JavaScript Naming Conventions

This document defines JavaScript naming conventions, which are split into essential, coding and naming conventions.

Contents

Essential Conventions

Essential conventions include generic patterns that should be adhered in order to write *readable*, *consistent* and *maintainable code*.

Minimizing Globals

Variable declarations should always be made using **var** to not declare them as global variables. This avoids conflicts from using a variable name across different functions as well as conflicts with global variables declared by 3rd party plugins.

Good Example

```
function sum(x, y) {
  var result = x + y;
  return result;
}
```

Bad Example

```
function sum(x, y) {
  // missing var declaration, implied global
  result = x + y;
  return result;
}
```

Use JSLint

JSLint (http://jslint.com/) is a quality tools that inspects code and warns about potential problems. It is available online and can also be integrated into several development environments, so errors will be highlighted when writing code.

Before validating your code in JSLint, you should use a code beautifier to fix basic syntax errors (like indentation) automatically. There are a number of beautifiers available online. The following seem to be the best working:

- JSbeautifier.org
- JS-Beautify

Here, javascript sources have to begin with this header: `/*jslint indent: 2, maxlen: 80, nomen: true */`, which means it uses two spaces indentation, 80 maximum characters by line and allow the use of '_' as first variable name character. Other JSLint options can be added in sub functions if necessary; Allowed options are:

- `ass: true` if assignment should be allowed outside of statement position.
- `bitwise: true` if bitwise operators should be allowed.
- `continue: true` if the `continue` statement should be allowed.
- `newcap: true` if Initial Caps with constructor function is optional.
- `regexp: true` if `.` and `[^...]` should be allowed in RegExp literals. They match more material than might be expected, allowing attackers to confuse applications. These forms should not be used when validating in secure applications.
- `unparam: true` if warnings should not be given for unused parameters.

Coding Conventions

Coding conventions include generic patterns that ensure that written code adheres to certain *formatting* conventions.

Uses two-space indentation

Tabs and 2-space indentation are being used equally. Since a lot of errors on JSLint often result from *mixed use of space and tab* using 2 spaces throughout prevents these errors up front.

Good Example

```
function outer(a, b) {
  var c = 1,
    d = 2,
    inner;
  if (a > b) {
    inner = function () {
      return {
        "r": c - d
      };
    };
  } else {
    inner = function () {
      return {
        "r": c + d
      };
    };
  }
  return inner;
}
```

Bad Example

```
function outer(a, b) {
var c = 1,
d = 2,
inner;

if (a > b) {
inner = function () {
return {
"r": c - d
}}};
```

Using shorthand for conditional statements

An alternative for using braces is the shorthand notation for conditional statements. When using multiple conditions, the conditional statement can be split on multiple lines.

Good Example

```
// single line
var results = test === true ? alert(1) : alert(2);

// multiple lines
var results = (test === true && number === undefined ?
               alert(1) : alert(2));

var results = (test === true ?
               alert(1) : number === undefined ?
               alert(2) : alert(3));
```

Bad Example

```
// multiple conditions
var results = (test === true && number === undefined) ?
  alert(1) :
  alert(2);
```

Opening Brace Location

Always put the opening brace on the same line as the previous statement.

Bad Example

```
function func()
{
  return
  {
    "name": "Batman"
  };
}
```

Good Example

```
function func () {
  return {
    "name": "Batman"
  };
}
```

Closing Brace Location

The closing brace should be on the same indent as the original function call.

Bad Example

```
function func() {
  return {
           "name": "Batman"
         };
}
```

Good Example

```
function func() {
  return {
    "name": "Batman"
  };
}
```

Function Declaration Location

Non anonymous functions should be declared before use.

Bad Example

```
// [...]
return {
  "namedFunction": function namedFunction() {
    return;
  }
};
```

Good Example

```
// [...]
function namedFunction() {
  return;
}
return {
  "namedFunction": namedFunction
};
```

Object Declaration

On some interpreters, declaring an object with object keys not wrapped in quotes can throw syntax errors. Here we use double quotes.

Bad example

```
var my_object = {
  key: "value"
};
```

Good example

```
var my_object = {
  "key": "value"
};
```

Naming Conventions

Naming conventions include generic patterns for setting names and identifiers throughout a script.

Constructors

A constructor function starting with new should always start with a capital letter

```
// bad example
var test = new application();

// good example
var test = new Application();
```

Methods/Functions

A method/function should always start with a small letter.

```
// bad example
function MyFunction() {...}

// good example
function myFunction() {...}
```

TitleCase, camelCase

Follow the camel case convention, typing the words in lower-case, only capitalizing the first letter in each word.

Examples

```
// Good example constructor = TitleCase
var test = new PrototypeApplication();

// Bad example constructor
var test = new PROTOTYPEAPPLICATION();

// Good example functions/methods = camelCase
myFunction();
calculateArea();

// Bad example functions/methods
MyFunction();
CalculateArea();
```

Variables

Variables with multiple words should always use an underscore between words.

Example

```
// bad example
var deliveryNote = 1;

// good example
var delivery_note = 1;
```

Confusing variable names should end with the variable type.

Example

```
// implicit type
var my_callback = doSomething();
var Person = require("./person");

// confusing names + var type
var do_something_function = doSomething.bind(context);
```

```
var value_list = getObjectOrArray();
// value_list can be an object which can be cast into an array
```

To use camelCase, when sometimes it is impossible to declare a function directly, the function variable name should match some patterns which shows that it is a function.

```
// good example
var doSomethingFunction = function () { ... };
// or
var tool = {"doSomething": function () { ... }};

// bad example
var doSomething = function () { ... };
```

Element Classes and IDs

JavaScript can access elements by their ID attribute and class names. When assigning IDs and class names with multiple words, these should also be separated by an underscore (same as variables).

Example

```
// bad example
test.setAttribute("id", "uniqueIdentifier");

// good example
test.setAttribute("id", "unique_identifier");
```

Discuss - checked with jQuery UI/jQuery Mobile, they don't use written name conventions, only

- events names should fit their purpose (pageChange for changing a page)
- element classes use “-” like in ui-shadow
- "ui" should not be used by third party developers
- variables and events use lower camel-case like pageChange and activePage

Underscore Private Methods

Private methods should use a leading underscore to separate them from public methods (although this does not technically make a method private).

Good Example

```
var person = {
  "getName": function () {
    return this._getFirst() + " " + this._getLast();
  },
  "_getFirst": function () {
    // ...
  },
  "_getLast": function () {
    // ...
  }
};
```

Bad Example

```
var person = {
  "getName": function () {
    return this.getFirst() + " " + this.getLast();
  },
  // private function
  "getFirst": function () {
    // ...
```

```
  }
};
```

No Abbreviations

Abbreviations should not be used to avoid confusion

Good Example

```
// delivery note
var delivery_note = 1;
```

Bad Example

```
// delivery note
var del_note = 1;
```

No Plurals

Plurals should not be used when assigning names

Good Example

```
var delivery_note_list = ["one", "two"];
```

Bad Example

```
var delivery_notes = ["one", "two"];
```

Use Comments

Should be used with reason but include enough information so that a reader can get a first grasp of what a part of code is supposed to do.

Good Example

```
var person = {
  // returns full name string
  "getName": function () {
    return this._getFirst() + " " + this._getLast();
  }
};
```

Bad Example

```
var person = {
  "getName": function () {
    return this._getFirst() + " " + this._getLast();
  }
};
```

Documentation

You can use YUIDoc (http://yuilibrary.com/projects/ yuidoc) and their custom comment tags together with Node.js to generate the documentation from the script file itself. Comments will look something like this:

Good Example

```
/**
```

```
 * Reverse a string
 *
 * @param  {String} input_string String to reverse
 * @return {String} The reversed string
 */
function reverse(input_string) {
  // ...
  return output_string;
};
```

Bad Example

```
function reverse(input_string) {
  // ...
  return output_string;
};
```

Additional Readings

Resources, additional reading materials and links used

- Javascript Patterns, main ressource used.
- JSLint, code quality.
- YUIDoc, generate documentation from code.

Status:

- *2012-11-16 first version*
- *2013-01-10 updated to JSLint, removed rules enforced by JSLint*