JavaScript - Best Practices

Use correct <script> tags

- The `language` attribute has been deprecated, and should not be used. Following HTML5, all browsers have "text/javascript" as the default script `type`, so it is standardised to be the default value . Hence, you don't need `type` either.
- For pages in XHTML 1.0 or HTML 4.01 omitting `type` is considered invalid.

Using the <noscript> tag

- Supported in most modern browsers, this tag displays a message to non-JavaScript browsers.
- Browsers that support JavaScript will ignore the tag including text between the <noscript> tags.
- This tag is not consistently supported by all browsers that support JavaScript.
- An alternative that avoids the use of this tag is to send users with JavaScript support to another page. This can be accomplished with the statement below:

```
<script type="text/javascript">
window.location="javascript-support.html";
</script>
```

Don't overuse JavaScript

- As HTML5 and CSS3 have matured as has browser support for these advanced standards, there's much less of a need to use JavaScript for some of the enhanced interactions.
- Many of the visual effects that once needed to be coded in JavaScript can now be achieved perfectly well using CSS.
- CSS is usually preferable if there is a choice. It is well supported across browsers, and isn't as commonly turned off by users.
- In the rare case that CSS isn't supported, the page is rendered as standard HTML, usually leaving a page that is at least perfectly functional, even if it is not so pretty.

Keep JavaScript Optional

Although you can detect JavaScript browsers and print custom messages, the best choice is to simply make your scripts unobtrusive.

- Keep JavaScript in separate files, assign event handlers in the JavaScript file rather than in the HTML. Browsers that don't support JavaScript will ignore them.

- Use JavaScript to enhance rather than as an essential feature.
- In cases where JavaScript is absolutely needed for execution for example in an AJAX application, you can warn users that JavaScript is required.
- You should definitely not require JavaScript in the navigation of your site. Though you can create for example drop-down menus and other tools using JavaScript, they prevent non-JavaScript browsers from viewing other areas of your site's pages.
- They also prevent search engines from viewing the entire site compromising your chances of getting search traffic.

Encapsulate, Abstract and Modularise code

- To remove complexities, scale, and future proof your code**.**
- Good code should be easy to build upon without rewriting the core.
- It is tempting and easy to write one function that does everything. However, as you extend the functionality you will find that you do the same things in several functions. To prevent that, make sure to write smaller, generic helper functions that fulfill one specific task rather than catch-all methods.
- At a later stage you can also expose these functions when using the revealing module pattern to create an API to extend the main functionality.

For example if you have been asked to do this task: "Put a red border around all fields with a class of “mandatory” when they are empty."

Bad

```
  var f = document.getElementById('mainform');
  var inputs = f.getElementsByTagName('input');
  for(var i=0,j=inputs.length;i<j;i++){
        if(inputs[i].className === 'mandatory' && inputs.value ===
  ''){
             inputs[i].style.borderColor = '#f00';
             inputs[i].style.borderStyle = 'solid';
             inputs[i].style.borderWidth = '1px';
        }
  }
```

Better - Use CSS inheritance to avoid having to loop over a lot of elements.

```
var f = document.getElementById('mainform');
var inputs = f.getElementsByTagName('input');
for(var i=0,j=inputs.length;i<j;i++){
    if(inputs[i].className === 'mandatory' && inputs.value === ''){
     inputs[i].className+=' error';
    }
}
```

OK

```
  function getElementArea(){
  var high = document.getElementById("id1").style.height;
  var wide = document.getElementById("id2").style.width;
    return high * wide;
  }
```

Better

```
  function getElementArea(elementId){
  var element = document.getElementById("elementId");
  var high = element.style.height;
  var wide = element.style.width;
  return parseInt(high) * parseInt(wide);

}
```

You could now call the function into action for a particular 'id' by passing the value of 'id' as a parameter.

```
  var area1 =
  getElementArea("id1");
  var area2 =
  getElementArea("id2");
```

Avoid describing a value with your variable or function name:

- This might not make sense in some countries:

  isOverEighteen()

- Works everywhere:

  isLegalAge()

Avoid Globals

- You run the danger of your code being overwritten by any other JavaScript added to the page after yours.
- Use closures and Modal pattern

Use short cut notations

A

- This code

```
if(v) {
    var x = v;
} else {
   var x =10;
}
```

- Is the same as...

```
var x = v || 10;
```

B

- This code

```
var direction;
if(x > 100) {
   direction = 1;
} else {
   direction = -1;
}
```

- Is the same as

```
var direction = (x > 100) ? 1 : -1;
```

Commenting

- They are added with the purpose of making the source code easier to understand, and are generally ignored by compilers and interpreters.
- Comments are sometime processed in various ways to generate documentation external to the source code itself by documentation generators, or used for integration with source code management systems and other kinds of external programming tools.
- You can use jsdoc style which will let you generate documentation later.

```
/**
* Adds two numbers
* @param {Number} a
* @param {Number} b
* @return {Number}
sum
*/
function sum(a,b) {
return a + b;
}
```

- Single line comment

```
some code //comment
```

- Multiple line comment

```
some code
/*
comment
comment
*/
```

Single and Double quotes

- You can use either. The difference is purely stylistic. However If you are using one form of quote in the string, you might want to use the other as the literal.
- You can escape with the backslash using repeated quotes within the string.
- Note that single quoting a string is in fact not according to JSON standards.

Enhance progressively

- DOM generation is slow and expensive.
- Elements that are dependent on JavaScript but are available when JavaScript is turned off are a broken promise to users.
- For example, insert navigation arrow elements via JavaScript. When JavaScript is turned off, the arrows must not show in the page. The page must degrade gracefully. Providing possibly a vertical or horizontal scrollbars to reveal more assets or using a standard navigation for pagination.

Allow for configuration translation

- Everything that is likely to change in your code should not be scattered throughout your code.
- This includes labels, CSS classes, IDs and presets.
- By putting these into a configuration object and making this one public we make maintenance easy and allow for customisation. For example:

```
carousel = function(){
        var config = {
                        CSS:{
                                classes:{
                                   current:'current',
                                   scrollContainer:'scroll'
                                 },
                         },
                          settings:{
                                    amount:5,
                                    skin:'blue',
                                    autoplay:false
                          },
                };
                function init(){

                };

        return {config:config,init:init}
}();
```

Initialise variables

It is a good coding practice to initialise variables when you declare them.

This will:

- Give cleaner code
- Provide a single place to initialise variables
- Avoid undefined values

Automatic Type Conversion

- Beware that numbers can accidentally be converted to strings or NaN (Not a Number).
- In JavaScript, the + operator is used for both addition and concatenation. This can cause problems when adding up form field values, for example, since JavaScript is a non-typed language. Form field values will be treated as strings, and if you + them together, javascript will treat it as concatenation instead of addition.
- To fix this problem, Javascript needs a hint to tell it to treat the values as numbers, rather than strings. You can use the unary + operator to convert the string value into a number. Prefixing a variable or expression with + will force it to evaluate as a number, which can then be successfully used in a math operation.

Bad

```
<form name="myform" action="[url]">
<input type="text" name="val1" value="1">
<input type="text" name="val2" value="2">
</form>
.........................................
function total() {
 var theform = document.forms["myform"];
 var total = theform.elements["val1"].value + theform.elements["val2"].value;
 alert(total); // This will alert "12", but what you wanted was 3!
```

Better

```
function total() {
 var theform = document.forms["myform"];
 var total = (+theform.elements["val1"].value) + (+theform.elements["val2"].value);
 alert(total); // This will alert 3
}
```

Never declare Number, String, or Boolean Objects

- Always treat numbers, strings, or booleans as primitive values. Not as objects.
- Declaring these types as objects, slows down execution speed, and produces nasty side effects:

Use === Comparison

- The == comparison operator always converts (to matching types) before comparison.
- The === operator forces comparison of values and type:

Use Parameter Defaults

- If a function is called with a missing argument, the value of the missing argument is set to undefined.
- Undefined values can break your code. It is a good habit to assign default values to arguments. For example:

```
function myFunction(x, y) {
        if (y === undefined) {
           y = 0;
        }
    }
```

End your switches with Defaults

- Always end your switch statements with a default. Even if you think there is no need for it.

Always Utilize JS Lint

- Don't ship or integrate code without linting. JS Lint takes a JavaScript source and scans it. If it finds a problem, it returns a message describing the problem with an approximate location of the problem in the code: http://www.jslint.com/

Where to position SCRIPTS

Place SCRIPTS at the bottom of the page. Remember the primary goal is to make pages load as quickly as possible for the user. When loading a script, the browser can't continue on until the entire file has been loaded. Thus, the user will have to wait longer before noticing any progress.

Better

```
<p>And now you know my favorite kinds of corn. </p>
<script type="text/javascript" src="path/to/file.js"></script>
<script type="text/javascript" src="path/to/anotherFile.js"></script>
</body>
</html>
```

Declaring variables in Loops

When executing lengthy "for" statements, don't make the engine work any harder than it must. For example:

Bad

```
for(var i = 0; i < someArray.length; i++) {
   var container = document.getElementById('container');
   container.innerHtml += 'my number: ' + i;
   console.log(i);
}
```

Notice how we must determine the length of the array for each iteration, and how we traverse the dom to find the "container" element each time -- highly inefficient!

Better

```
var container = document.getElementById('container');
for(var i = 0, len = someArray.length; i < len;  i++) {
   container.innerHtml += 'my number: ' + i;
   console.log(i);
}
```

Don't pass 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.

Instead, pass a function name.

```
setInterval(someFunction, 3000);
```

Use {} instead of New Object()

Objects literals enable us to write code that supports lots of features yet still make it a relatively straightforward for the implementers of our code. No need to invoke constructors directly or maintain the correct order of arguments passed to functions.

There are multiple ways to create objects in JavaScript. Perhaps the more traditional method is to use the "new" constructor, like so:

```
var o = new Object();
o.name = 'Jeffrey';
o.lastName = 'Way';
o.someFunction = function() {
   console.log(this.name);
}
```

However, this method receives the "bad practice" stamp without actually being so. Use rather the much more robust object literal method.

Better

```
var o = {
   name: 'Jeffrey',
   lastName = 'Way',
   someFunction : function() {
      console.log(this.name);
   }
};
```

Note that if you simply want to create an empty object, {} will do the trick.

Use [] instead of New Array()

The same applies for creating a new array.

Okay

```
var a = new Array();
a[0] = "Joe";
a[1] = 'Plumber';
```

Better

```
var a = ['Joe','Plumber'];
```

"A common error in JavaScript programs is to use an object when an array is required or an array when an object is required. The rule is simple: when the property names are small sequential integers, you should use an array. Otherwise, use an object." - Douglas Crockford.

Don't overuse the "var" keyword

Long List of Variables? Omit the "Var" Keyword and use commas Instead.

```
var someItem = 'some string';
var anotherItem = 'another string';
var oneMoreItem = 'one more string';
```

Better

```
var someItem = 'some string',
    anotherItem = 'another string',
    oneMoreItem = 'one more string';
```

This should be rather self-explanatory. There is no real speed improvements here, but it cleans up your code a bit.

Always use semi-colons to terminate statements

Even if it is a one line statement. Technically, most browsers will allow you to get away with omitting semi-colons.

```
var someItem = 'some string'
function doSomething() {
  return 'something'
}
```

Having said that, this is a very bad practice that can potentially lead to much bigger, and harder to find, issues.

Better

```
var someItem = 'some string';
function doSomething() {
  return 'something';
}
```

"For in" statements

When looping through items in an object, you might find that you'll also retrieve method functions as well. In order to work around this, always wrap your code in an if statement which filters the information

```
for(key in object) {
   if(object.hasOwnProperty(key) {
      ...then do something...
   }
}
```

```
var people = ["Bonnie", "Isaac", "Bridget", "Ted", "Jamey"];
for (var person in people){
  if( people.hasOwnProperty(person)) {
    window.alert("Hello, " + people[person]);
   }
}
```

Use Firebug's "Timer" to optimise your code

A quicker and easy way to determine how long an operation takes is to use Firebug's "timer" feature to log the results.

```
function TimeTracker(){
 console.time("MyTimer");
 for(x=5000; x > 0; x--){}
 console.timeEnd("MyTimer");
}
```

Self executing functions

Rather than calling a function, it's quite simple to make a function run automatically when a page loads, or a parent function is called. Simply wrap your function in parenthesis, and then append an additional set, which essentially calls the function.

```
(function doSomething() {
   return {
      name: 'jeff',
      lastName: 'way'
   };
})();
```

JSON

When storing data structures as plain text or sending/retrieving data structures via Ajax, use JSON (JavaScript Object Notation) instead of XML when possible. JSON is a more compact and efficient data format, and is language-neutral. JSON is a subset of the object literal notation of JavaScript. Since JSON is a subset of JavaScript, it can be used in the language with no fuss.

- In this example, an object is created containing a single member "bindings", which contains an array containing three objects, each containing "ircEvent", "method", and "regex" members.

```
        var myJSONObject = {"bindings": [
                {"ircEvent": "PRIVMSG", "method": "newURI", "regex": "^http://.*"},
                {"ircEvent": "PRIVMSG", "method": "deleteURI", "regex": "^delete.*"},
                {"ircEvent": "PRIVMSG", "method": "randomURI", "regex": "^random.*"}
            ]
        };
```

- Members can be retrieved using dot or subscript operators.

```
myJSONObject.bindings[0].method    // "newURI"
```

eval() or JSON parser?

- The `eval` function is very fast. However, it can compile and execute any JavaScript program, so there can be security issues. The use of `eval` is indicated when the source is trusted and competent.
- It is much safer to use a JSON parser. In web applications over XMLHttpRequest, communication is permitted only to the same origin that provide that page, so it is trusted. But it might not be competent. If the server is not rigorous in its JSON encoding, or if it does not scrupulously validate all of its inputs, then it could deliver invalid JSON text that could be carrying dangerous script. The `eval` function would execute the script, unleashing its malice.
- To convert a JSON text into an object, you can use the eval() function. eval() invokes the JavaScript compiler. Since JSON is a proper subset of JavaScript, the compiler will correctly parse the text and produce an object structure. The text must be wrapped in parenthesis to avoid tripping on an ambiguity in JavaScript's syntax.

```
var myObject = eval('(' + myJSONtext + ')');
```

- To defend against this, a JSON parser should be used. A JSON parser will recognise only JSON text, rejecting all scripts. In browsers that provide native JSON support, JSON parsers are also much faster than `eval`.

```
        var myObject = JSON.parse(myJSONtext, reviver);
```

- The optional `reviver` parameter is a function that will be called for every key and value at every level of the final result. Each value will be replaced by the result of the `reviver` function. This can be used to reform generic objects into instances of pseudoclasses, or to transform date strings into `Date` objects.

```
        myData = JSON.parse(text, function (key, value) {
            var type;
            if (value && typeof value === 'object') {
                type = value.type;
                if (typeof type === 'string' && typeof window[type] === 'function') {
                    return new (window[type])(value);
                }
            }
            return value;
        });
```

Avoid sync "AJAX" calls

- When making "Ajax" requests, you may choose either async or sync mode. Async mode runs the request in the background while other browser activities can continue to process. Sync mode will wait for the request to return before continuing.
- Requests made with sync mode should be avoided. These requests will cause the browser to lock up for the user until the request returns. In cases where the server is busy and the response takes a while, the user's browser (and maybe OS) will not allow anything else to be done. In cases where a response is never properly received, the browser may continue to block until the request is timed out.
- If you think that your situation requires sync mode, it is most likely time to re-think your design. Very few (if any) situations actually require Ajax requests in sync mode.

Closure

Variables declared inside a function are called local variables and can only be used within that function. Conversely, “global” variables are declared outside functions and can be accessed by any function within the same document.

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function. It also forms a *closure*.

A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Since a nested function is a closure, this means that a nested function can "inherit" the arguments and variables of its containing function. In other words, the inner function contains the scope of the outer function.

- The inner function can be accessed only from statements in the outer function.

The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

```
  function addSquares(a,b) {
     function square(x) {
        return   x * x;
     }
     return square(a) + square(b);
  }
```

Hoisting

- Hoisting is (to many developers) an unknown or overlooked behaviour of JavaScript. If a developer doesn't understand hoisting, programs may contain bugs. To avoid bugs, always declare all variables at the beginning of every scope. Since this is how JavaScript interprets the code, it is always a good rule.
- Hoisting is JavaScript's default behaviour of moving all declarations to the top of the current scope (to the top of the current script or the current function).
- In JavaScript, a variable can be declared after it has been used. In other words, a variable can be used before it has been declared.
- JavaScript only hoists declarations, not initialisations.

Avoid 'with'

- The 'with' statement in JavaScript inserts an object at the front scope chain, so any property/variable references will first try to be resolved against the object. This is often used as a shortcut to avoid multiple long references.

Example Using 'with'

```
with (document.forms["mainForm"].elements) {
   input1.value = "junk";
   input2.value = "junk";
}
```

- The problem is that the programmer has no way to verify that input1 or input2 are actually being resolved as properties of the form elements array. It is checked first for properties with these names, but if they aren't found then it continues to search up the scope chain. Eventually, it reaches the global object where it tries to treat "input1" and "input2" as global variables and tries to set their "value" properties, which result in an error.
- Instead, create a reference to the reused object and use it to resolve references.

```
var elements = document.forms["mainForm"].elements;
elements.input1.value = "junk";
elements.input2.value = "junk";
```

Feature detect rather than Browser detect

- Some code is written to detect browser versions and to take different action based on the user agent being used. This, in general, is a very bad practice. Any code which even looks at the global "navigator" object is suspect.
- The better approach is to use feature detection or feature sensing. That is, before using any advanced feature that an older browser may not support, check to see if the function or property exists first, then use it. This is better than detecting the browser version specifically, and assuming that you know its capabilities.
- If the problem is that a feature is missing in one browser, use the feature sensing to check for that feature:

```
if (document.getElementById) {
   var element = document.getElementById('MyId');
}
else {
   alert('Your browser lacks the capabilities required to run this script!');
}
```

- To support multiple event handlers you can use some "*if*" statement with either the W3C method or Microsoft's method. The following code adds the "*clickMe()*" function as an event for the element with the id attribute "*btn*":

```
        obj = document.getElementById("btn");
            if(obj.addEventListener){
             obj.addEventListener('click',clickMe,false);
            } else if(obj.attachEvent){
                obj.attachEvent('onclick',clickMe);
            }
              else {
                obj.onclick = clickMe;
          }
```

Dealing with Browser Quirks

As you develop a complex script and test in multiple browsers, you might run across a situation in which your perfectly standard code works as it should in one Browser but not in another. Assuming you have eliminated the possibility of a problem with your script, you've probably run into a browser bug or a difference in features between browsers. Some tips to use include:

- Double check for a bug in your own code.
- If a feature is missing in one browser, use feature sensing to check for the feature.
- When all else fails, use the *navigator* object to detect a particular browser, and substitute some code that works in that browser.

Error Handling

- All code has errors at some point. Handling errors well from beginning is helpful. A useful way to try and intercept potential errors and deal with them cleanly is by using *"try"* and *"catch"* statements.
- The "*try*" statement enables you to attempt to run a piece of code. If the code runs without errors, all is well.
- However, should an error occur you can use the "*catch*" statement to intervene before an error message is sent to the user, and determine what the program should then do about the error:

```
        try {
            doSomething();
        }
        catch(err){
           doSomethingElse();
        }
```

Note the syntax

```
        catch(identifier)
```

The "*identifier"* is an object created when an error is caught. It contains information about the error. If you wanted to alert the user to the nature of a JavaScript runtime error, you could use a code construct like this to open a dialog containing details of the code error:

```
        catch(err) {
            alert(err.description);
        }
```

Square brackets versus dot notations

- One good rule of thumb is to use dot notation to access standard properties of objects, and square bracket notation to access properties which are defined as objects in the page.
- Mixing the use of dot and square bracket notation makes it clear which properties are standard and which are names defined by the content:

```
document.forms["myformname"].elements["myinput"].value
```

- Here, the forms property is a standard property of document, while the form name myformname is defined by the page content. Likewise, the elements property and value property are both defined by the specs, but then myinput name is defined in the page. This syntax is very clear and easy to understand and is a recommended convention to follow, but not a strict rule.
- With dot notation, the property name is hard-coded and cannot be changed at run-time. With bracket notation, the property name is a string which is evaluated to resolve the property name.
  This Will Work.
- The string can be hard-coded, or a variable, or even a function call which returns a string property name.
  - If a property name is being generated at run-time, the bracket notation is required. For example, if you have properties "value1", "value2", and "value3", and want to access the property using a variable i=2:

    ```
    MyObject["value"+i]
    ```

    This Will Not

    ```
    MyObject.value+i
    ```

Reference FORMS and FORM Elements correctly

- All forms in an HTML form should have a name attribute. For XHTML documents, the name attribute is not required and instead the form tag should have an id attribute and should be referenced using document.getElementById().
- Referencing forms using indexes, such as document.forms[0] is a bad practice in almost all cases. Some browsers make the form available as a property of the document itself using its name. This is not reliable and shouldn't be used.
- Using square bracket notation and correct object references to show is the most fool-proof way of referencing a form input.
- If you will be referencing multiple form elements within a function, it's best to make a reference to the form object first and store it in a variable. This avoids multiple lookups to resolve the form object reference.

```
var formElements = document.forms["mainForm"].elements;
formElements["input1"].value="a";
formElements["input2"].value="b";
```

When validating an input field using onChange or similar event handlers, it is always a good idea to pass a reference to the input element itself into the function. Every input element within a form has a reference to the form object that it is contained in.

```
<input type="text" name="address" onChange="validate(this)">
          ..............................
function validate(input_obj) {
   // Get a reference to the form which contains this element
   var theform = input_obj.form;
   // Now you can check other inputs in the same form without
   // hard-coding a reference to the form itself
   if (theform.elements["city"].value=="") {
      alert("Error");
   }
}
```

By passing a reference to the form element and accessing its form property, you can write a function which does not contain a hard reference to any specific form name on the page. This is a good practice because the function becomes more reusable.

Avoid 'document.all'

- document.all was introduced by Microsoft in IE and is not a standard JavaScript DOM feature. Although many newer browsers do support it to try to support poorly-written scripts that depend on it, many browsers do not.
- There is never a reason to use document.all in JavaScript except as a fall-back case when other methods are not supported.
- You should never use document.all support as a way to determine if the browser is IE, since other browsers also now support it.
- Only Use document.all As A Last Resort

```
if (document.getElementById) {
 var obj = document.getElementById("myId");
}
else if (document.all) {
 var obj = document.all("myId");
}

}
```

Some rules for using document.all are

- Always try other standard methods first
- Only fall back to using document.all as a last resort
- Only use it if you need to support IE versions earlier than 5.0
- Always check that it is supported with "if (document.all) { }" around the block where you use it.

Accessibility

- Accessibility guidelines require scripted interfaces to be accessible. While WCAG 1.0 from 1999 required that pages be functional and accessible with scripting disabled, WCAG 2.0 and all other modern guidelines allow you to require JavaScript, but the scripted content or interactions must be compliant with the guidelines.
- It is important to keep in mind, however, that some users do disable JavaScript or may be using technologies that don't support or fully support scripting.
- If your web page or application requires scripting, ensure that you account for users without JavaScript.
- While this does not necessarily mean that all functionality must work without scripting, if it does not work without scripting, you must avoid a confusing or non-functional presentation that may appear to function, but does not because of lack of JavaScript support.

ECMAScript 2015 (ES6)

- https://github.com/airbnb/javascript