**JavaScript Coding Guidelines**

**Authors**

[Manas](http://www.ourgoalplan.com/PublicProfile.aspx?id=1096&search=manas.pattanaik)

[Manisha](http://www.ourgoalplan.com/PublicProfile.aspx?id=485&search=manisap)

[Sathyan](http://www.ourgoalplan.com/PublicProfile.aspx?id=28)

[Suchismita](http://www.ourgoalplan.com/PublicProfile.aspx?id=1267)

[Swaraj](http://www.ourgoalplan.com/PublicProfile.aspx?id=1325&search=swaraj.gochhayat)

[Swati](http://www.ourgoalplan.com/PublicProfile.aspx?id=1274)

**Dear fellow Mindfirean,**

**This is more of a compilation and less of new writing, hopefully you would find this useful, let us know your feedback and we are sharing it in editable doc format and not in a PDF format so that you can add/edit/annotate as you like.**

This is a set of coding conventions and rules for use in JavaScript programming.

The longevity of any software depends on the quality of the codebase. Over its lifetime, a program is handled by many pairs of hands and eyes. If a program is able to clearly communicate its structure and characteristics, it is less likely that it will break when modified in the never-too-distant future.

Code conventions can help in reducing the brittleness of programs.

All of our JavaScript code is sent directly to the public. It should always be of publication quality.

Neatness counts.

Whatever you choose as the style for your project should be considered law - law of code style consistency.

**‘All code in any code-base should look like a single person typed it, no matter how many people contributed.’**

**Performance CheckList**

* No inline JavaScript.
* Remember, CSS on top and JS on Bottom.
* Refer only the necessary js files for the page.
* Always remove code that you do not need.
* Minimal comments, comment only what the code does and not how!
* Use JSON instead of long cumbersome traditional response.
* String concatenation is Bad in client side too, use join() method to append lengthy strings and long running loops!
* DRY! Don’t Repeat Yourself!
* Loops are slow and bad in any language, use only when needed.
* If you have larger number of conditions to check for same category, then better to use **switch** instead of if-else.
* switch/if-else - place all the most common conditions at first then others.
* Don't Use new Object()
  + Use {} instead of new Object()
  + Use " " instead of new String()
  + Use 0 instead of new Number()
  + Use false instead of new Boolean()
  + Use [] instead of new Array()
  + Use /()/ instead of new RegExp()
  + Use function (){} instead of new function()

**JavaScript Files**

* JavaScript programs should be stored in and delivered as .js files.
* JavaScript code should not be embedded in HTML files unless the code is specific to a single session. Code in HTML adds significantly to page weight with no opportunity for mitigation by caching and compression.
* ***<script src=filename.js>*** tags should be placed as late in the body as possible. This reduces the effects of delays imposed by script loading on other page components.
* There is no need to use the language or type attributes. It is the server, not the script tag, that determines the MIME type.

**Whitespace and Line Length**

* Never mix spaces and tabs.
* When beginning a project, before you write any code, choose between soft indents (spaces) or real tabs, stick to whatever you choose, consider this **law**.
* Avoid lines longer than 80 characters. When a statement will not fit on a single line, it may be necessary to break it.

**Comments**

* Be generous with comments. It is useful to leave information that will be read at a later time by people (possibly yourself) who will need to understand what you have done. There was a time when a lot of commenting in JS was considered bad, but with the advent of modern browsers and minification/bundling, this is not a concern anymore.
* The comments should be well-written and clear, just like the code they are annotating.
* It is important that comments be kept up-to-date.
* Erroneous comments can make programs even harder to read and understand.
* Make comments meaningful. Focus on what is not immediately visible. Don't waste the reader's time with stuff like

i = 0; // Set i to zero.

**Variables Declaration**

* All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables.
* Always declare a variable with var keyword. When you fail to specify var, the variable gets placed in the global context, potentially clobbering existing values. Also, if there's no declaration, it's hard to tell in what scope a variable exists (e.g., it could be in the Document or Window just as easily as in the local scope).
* Use of global variables should be minimized.
* The var statement should be the first statement in the function body.
* It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order if possible.

**Example:**

  var currentEntry, // currently selected table entry   
       level,        // indentation level  
        size;         // size of table

* JavaScript does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages.
* Define all variables at the top of the function.

**Function Declarations**

* All functions should be declared before they are used.
* Inner functions should follow the var statement. This helps make it clear what variables are included in its scope.
* There should be no space between the name of a function and the ( (left parenthesis) of its parameter list.
* There should be one space between the ) (right parenthesis) and the { (left curly brace) that begins the statement body.
* The body itself is indented four spaces. The } (right curly brace) is aligned with the line containing the beginning of the declaration of the function.
* Use of global functions should be minimized.

**Example:**

  function outer(c, d)  {  
       var e = c \* d;  
  
       function inner(a, b)  {  
           return (e \* a) + b;  
       }  
       return inner(0, 1);  
   }

* If a function literal is anonymous, there should be one space between the word function and the ( (left parenthesis). If the space is omitted, then it can appear that the function's name is function, which is an incorrect reading.

**Example:**

 div.onclick = function (e)  {  
       return false;  
   };  
  
   that = {  
       method: function () {  
           return this.datum;  
       },  
       datum: 0  
   };

**Naming Conventions**

* Names should be descriptive, self-explanatory but not excessively so.
* It has to be meaningful - end of story! Don’t use i, j, k etc for variables and xyz() for function names. One can easily understand by looking at your variables and function names, what is it meant for. By providing meaningful names we can avoid unnecessary long comments for each variables and functions.
* Avoid use of special characters because they may not read well or be understood elsewhere.
* Do not use $ (dollar sign) or \(backslash) in names.
* Do not use \_ (underscore) as the first or last character of a name.
* It is sometimes intended to indicate privacy, but it does not actually provide privacy in JavaScript. Most variables and functions should start with a lowercase letter.
* JavaScript issues neither a compile-time warning nor a run-time warning if a required new is omitted. Bad things can happen if new is not used, so the capitalization convention is the only defense we have.
* If you are using Accessor functions use GetVal() , SetVal()

**Example**

// bad  
 patient.age();  
  
 // good  
 patient.getAge();  
  
 // bad  
 patient.age(25);  
  
 // good  
 patient.setAge(25);

* If the property is a boolean use isVal()  or hasVal()

Example

// bad  
 if (!patient.age()) {  
  return false;  
 }  
  
 // good  
 if (!patient.hasAge()) {  
  return false;  
 }

**Simple Statements**

* Each line should contain at most one statement.
* Put a ; (semicolon) at the end of every simple statement.
* Note that an assignment statement that is assigning a function literal or object literal is still an assignment statement and must end with a semicolon.
* JavaScript allows any expression to be used as a statement. This can mask some errors, particularly in the presence of semicolon insertion.
* The only expressions that should be used as statements are assignments and invocations.

**Compound Statements**

Compound statements are statements that contain lists of statements enclosed in { } (curly braces).

* The enclosed statements should be indented four more spaces.
* The { (left curly brace) should be at the end of the line that begins the compound statement.
* The } (right curly brace) should begin a line and be indented to align with the beginning of the line containing the matching { (left curly brace).

Braces should be used around all statements, **even single statements**, when they are part of a control structure, such as an if or for statement. This makes it easier to add statements without accidentally introducing bugs.

**Multi-line Statements**

When a statement is too long to fit on one line, line breaks must occur after an operator.

// Bad

var html = "<p>The sum of " + a + " and " + b + " plus " + c

+ " is " + ( a + b + c )

;

// Good

var html = "<p>The sum of " + a + " and " + b + " plus " + c +

" is " + ( a + b + c );

Lines should be broken into logical groups if it improves readability, such as splitting each expression of a ternary operator onto its own line even if both will fit on a single line.

var baz = firstCondition( foo ) && secondCondition( bar ) ?

  qux( foo, bar ) :

  foo;

When a condition is too long to fit on one line, successive lines must be indented one extra level to distinguish them from the body.

 if ( firstCondition() && secondCondition() &&

          thirdCondition() ) {

      doStuff();

  }

**Link Chained Method Calls**

When a chain of method calls is too long to fit on one line, there must be one call per line, with the first call on a separate line from the object the methods are called on. If the method changes the context, an extra level of indentation must be used.

elements

  .addClass( "foo" )

  .children()

      .html( "hello" )

  .end()

  .appendTo( "body" );

**return Statement**

A return statement with a value should not use ( ) (parentheses) around the value. The return value expression must start on the same line as the return keyword in order to avoid semicolon insertion.

**if Statement**

The if class of statements should have the following form:

   if (condition) {

       statements

   }

   if (condition) {

       statements

   } else {

       statements

   }

   if (condition) {

       statements

   } else if (condition) {

       statements

   } else {

       statements

   }

**for Statement**

A for class of statements should have the following form:

   for (initialization; condition; update) {

       statements

   }

   for (variable in object) {

       if (filter) {

           statements

       }

   }

The first form should be used with arrays and with loops of a pre determinable number of iterations.

The second form should be used with objects. Be aware that members that are added to the prototype of the object will be included in the enumeration. It is wise to program defensively by using the hasOwnProperty method to distinguish the true members of the object:

  for (variable in object) {

       if (object.hasOwnProperty(variable)) {

           statements

       }

   }

**Example:**

o = new Object();  
o.prop = 'exists';

function changeO() {  
 o.newprop = o.prop;  
 delete o.prop;  
}  
  
o.hasOwnProperty('prop');   // returns true  
changeO();  
o.hasOwnProperty('prop');   // returns false

**while Statement**

A while statement should have the following form:

   while (condition) {

       statements

   }

**do Statement**

A do statement should have the following form:

 do {

       statements

   } while (condition);

Unlike the other compound statements, the do statement always ends with a ; (semicolon).

**switch Statement**

A switch statement should have the following form:

   switch (expression) {

   case expression:

       statements

   default:

       statements

   }

Each case is aligned with the switch. This avoids over-indentation. A case label is not a statement, and should not be indented like one.Each group of statements (except the default) should end with break, return, or throw. Do not fall through.

**try Statement**

The try class of statements should have the following form:

try {

       statements

   } catch (variable) {

       statements

   } finally {

       statements

   }

**Type Checking**

**=== and !== Operators**

Use the === and !== operators. The == and != operators do type coercion and should not be used.

**Example**

**A. Actual Types**

String

typeof variable === "string"

Number

typeof variable === "number"

Boolean

typeof variable === "boolean"

Object

typeof variable === "object"

Array:

Array.isArray( arrayLikeObject )  
(wherever possible)

Node

elem.nodeType === 1

null

variable === null

null or undefined

variable == null

undefined

Global Variables

typeof variable === "undefined"

Local Variables

variable === undefined

Properties

object.prop === undefined  
object.hasOwnProperty( prop )  
"prop" in object

**B. Coerced Types**

Consider the implications of the following...

Given this HTML

<input type="text" id="foo-input" value="1">

**// B.1.1**  
// `foo` has been declared with the value `0` and its type is `number`  
var foo = 0;  
  
// typeof foo;  
// "number"  
...  
  
// Somewhere later in your code, you need to update `foo`  
// with a new value derived from an input element  
  
foo = document.getElementById("foo-input").value;  
  
// If you were to test `typeof foo` now, the result would be `string`  
// This means that if you had logic that tested `foo` like:  
  
if ( foo === 1 ) {  
 importantTask();  
}  
  
// `importantTask()` would never be evaluated, even though `foo` has a value of "1"  
  
  
**// B.1.2**  
  
// You can pre-empt issues by using smart coercion with unary + or - operators:  
  
foo = +document.getElementById("foo-input").value;  
//    ^ unary + operator will convert its right side operand to a number  
  
// typeof foo;  
// "number"  
  
if ( foo === 1 ) {  
 importantTask();  
}  
// `importantTask()` will be called

**Example**  
var Use {}

= [ "a", "b", "c" ];  
  
// Good   
if ( array.indexOf( "a" ) >= 0 ) {  
 // ...  
}

var num = 2.5;  
parseInt( num, 10 );  
  
// is the same as...  
~~num;  
  
num >> 0;  
  
num >>> 0;  
  
// All result in 2  
  
  
Keep in mind however, that negative numbers will be treated differently...  
  
var neg = -2.5;  
parseInt( neg, 10 );  
  
// is the same as...  
  
~~neg;  
neg >> 0;  
  
// All result in -2  
// However...  
  
neg >>> 0;  
// Will result in some garbage value {376487236}

**Object and Array Expressions**

Object and array expressions can be on one line if they are short (remember the line length limits). When an expression is too long to fit on one line, there must be one property or element per line, with the opening and closing braces each on their own lines. Property names only need to be quoted if they are reserved words or contain special characters:

Examples

// Bad  
map = { ready: 9,  
   when: 4, "you are": 15 };  
  
array = [ 9,  
   4,  
   15 ];  
  
array = [ {  
   key: val  
} ];  
  
array = [ {  
   key: val  
}, {  
   key2: val2  
} ];

// Good  
map = { ready: 9, when: 4, "you are": 15 };   
array = [ 9, 4, 15 ];   
array = [ { key: val } ];   
array = [ { key: val }, { key2: val2 } ];

array = [  
   { key: val },  
   { key2: val2 }  
];

// Good as well  
map = {  
   ready: 9,  
   when: 4,  
   "you are": 15  
};

array = [  
   9,  
   4,  
   15  
];  
  
array = [  
   {  
       key: val  
   }  
];  
  
array = [  
   {  
       key: val  
   },  
   {  
       key2: val2  
   }  
];

**Strings**

Don't pass a string to “SetInterval” or “SetTimeOut”.

Consider the following code:

setInterval(

"document.getElementById('container').innerHTML += 'My new number: ' + i", 3000

);

Not only is this code inefficient, but it also functions in the same way as the "eval" function would. Never pass a string to SetInterval and SetTimeOut. Instead, pass a function name.

setInterval(someFunction, 3000);

**Comparison Operators & Equality**

* Use === and !== over == and != logically.
* Conditional IF statement evaluates with following rules:
* **Objects** evaluate to **true**
* **Undefined** evaluates to **false**
* **Null** evaluates to **false**
* **Booleans** evaluate to **the value of the boolean**
* **Numbers** evaluate to **false** if **+0, -0, or NaN**, otherwise **true**
* **Strings** evaluate to **false** if an empty string '', otherwise **true**

**Suggested Tools**

[JSLint](http://www.jslint.com/)

[JSHint](http://jshint.com/)

**References**

[https://github.com/rwaldron/idiomatic.js](https://github.com/rwaldron/idiomatic.js/#type)

<http://www.w3schools.com/js/js_conventions.asp>

<https://github.com/airbnb/javascript>

<https://developer.apple.com/legacy/library/documentation/ScriptingAutomation/Conceptual/JSCodingGuide/JSCodingGuide.pdf>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

<https://javascriptweblog.wordpress.com/2011/02/07/truth-equality-and-javascript/#more-2108>

<http://code.tutsplus.com/tutorials/24-javascript-best-practices-for-beginners--net-5399>