenting code that is understood by all.

Having a well-defined standard for a loosely typed, dynamic language like JavaScript is almost certainly more important than with stricter languages. JavaScript's very flexibility can make it a Pandora's box of coding syntax and practice. Whereas stricter languages provide structure and consistency inherently, JavaScript requires discipline and an applied standard to achieve the same effect.

The following standard has been used and improved over many years. The presentation here isn't very concise because we've added many explanations and examples. Most of it has been condensed into a three-page cheat-sheet found in this repository.

2. Code for readability first

Martin Fowler once said "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." Though well-defined and comprehensive standards don't ensure human-readable JavaScript, they certainly help, just as dictionaries and grammar guides help ensure human-readable English.

JavaScript code has two audiences that need to understand it: the machines that will execute it and the humans who will maintain or extend it. Research has shown that code is read by humans around ten times more frequently than it is written. Therefore, a consistent and considered code style is one of the best way to increase comprehension. We've found that code which is written to be understood tends to be more carefully considered and better constructed the first time around:

- It minimizes the chance of coding errors
- It results in code suitable for large-scale projects and teams
- It encourages code efficiency, effectiveness, and reuse
- It directs developers to use JavaScript's strengths and avoids its weaknesses

2.1 Use consistent indentation spacing and line lengths

We probably all have noticed the text columns in a newspaper are between 50 and 80 characters in length. Lines longer that 80 characters are progressively harder for the human eye to follow. Bringhurst's authoritative book, The Elements of Typographic Style, recommends line lengths between 45-75 characters for optimal reading comprehension and comfort, with 66 considered the optimal.

Longer lines are also hard to read on computer displays. Today more and more web pages have multi-column layouts—even though this is notoriously expensive to implement well. The only reason a web developer is going to go though such trouble is if there's a problem with long lines (or if they get paid by the hour).

Proponents for wider tab stops (4-8 spaces) say it makes their code more legible. But they often also advocate long line lengths to compensate for the wide tabs. We take the other approach: short tab width (2 spaces) and short-ish line length (78 characters) work together to provide a narrower, more legible document with significant content per line. The short tab stop also recognizes that an event-driven language like JavaScript is typically more indented than a purely procedural language due to the proliferation of callbacks and closures.

- Indent two spaces per code level.
- Use spaces, not tabs to indent as there's not a standard for the placement of tab stops.
- Limit lines to 78 characters.

Narrower documents work better across all displays, allowing an individual to open six views of files concurrently on two high-definition displays, or easily read a single document on the smaller screens found on notebooks, tablets, or smart phones. They also fit nicely as listings on e-readers or in a printed book format, which makes our editor much happier

2.2 Organize code in paragraphs

English and other written languages are presented in paragraphs to help the reader understand when one topic is complete and another is to be presented. Computer languages also benefit from this convention. These paragraphs can be annotated as a whole. Through the appropriate use of white space3 our JavaScript can read like a well-formated book.

- Organize your code in logical paragraphs and place blank lines between each.
- Each line should contain at most one statement or assignment although we do allow multiple variable declarations per line.
- Place white space between operators and variables so that variables are easier to spot.
- Place white space after every comma.
- Align like operators within paragraphs.
- Indent comments the same amount as the code they explain.
- Place a semicolon at the end of every statement.
- Place braces around all statements in a control structure. Control structures include for , if, and while constructs, among others.

- Do not omit braces on a single-line if statement even if “all the cool CofeeScript kids are doing it.” This makes it very easy to introduce subtle bugs.

When we lay out our code, we want to aim for clarity and not reduced byte-count. Once our code reaches production, our JavaScript will be concatenated, minified, and compressed before it reaches our users. As a result, the tools we use to aid comprehension like white space, comments, and more descriptive variable names will have no effect on product performance.

2.3 Break lines consistently

Always place a statement on a single line if it doesn’t exceed the maximum line length. If that is not possible, we must break it consistently to maximize readability:

- Break lines before operators as one can easily review all operators in the left column.
- Indent subsequent lines of the statement one level (two spaces).
- Break lines after comma separators.
- Place a closing bracket or parenthesis on its own line. This clearly indicates the conclusion of the statement without forcing the reader to scan horizontally for the semicolon.

2.4 Use K&R-style bracketing

K&R style bracketing balances the use of vertical space with readability. It should be used when formatting objects and maps, arrays, compound statements, or invocations. A compound statement contains one or more statements enclosed in curly braces. Examples include **`if`**, **`while`**, and **`for`** statements. An invocation like **`alert( 'I have been invoked!' );`** calls a function or a method.

- Prefer single lines when possible. For example, do not unnecessarily break a short array assignment into three lines when it can fit on one.
- Place the opening parenthesis, brace or bracket at the end of the opening line.
- Indent the code inside the delimiters (parenthesis, brace, or bracket) one level (two spaces).
- Place the closing parenthesis, brace or bracket on its own line with the same indentation as the opening line.

Adjusting elements to line up vertically really helps comprehension, but also can be time consuming if you don’t have a powerful text editor. Vertical text selection—as provided by Vim, Sublime, WebStorm, and others—is helpful in aligning values. WebStorm even provides tools to auto-align map values, which is a great time-saver. If your editor doesn’t allow for vertical selection, we highly recommend you change editors.

2.5 Use white space to distinguish functions and keywords

- Follow a function with **no space** between the function keyword and the opening left parenthesi, like **`JSON.stringify( example_map );`**

- Follow a keyword with **a single space** and then its opening left parenthesis, like **`function ( arg_map ) { … }`**
- When formatting a **`for`** statement, add a space after each semicolon.

2.6 Use quotes consistently

Prefer single quotes over double quotes for string delimiters, as the HTML standard attribute delimiter is double quotes. And HTML is typically quoted often in SPAs. HTML delimited with single quotes requires less character escaping or encoding. The result is shorter, easier to read, and less likely to have errors.

Many languages like Perl, PHP , and Bash have the concept of interpolating and non-interpolating quotes. Interpolating quotes expand variable values found inside, whereas non-interpolating quotes don't. Typically, double quotes (") are interpolating, and single quotes (') are not. JavaScript quotes never interpolate, yet both single and double quotes may be used with no variance in behavior. Our use is therefore consistent with other popular languages.

2.7 Comment to explain and document

Comments can be even more important than the code they explain because they can convey critical details that aren't otherwise obvious. This is especially evident in event-driven programming, as the number of callbacks can make tracing code execution a big time sink. This doesn't mean that adding more comments is always better.

Strategically placed, informative, and well-maintained comments are highly valued,whereas a clutter of inaccurate comments can be worse than no comments at all.

2.7.1 Explain code strategically

Our strategy is to **minimize the number of comments** by using conventions to make the meaning of the code as self-evident as possible. We then **maximize their value** by aligning them to the code blocks they describe and ensuring their content is of value to the reader.

Consistent, meaningful variable names can provide more information with substantially fewer comments. The section on variable naming appears a little later, but let's look at a few highlights: Variables that refer to functions all have a verb as their first word like **`makeSpecMap`** or **`initModule`**. Other variables are also named to help us understand their content. For example, **`welcome_html`** is an HTML string, **`house_color_list`** is an array of color names, and **`spec_map`** is a map of specifications. This helps reduce the number of comments we need to add or maintain to make the code understandable.

2.7.2 Document APIs in-line

Explain any non-trivial function by specifying its purpose, the arguments or settings it uses, the values it returns, and any exception it throws:

```
// BEGIN DOM Method /toggleSlider/
// Summary   : toggleSlider( <boolean>, [ <callback_fn> ] )
// Purpose   : Extends and retracts chat slider
// Example   : toggleSlider( true );
```

```
// Arguments : (positional)
//   0: do_extend (boolean, required).
//      A truthy value extends slider.
//      A falsey value retracts it.
//   1: callback_fn (function, optional).
//      A function that will be executed
//      after animation is complete
// Settings  :
//   * chat_extend_ms, chat_retract_ms
//   * chat_extend_ht_px, chat_retract_ht_px
// Returns   : boolean
//   * true  - slider animation successfully initiated
//   * false - slider animation not initiated
// Throws    : none
//
function toggleSlider ( do_extend, callback_fn ) { … }
// END DOM Method /toggleSlider/
```

Documentation about general architecture should not be placed in the code but instead into a dedicated architecture document. However, documentation about a function or an object API can and often should be placed right next to the code.

2.7.3 Document TODOs

We don't always have time to do things immediately, however, if we notice future work, we should create a TODO comment. These have become standard enough that JSLint and most IDEs recognize these comments as special. We recommend the following format:

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

The user name and date are valuable in deciding the freshness of the comment, and can be also used by automated tools to report on **`TODO`** items in the code base. Use syslog terms for <level>: **`emerg`**, **`alert`**, **`crit`**, **`err`**, **`warn`**, **`notice`**, **`info`**, and **`debug`**.

If you disable code, explain why with a comment using the following format:

```
// BEGIN TODO 2016-12-29 mikem warn:
//  - disabled code while testing alternative
//  - remove after testing.
// ...
//
// END TODO 2016-12-29 mikem: warn
```

Temporarily disabling code which we'll likely need again is more efficient than trying to find the version where the disabled code was pristine and then merging it back. After the code has been disabled for a while, we can safely remove it.

2.8 Name variable to mean something

Ever notice how books often include an ad-hoc naming convention in their code listings? For example, you'll see lines like **`person_str = 'fred';`**. The author typically does this because he doesn't want to insert a clumsy, time-and-focus-sapping reminder later about

what the variable represents. The name is self-evident. A good naming convention provides the greatest value when all members of a team understand it and use it. When they do, they're liberated from dull code tracing and error-prone comment maintenance, and can instead focus on the purpose and logic of the code.

Consistent and descriptive names are extremely important for enterprise-class JavaScript applications, as they can greatly speed cognition and also help avoid common errors. Consider this JavaScript code:

```
var creator = maker( 'house' );
```

Now let's rewrite it using our naming conventions, which we'll discuss shortly:

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

While this is a bit longer, it provides a great deal of information without resorting to comments. Even without knowing the standard, one can determine much of the following:

- The returned value, **`make_house_fn`** is an object constructor function.
- The called function curries the **`build_item`** function, setting **`_item_type_`** to **`'house'`**
- The called function takes a string argument that indicates a type.
- All variables and keys are local in scope (as implied by the **`snake_case`** names).
- The **`_item_type_`** is a private object key that will be compressed during the build process.

Now we could determine all this information in the first example by inspecting hundreds of lines of code from numerous files with equally useless names and running code inspector in the browser. It could take 1-2 hours to determine all this information with certainty especially if the rest of the code is so vaguely annotated.. And then we'll need to remember it all while working with or around this code. Not only will we lose lots of time, but we might lose focus on what we're trying to accomplish in the first place.

This avoidable expense will be incurred every time a new developer works with this code. And remember, after a few weeks away from this code, any developer, including the original author, is effectively a new developer. Obviously, this is horribly inefficient and error-prone.

Lets see how our first example would look if we used comments to provide the same amount of meaning as we did using our naming convention:

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

This is much more verbose, took much longer to write, does not convey as much information as the naming convention, ***and*** is much more prone to become inaccurate over time as the code changes and developers exert their laziness. IDEs are much worse at maintaining text in comments than variable names because the former can be tracked much more easily.

Let's say we decide to change a few names a few weeks later:

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

If we forget to meticulously update the comments they will be wrong and misleading as the example above shows. Not only are the comments now completely misleading, they also obscure the code because they are **nine times longer** than the code using the naming convention.

Now lets make the same change to our second example:

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

These revisions are immediately correct, as there are no comments to adjust! A well-considered naming convention like this is a great way to self-document code by the original author, with greater precision and without a clutter of comments that are near-impossible to maintain. It helps speed development, improve quality, and ease maintenance.

A variable name can convey a lot of information, as we have illustrated above. Let's step through some guidelines we've found most useful.

2.8.1 Use common characters

Though much of our team might think it clever to name a variable queensrÿcheEmpire, those who try to find the ÿ key on their keyboard might have different and significantly more negative opinions. It's better to limit variable names to characters available on most of the world's keyboards.

- Use a-z, A-Z, 0-9, undescore, and $ characters in variable names.
- Don't begin a variable name with a number.

2.8.2 Use snake_case and camelCase to communicate variable scope

Our JavaScript files and modules have a one-to-one correspondence. We've found it very useful to distinguish between variables that are available anywhere in the module and those that have a more limited scope.

- Use **`camelCase`** when the variable is module scope.

- Use **`snake_case`** when the variable is local to a function within a module.
- Ensure module scope variables have at least two syllables so that the scope is clear. For example, instead of using a variable called **`config`** we can use the more descriptive and obviously module-scoped **`configMap`**.

2.8.3 Communicate variable type with names

Just because JavaScript allows you to play fast and loose with variable types doesn't mean you should. Consider the following example:

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

In this case, JavaScript converts x into a string and concatenates it to y (02) to get the string 1002. Which is probably not what was intended. The results of type conversion can have more profound effects as well:

```
var
  x = 10,
  z = [ 03, 02, '01' ],
  i , p;

for ( i in z ) {
  p = x + z[ i ];
  console.log( p.toFixed( 2 ) );
}
// Output:
// 13.00
// 12.00
// TypeError: Object 1001 has no method 'toFixed'
```

Unintentional type conversion leads to difficult to find and solve bugs. We hardly ever purposely change a variable's type because (among other reasons) doing so is almost always too confusing or difficult to manage to be worth the benefit. One solution is either to add an enormously complex and verbose framework around JavaScript like TypeScript or Google Closure. We prefer to simply add a short suffix to names. This avoids all that overhead, and makes the code immediately more readable.

2.8.4 Avoid Plurals

Avoid using plurals in for any variable name. Variable names like **`persons`**, **`automobiles`**, **`cats`**, are inherently confusing. Is **`persons`** a string of comma separated names, or an array, or an object? Or perhaps its a boolean we use to indicate that persons now exist? There are much better, and more exact ways to communicate the correct purpose such as **`personList`**, **`personMap`**, and **`hasPeople`**.

2.8.5 Name map keys like local variables

Key names should use the same convention as local variables. However, they should be wrapped by underscores if they are local to the application. This allows them to be compressed during the build process.

2.8.6 Booleans

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

2.8.7 Strings

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	employee_name	employeeName
_txt	email_text	emailText
_type	item_type	itemType

2.8.8 Integers

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 doesn't generally expose integers as a supported variable type, but there are many instances where the language won't work properly unless we provide an integer. When

iterating over an array, for example, the use of a floating-point number as an index does not work correctly.

```
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"]'
```

Other built-ins also expect integer values, like the string `substr()` method. So let's indicate if we expect a variable to be an integer, OK people?

2.8.9 Numbers

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	—

2.8.10 Regular Expressions

Indicator	Local scope	Module scope
_rx	match_rx	matchRx

2.8.11 Arrays

Indicator	Local scope	Module scope
_list [generic]	timestamp_list color_list	timestampList colorList
_table [list of lists]	user_table	userTable

Please only using singular nouns as the suffix indicates plurality.

2.8.12 Maps

JavaScript doesn't officially have a map data type—it just has objects. But we've found it very useful to distinguish between simple objects used only to store data (maps) and full-featured objects. This map structure is analogous to 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 developer intent.

Indicator	Local scope	Module scope

_map [generic]	employee_map receipt_map	employeeMap receiptMap

Please use only singular nouns before the map suffix. Sometimes the key of a map is an unusual or a distinguishing feature. In such cases, we indicate the key in the name, for example, receipt_timestamp_map.

2.8.13 Objects

Indicator	Local scope	Module scope
_obj [generic]	employee_obj receipt_obj error_obj	employeeObj receiptObj errorObj
$ (jQuery object)	$header $area_tabs	$Header $areaTabs
_proto (protype)	user_proto	userProto

2.8.14 Functions

Functions perform actions on data. Therefore we always like to place the action verb as the first part of a function name. It is not obvious we are dealing with a function in local scope, so the fn suffix is recommended.

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

Module-scoped functions should always contain two or more syllables so the scope is clear, for example, **`getCarList`** or **`emptyCacheMap`**. The name should always indicate what data structure is returned.

Use verbs consistently as shown below:

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

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

Event handlers		
Verb	**Example**	**Meaning**
on	onMouseover onClickHeader	An event handler. Use <on><event-name><modifier>

2.8.15 Unknown type

Sometimes we really don't know what type of data our variables contain. There are two situations where this is common:

- We're writing a polymorphic function—one that accepts multiple data types.
- We're receiving data from an external data source, such as an AJAX or web socket feed.

In these cases, the primary feature of the variable is the uncertainty of its data type. We've settled on a practice of ensuring the word data is in the name.

Indicator	Local scope	Module scope
data	http_data socket_data arg_data data	httpData, socketData

2.9 Put the guidelines to use

Let’s compare the readability of an object prototype before and after we apply naming guidelines.

Before:

```
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
};
```

After:

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

  _check_destroy_ : checkDestroy,
  _destroy_dog_   : destroyDog,
  _print_greet_   : printGreet,
  _print_name_    : printName,
  _print_speak_   : printSpeak,
  _redraw_dog_    : redrawDog
  _show_flash_    : showFlash
};
```

We find the ‘after’ code to be far prettier, easier to read, and easier to maintain.

3. Variable declaration and assignment

Variables can be assigned to functions pointers, object pointers, array pointers, strings, numbers, 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's no formal interface to enforce this typing.

3.1 Avoid module scope variables

Instead, place static values in the "top config map" (**`topCmap`**) and dynamic values in the "top state map" (**`topSmap`**). Remember to use underscores on key names that are private to the module, which should be almost all of them:

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

// Better
topSmap = {
  _is_slider_open_ : false,
  _is_slider_active_ : true
};
```

3.2 Avoid the new keyword

Use {} or [] instead of new Object() or new Array() to create a new object, map,or array. Remember, a map is a simple data-only object with no methods. If you require object inheritance, use Object.create().

3.3 Copy carefully

Use utilities to copy objects and arrays. 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 for this purpose, such as those provided by jQuery.

3.4 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 eas ier to read and to detect undeclared variables which are never acceptable.

```
// Example use of a single var statement
//
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: declaring informs the JavaScript engine that the variable exists within a scope. Assigning provides the variable a value (instead of undefined). As a convenience, you may combine declaration and assignment with the var statement, but it's not required.

- Used named functions to assist with debugging

```
// GOOD
function getMapCopy( arg_map ) { ... };

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

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

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

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

- Use one line per variable assignment. Order them alphabetically or in logical groups when possible. More than one declaration may be placed on a single line:

```
// vars for lasso and drag function
var
  $cursor         = null,
  scroll_up_toid = null,

  index, length, ratio
  ;
```

4. Functions

Functions play a central role in JavaScript: they organize code, provide a container for variable scope, and they provide an execution context which can be used to construct prototype-based objects.

- Use the factory pattern for object constructors, as it better illustrates how JavaScript objects actually work, is fast, and can be used to provide class-like capabilities like object count.

```
// Preferred
var dog_obj = makeDog();

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

- Avoid pseudoclassical object constructors—those that take a new keyword. If we call such a constructor without the new keyword, the global namespace gets corrupted. If we must keep such a constructor, its first letter should be capitalized so it may be recognized as a pseudo classical constructor.
- Declare all functions before they are used
- When a function is to be invoked immediately, 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: spa.shell = (function () { ... }());

5. Namespaces

Much early JavaScript code was relatively small and used alone on a single web page. These scripts could (and often did) use global variables with few repercussions. But as JavaScript applications have become more ambitious and third-party libraries have become common, the chance that someone else is going to want the global i variable rises steeply. And when two code bases claim the same global variable, all hell can break loose.

We can greatly minimize this problem by using only a single global function inside of which all our other variables are scoped as illustrated here:

```
var spa = (function () {
  // other code here
  function initModule () {
    console.log( 'hi there' );
  };
  return { _initModule_ : initModule };
}());
```

We call this single global function (spa, in this example) our namespace. The function we assign to it executes on load, and of course, any local variables assigned within that function won't be available to the global namespace. Note that we did make the init Module method

available. So other code can call the initialization function, but it can't access anything else. And it has to use our spa prefix:

```
// from another library, call the spa initialization function
spa._initModule_();
```

We can subdivide the namespace so that we aren't forced to cram a 50KB application into a single file. For example, we can create the namespaces of spa, spa.shell, and spa.slider:

```
// In the file spa.js:
var spa = (function () {
  // some code here
}());

// In the file spa.shell.js:
var spa._shell_ = (function () {
  // some code here
}());

// In the file spa.slider.js:
var spa._slider_ = (function () {
  // some code here
}());
```

Namespacing is key to creating manageable code in JavaScript.

6. File names and layout

Namespacing is the foundation of our file naming and layout. Here are the general guidelines:

- Use jQuery for DOM manipulations.
- 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.
- Avoid embedding JavaScript code in HTML; use external libraries instead.
- Minify, obfuscate, and gzip JavaScript and CSS before go-live. For example, use Uglify and Superpack to minify and obsfucate Javascript during preparation, and use a gzip server module to compress the files on delivery.

JavaScript file guidelines are as follows:

- Include third-party JavaScript files first in our HTML so their functions may be evaluated and made ready for our application.
- Include our JavaScript files next, in order of namespace. You can't load namespace spa.shell, for example, if the root namespace, spa, has not yet been loaded.
- Give all JavaScript files a .js suffix.
- Store all static JavaScript files under a directory called js.
- Name JavaScript files according to the namespace they provide, one namespace per file:

```
spa.js        // spa.* namespace
spa.shell.js  // spa._shell_.* namespace
```

```
spa.slider.js // spa._slider_.* namespace
```

- Use the template to start any JavaScript module file. One is found at the end of this appendix.
- Maintain a parallel structure between JavaScript and CSS files and class names. Create a CSS file for each JavaScript file that generates HTML:

```
spa.css        // spa-* namespace
spa.shell.css  // spa-_shell_* namespace
spa.slider.css // spa-_slider_* namespace
```

- Store all CSS files under a css directory with css file extensions.
- 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:

```
spa.css defines #spa, .spa-_x_clearall_
spa.shell.css defines .spa-_shell_, spa-_shell_header_,
   .spa-_shell_footer_, and .spa-_shell_main_
```

- 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.
- When using PowerCSS, keep the same parallel structure, replacing CSS files with JS files like so: spa.shell.js is paired with spa.css_shell.js

These are simple guidelines and easy to follow. The resulting organization and consistency make the correlation between CSS and JavaScript much easier to understand.

7. Syntax

This section is a survey of JavaScript syntax and the guidelines we follow.

7.1 Labels

Statement labels are optional. Only these statements should be labeled: while, do, for , switch. Labels should always be uppercase and should be a singular noun:

```
var
  horseList = [ Anglo-Arabian', 'Arabian', 'Azteca', 'Clydesdale' ],
  horseCount = horseList.length,
  breedName, idx
;

HORSE: for ( idx = 0; idx < horseCount; idx++ ) {
  breedName = horseList[ idx ];
  if ( breedName === 'Clydsedale' ) { continue HORSE; }

  // process for non-bud horses below
  // ...
}
```

7.2 Statements

Common JavaScript statements are listed next, along with our suggested use.

7.2.1 continue

Avoid use of the continue statement unless we use a label. It otherwise tends to obscure the control flow. The inclusion of a label also makes continue more resilient.

```
// discouraged
continue;

// encouraged
continue HORSE;
```

7.2.2 do

A do statement should have the following form:

```
do {
// statements
} while ( condition );
```

Always end a do statement with a semicolon.

7.2.3 for

A for statement should always have the following form:

```
for ( initialization; condition; update ) {
  // statements
}
```

Avoid the for-in form. Use Object.keys() to get a list of object keys instead and iterate over that.

7.2.4 if

The if statement should have one of the forms illustrated as follows. An else keyword should begin its own line:

```
if ( condition ) {
  // statements
}

if ( condition ) {
  // statements
}
else {
```

```
  // statements
}

if ( condition ) {
  // statements
}
else if ( condition ) {
  // statements
}
else {
  // statements
}
```

7.2.5 return

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

7.2.6 switch

A switch statement should have the following form:

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

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.

7.2.7 try

```
A try statement should have one of the following forms:
try {
  // statements
}
catch ( error_data ) {
  // statements
}
try {
  // statements
}
catch ( error_data ) {
  // statements
}
```

```
finally {
  // statements
}
```

7.2.8 while, setInterval

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

7.2.9 with

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

8. Other syntax

Of course there's more to JavaScript than just labels and statements. Here are some additional guidelines we follow:

8.1.1 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, var statements, and parameter lists.

8.1.2 Avoid assignment expressions

Avoid using assignments in the condition part of if and while statements—don't write if (a = b) { ... as it's not clear if you intended to test for equality or a successful assignment.

8.1.3 Always use === and !== comparisons

It is always better to use the === and !== operators. The == and != operators do type coercion. 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, use a construct like this:

```
if ( is_drag_mode ) { // is_drag_mode is truthy!
  runReport();
}
```

8.1.4 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.

```
// confusing:
total = total_count + +arg_map.cost_dollars;

// better:
total = total_count + (+arg_map.cost_dollars);
```

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

8.1.5 Don’t use eval

Be careful— eval has evil alter-egos. Don’t use the Function constructor. Don’t pass strings to setTimeout or setInterval. Use JSON.parse() instead of eval to convert JSON strings into internal data structures.

8.1.6 Install JSLint

JSLint is a JavaScript validation tool written and maintained by Douglas Crockford. It’s very popular and useful in spotting code errors and ensuring fundamental guidelines are followed. If you’re creating professional-grade JavaScript, you should be using JSLint or a similar validator. It helps avoid numerous types of bugs and significantly shortens development time.

After you have Node.js installed, you may easily install JSLint:

```
npm install jslint
```

8.1.7 Configure JSLint

Our module template includes the configuration for JSLint. These settings are used to match our code standard:

```
/*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
*/
```

browser : true—Allow browser keywords like document, history , clearInterval, and so on
continue : true—Allow the continue statement
devel : true—Allow development keywords like alert, console, and so forth
indent : 2—Expect two-space indentation
maxerr : 50—Abort JSLint 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.

8.1.8 Use JSLint

We can use JSLint from the command line whenever we want to check code validity. The syntax is:

```
jslint filepath1 [filepath2, ... filepathN]
# example: jslint spa.js
# example: jslint *.js
```

We've written a git commit hook to test all changed JavaScript files before allowing a commit into the repository. The following shell script can be added as repo/.git/hooks/pre-commit.

8.1.9 template for modules

```
See <ns>.module-tmplt.js
```