Coding Guidelines for JavaScript

Who put this together?

There's no point in creating a whole document on coding standards specifically for Talent Toolbox. Coding guidelines should be pretty standard wherever you work.

So instead of creating our own, we adapted the guidelines made freely available by airbnb at github.com/airbnb/javascript.

What is this?

This document attempts to provide guidelines (or coding standards if you like) for coding in JavaScript that are both useful and pragmatic. Of course, if you create such a document you should practice what you preach.

ReSharper, Visual Studio's Static Code Analysis (which is also known as FxCop) and StyleCop can already automatically enforce a lot of coding and design rules by analyzing the JavaScript files in your project. This document just provides an additional set of rules and recommendations that should help you end up with a better maintainable code base.

Why would you use this document?

Although some might see coding guidelines as undesired overhead or something that limits creativity, this approach has already proven its value for many years. Why? Well, because not every developer

- is aware that code is generally read 10 times more than it is changed;
- is aware of the potential pitfalls of certain aspects of JavaScript;
- is up to speed with what modern and legacy browsers expect of JavaScript code;
- realizes that not every developer is as capable, skilled or experiences to understand elegant, but potentially very abstract solutions;

Basic principles

Not everything is covered in this document. It's already hefty and a lot to take in, but could be a lot worse. If you follow these principles you'll do well:

- The Principle of Least Surprise (or Astonishment), which means that you should choose a solution that does not include anything people might not understand or that put them on the wrong track.
- Keep It Simple Stupid (a.k.a. KISS), a funny way of saying that the simplest solution is more than sufficient.
- You Ain't Gonna Need It (a.k.a. YAGNI), which tells you to create a solution for the problem at hand rather than the ones you think will happen later on. Since when can you predict the future?
- Don't Repeat Yourself (a.k.a. DRY), which encourages you to prevent duplication within a component, a source control repository or a bounded context, without forgetting the Rule of Three heuristic.
- In general, generated code should not need to comply with coding guidelines. However, if it is possible to modify the templates used for generation, try to make them generate code that complies as much as possible.

Regardless of the elegance of somebody's solution, if it's too complex for the ordinary developer, exposes unusual behavior, or tries to solve many possible future issues, it is very likely the wrong solution and needs redesign. The worst response a developer can give you to these principles is: "But it works?".

Table of Contents

1. Types
2. Objects
3. Arrays
4. Strings
5. Functions
6. Properties
7. Variables
8. Hoisting

9. Conditional Expressions & Equality
10. Blocks
11. Comments
12. Whitespace
13. Commas
14. Semicolons
15. Type Casting & Coercion
16. Naming Conventions
17. Accessors
18. Constructors
19. Events
20. Modules
21. jQuery
22. ECMAScript 5 Compatibility
23. Testing
24. Performance
25. Resources
26. In the Wild
27. Translation
28. The JavaScript Style Guide Guide
29. Contributors
30. License

Types

- **Primitives**: When you access a primitive type you work directly on its value
  - `string`
  - `number`
  - `boolean`
  - `null`
  - `undefined`

```
var foo = 1,
    bar = foo;

bar = 9;

console.log(foo, bar); // => 1, 9
```

- **Complex**: When you access a complex type you work on a reference to its value
  - `object`
  - `array`
  - `function`

```
var foo = [1, 2],
    bar = foo;

bar[0] = 9;

console.log(foo[0], bar[0]); // => 9, 9
```

Objects

- Use the literal syntax for object creation.

```
// bad
var item = new Object();
```

```
// good
var item = {};
```

- Don't use reserved words as keys. It won't work in IE8. More info

```
// bad
var superman = {
  default: { clark: 'kent' },
  private: true
};

// good
var superman = {
  defaults: { clark: 'kent' },
  hidden: true
};
```

- Use readable synonyms in place of reserved words.

```
// bad
var superman = {
  class: 'alien'
};

// bad
var superman = {
  klass: 'alien'
};

// good
var superman = {
  type: 'alien'
};
```

Arrays

- Use the literal syntax for array creation

```
// bad
var items = new Array();

// good
var items = [];
```

- If you don't know array length use Array#push.

```
var someStack = [];


// bad
someStack[someStack.length] = 'abracadabra';

// good
someStack.push('abracadabra');
```

- When you need to copy an array use Array#slice. jsPerf

```
var len = items.length,
    itemsCopy = [],
    i;

// bad
```

```
for (i = 0; i < len; i++) {
  itemsCopy[i] = items[i];
}

// good
itemsCopy = items.slice();
```

- To convert an array-like object to an array, use Array#slice.

```
function trigger() {
  var args = Array.prototype.slice.call(arguments);
  ...
}
```

Strings

- Use single quotes `''` for strings

```
// bad
var name = "Bob Parr";

// good
var name = 'Bob Parr';

// bad
var fullName = "Bob " + this.lastName;

// good
var fullName = 'Bob ' + this.lastName;
```

- Strings longer than 80 characters should be written across multiple lines using string concatenation.

- Note: If overused, long strings with concatenation could impact performance. jsPerf & Discussion

```
// bad
var errorMessage = 'This is a super long error that was thrown because of Batman. When you stop to think about how Batman had

// bad
var errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// good
var errorMessage = 'This is a super long error that was thrown because ' +
  'of Batman. When you stop to think about how Batman had anything to do ' +
  'with this, you would get nowhere fast.';
```

- When programmatically building up a string, use Array#join instead of string concatenation. Mostly for IE: jsPerf.

```
var items,
    messages,
    length,
    i;

messages = [{
  state: 'success',
  message: 'This one worked.'
}, {
  state: 'success',
  message: 'This one worked as well.'
}, {
  state: 'error',
```

```
  message: 'This one did not work.'
}];

length = messages.length;

// bad
function inbox(messages) {
  items = '<ul>';

  for (i = 0; i < length; i++) {
    items += '<li>' + messages[i].message + '</li>';
  }

  return items + '</ul>';
}

// good
function inbox(messages) {
  items = [];

  for (i = 0; i < length; i++) {
    items[i] = messages[i].message;
  }

  return '<ul><li>' + items.join('</li><li>') + '</li></ul>';
}
```

Functions

- Function expressions:

```
// anonymous function expression
var anonymous = function() {
  return true;
};

// named function expression
var named = function named() {
  return true;
};

// immediately-invoked function expression (IIFE)
(function() {
  console.log('Welcome to the Internet. Please follow me.');
})();
```

- Never declare a function in a non-function block (if, while, etc). Assign the function to a variable instead. Browsers will allow you to do it, but they all interpret it differently, which is bad news bears.

- **Note:** ECMA-262 defines a `block` as a list of statements. A function declaration is not a statement. Read ECMA-262's note on this issue.

```
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// good
var test;
if (currentUser) {
  test = function test() {
    console.log('Yup.');
  };
}
```

- Never name a parameter `arguments`, this will take precedence over the `arguments` object that is given to every function scope.

```
// bad
function nope(name, options, arguments) {
  // ...stuff...
}

// good
function yup(name, options, args) {
  // ...stuff...
}
```

Properties

- Use dot notation when accessing properties.

```
var luke = {
  jedi: true,
  age: 28
};

// bad
var isJedi = luke['jedi'];

// good
var isJedi = luke.jedi;
```

- Use subscript notation `[]` when accessing properties with a variable.

```
var luke = {
  jedi: true,
  age: 28
};

function getProp(prop) {
  return luke[prop];
}

var isJedi = getProp('jedi');
```

Variables

- Always use `var` to declare variables. Not doing so will result in global variables. We want to avoid polluting the global namespace. Captain Planet warned us of that.

```
// bad
superPower = new SuperPower();

// good
var superPower = new SuperPower();
```

- Use one `var` declaration for multiple variables and declare each variable on a newline.

```
// bad
var items = getItems();
var goSportsTeam = true;
var dragonball = 'z';

// good
var items = getItems(),
    goSportsTeam = true,
```

```
    dragonball = 'z';
```

- Declare unassigned variables last. This is helpful when later on you might need to assign a variable depending on one of the previous assigned variables.

```
// bad
var i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
var i, items = getItems(),
    dragonball,
    goSportsTeam = true,
    len;

// good
var items = getItems(),
    goSportsTeam = true,
    dragonball,
    length,
    i;
```

- Assign variables at the top of their scope. This helps avoid issues with variable declaration and assignment hoisting related issues.

```
// bad
function() {
  test();
  console.log('doing stuff..');

  //..other stuff..

  var name = getName();

  if (name === 'test') {
    return false;
  }

  return name;
}

// good
function() {
  var name = getName();

  test();
  console.log('doing stuff..');

  //..other stuff..

  if (name === 'test') {
    return false;
  }

  return name;
}

// bad
function() {
  var name = getName();

  if (!arguments.length) {
    return false;
  }

  return true;
```

```
}

// good
function() {
  if (!arguments.length) {
    return false;
  }

  var name = getName();

  return true;
}
```

Hoisting

- Variable declarations get hoisted to the top of their scope, their assignment does not.

```
// we know this wouldn't work (assuming there
// is no notDefined global variable)
function example() {
  console.log(notDefined); // => throws a ReferenceError
}

// creating a variable declaration after you
// reference the variable will work due to
// variable hoisting. Note: the assignment
// value of `true` is not hoisted.
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// The interpreter is hoisting the variable
// declaration to the top of the scope.
// Which means our example could be rewritten as:
function example() {
  var declaredButNotAssigned;
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}
```

- Anonymous function expressions hoist their variable name, but not the function assignment.

```
function example() {
  console.log(anonymous); // => undefined

  anonymous(); // => TypeError anonymous is not a function

  var anonymous = function() {
    console.log('anonymous function expression');
  };
}
```

- Named function expressions hoist the variable name, not the function name or the function body.

```
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  superPower(); // => ReferenceError superPower is not defined

  var named = function superPower() {
    console.log('Flying');
```

```
  };
}

// the same is true when the function name
// is the same as the variable name.
function example() {
  console.log(named); // => undefined

  named(); // => TypeError named is not a function

  var named = function named() {
    console.log('named');
  }
}
```

- Function declarations hoist their name and the function body.

```
function example() {
  superPower(); // => Flying

  function superPower() {
    console.log('Flying');
  }
}
```

- For more information refer to JavaScript Scoping & Hoisting by Ben Cherry

Conditional Expressions & Equality

- Use `===` and `!==` over `==` and `!=`.
- Conditional expressions are evaluated using coercion with the `ToBoolean` method and always follow these simple 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**

```
if ([0]) {
  // true
  // An array is an object, objects evaluate to true
}
```

- Use shortcuts.

```
// bad
if (name !== '') {
  // ...stuff...
}

// good
if (name) {
  // ...stuff...
}

// bad
if (collection.length > 0) {
  // ...stuff...
}

// good
if (collection.length) {
  // ...stuff...
```

```
}
```

- For more information see Truth Equality and JavaScript by Angus Croll

Blocks

- Use braces with all multi-line blocks.

```
// bad
if (test)
  return false;

// good
if (test) return false;

// good
if (test) {
  return false;
}

// bad
function() { return false; }

// good
function() {
  return false;
}
```

Comments

- Use `/** ... */` for multiline comments. Include a description, specify types and values for all parameters and return values.

```
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param <String> tag
// @return <Element> element
function make(tag) {

  // ...stuff...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed in tag name
 *
 * @param <String> tag
 * @return <Element> element
 */
function make(tag) {

  // ...stuff...

  return element;
}
```

- Use `//` for single line comments. Place single line comments on a newline above the subject of the comment. Put an empty line before the comment.

```
// bad
var active = true;  // is current tab

// good
// is current tab
var active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  var type = this._type || 'no type';

  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  var type = this._type || 'no type';

  return type;
}
```

- Prefixing your comments with `FIXME` or `TODO` helps other developers quickly understand if you're pointing out a problem that needs to be revisited, or if you're suggesting a solution to the problem that needs to be implemented. These are different than regular comments because they are actionable. The actions are `FIXME -- need to figure this out` or `TODO -- need to implement`.

- Use `// FIXME:` to annotate problems

```
function Calculator() {

  // FIXME: shouldn't use a global here
  total = 0;

  return this;
}
```

- Use `// TODO:` to annotate solutions to problems

```
function Calculator() {

  // TODO: total should be configurable by an options param
  this.total = 0;

  return this;
}
```

Whitespace

- Use soft tabs set to 2 spaces

```
// bad
function() {
····var name;
}

// bad
function() {
·var name;
}
```

```
// good
function() {
··var name;
}
```

- Place 1 space before the leading brace.

```
// bad
function test(){
  console.log('test');
}

// good
function test() {
  console.log('test');
}

// bad
dog.set('attr',{
  age: '1 year',
  breed: 'Bernese Mountain Dog'
});

// good
dog.set('attr', {
  age: '1 year',
  breed: 'Bernese Mountain Dog'
});
```

- Set off operators with spaces.

```
// bad
var x=y+5;

// good
var x = y + 5;
```

- End files with a single newline character.

```
// bad
(function(global) {
  // ...stuff...
})(this);
```

```
// bad
(function(global) {
  // ...stuff...
})(this);↵
↵
```

```
// good
(function(global) {
  // ...stuff...
})(this);↵
```

- Use indentation when making long method chains.

```
// bad
$('#items').find('.selected').highlight().end().find('.open').updateCount();

// good
$('#items')
  .find('.selected')
    .highlight()
```

```
    .end()
  .find('.open')
    .updateCount();

// bad
var leds = stage.selectAll('.led').data(data).enter().append('svg:svg').class('led', true)
    .attr('width',  (radius + margin) * 2).append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin) + ')')
    .call(tron.led);

// good
var leds = stage.selectAll('.led')
    .data(data)
  .enter().append('svg:svg')
    .class('led', true)
    .attr('width',  (radius + margin) * 2)
  .append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius + margin) + ')')
    .call(tron.led);
```

Commas

- Leading commas: **Nope.**

```
// bad
var once
  , upon
  , aTime;

// good
var once,
    upon,
    aTime;

// bad
var hero = {
    firstName: 'Bob'
  , lastName: 'Parr'
  , heroName: 'Mr. Incredible'
  , superPower: 'strength'
};

// good
var hero = {
  firstName: 'Bob',
  lastName: 'Parr',
  heroName: 'Mr. Incredible',
  superPower: 'strength'
};
```

- Additional trailing comma: **Nope.** This can cause problems with IE6/7 and IE9 if it's in quirksmode. Also, in some implementations of ES3 would add length to an array if it had an additional trailing comma. This was clarified in ES5 (source):

> Edition 5 clarifies the fact that a trailing comma at the end of an ArrayInitialiser does not add to the length of the array. This is not a semantic change from Edition 3 but some implementations may have previously misinterpreted this.

```
// bad
var hero = {
  firstName: 'Kevin',
  lastName: 'Flynn',
};

var heroes = [
  'Batman',
  'Superman',
```

```
];

// good
var hero = {
  firstName: 'Kevin',
  lastName: 'Flynn'
};

var heroes = [
  'Batman',
  'Superman'
];
```

Semicolons

- **Yup.**

```
// bad
(function() {
  var name = 'Skywalker'
  return name
})()

// good
(function() {
  var name = 'Skywalker';
  return name;
})();

// good (guards against the function becoming an argument when two files with IIFEs are concatenated)
;(function() {
  var name = 'Skywalker';
  return name;
})();
```

Read more.

Type Casting & Coercion

- Perform type coercion at the beginning of the statement.
- Strings:

```
//  => this.reviewScore = 9;

// bad
var totalScore = this.reviewScore + '';

// good
var totalScore = '' + this.reviewScore;

// bad
var totalScore = '' + this.reviewScore + ' total score';

// good
var totalScore = this.reviewScore + ' total score';
```

- Use `parseInt` for Numbers and always with a radix for type casting.

```
var inputValue = '4';

// bad
var val = new Number(inputValue);
```

```
// bad
var val = +inputValue;

// bad
var val = inputValue >> 0;

// bad
var val = parseInt(inputValue);

// good
var val = Number(inputValue);

// good
var val = parseInt(inputValue, 10);
```

- If for whatever reason you are doing something wild and `parseInt` is your bottleneck and need to use Bitshift for performance reasons, leave a comment explaining why and what you're doing.

```
// good
/**
 * parseInt was the reason my code was slow.
 * Bitshifting the String to coerce it to a
 * Number made it a lot faster.
 */
var val = inputValue >> 0;
```

- **Note:** Be careful when using bitshift operations. Numbers are represented as 64-bit values, but Bitshift operations always return a 32-bit integer (source). Bitshift can lead to unexpected behavior for integer values larger than 32 bits. Discussion. Largest signed 32-bit Int is 2,147,483,647:

```
2147483647 >> 0 //=> 2147483647
2147483648 >> 0 //=> -2147483648
2147483649 >> 0 //=> -2147483647
```

- Booleans:

```
var age = 0;

// bad
var hasAge = new Boolean(age);

// good
var hasAge = Boolean(age);

// good
var hasAge = !!age;
```

Naming Conventions

- Avoid single letter names. Be descriptive with your naming.

```
// bad
function q() {
  // ...stuff...
}

// good
function query() {
  // ..stuff..
}
```

- Use camelCase when naming objects, functions, and instances

```
// bad
var OBJEcttssss = {};
var this_is_my_object = {};
function c() {}
var u = new user({
  name: 'Bob Parr'
});

// good
var thisIsMyObject = {};
function thisIsMyFunction() {}
var user = new User({
  name: 'Bob Parr'
});
```

- Use PascalCase when naming constructors or classes

```
// bad
function user(options) {
  this.name = options.name;
}

var bad = new user({
  name: 'nope'
});

// good
function User(options) {
  this.name = options.name;
}

var good = new User({
  name: 'yup'
});
```

- Use a leading underscore `_` when naming private properties

```
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
this._firstName = 'Panda';
```

- When saving a reference to `this` use `_this`.

```
// bad
function() {
  var self = this;
  return function() {
    console.log(self);
  };
}

// bad
function() {
  var that = this;
  return function() {
    console.log(that);
  };
}

// good
function() {
  var _this = this;
  return function() {
```

```
    console.log(_this);
  };
}
```

- Name your functions. This is helpful for stack traces.

```
// bad
var log = function(msg) {
  console.log(msg);
};

// good
var log = function log(msg) {
  console.log(msg);
};
```

- **Note:** IE8 and below exhibit some quirks with named function expressions. See http://kangax.github.io/nfe/ for more info.

Accessors

- Accessor functions for properties are not required
- If you do make accessor functions use getVal() and setVal('hello')

```
// bad
dragon.age();

// good
dragon.getAge();

// bad
dragon.age(25);

// good
dragon.setAge(25);
```

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

```
// bad
if (!dragon.age()) {
  return false;
}

// good
if (!dragon.hasAge()) {
  return false;
}
```

- It's okay to create get() and set() functions, but be consistent.

```
function Jedi(options) {
  options || (options = {});
  var lightsaber = options.lightsaber || 'blue';
  this.set('lightsaber', lightsaber);
}

Jedi.prototype.set = function(key, val) {
  this[key] = val;
};

Jedi.prototype.get = function(key) {
  return this[key];
};
```

Constructors

- Assign methods to the prototype object, instead of overwriting the prototype with a new object. Overwriting the prototype makes inheritance impossible: by resetting the prototype you'll overwrite the base!

```
function Jedi() {
  console.log('new jedi');
}

// bad
Jedi.prototype = {
  fight: function fight() {
    console.log('fighting');
  },

  block: function block() {
    console.log('blocking');
  }
};

// good
Jedi.prototype.fight = function fight() {
  console.log('fighting');
};

Jedi.prototype.block = function block() {
  console.log('blocking');
};
```

- Methods can return `this` to help with method chaining.

```
// bad
Jedi.prototype.jump = function() {
  this.jumping = true;
  return true;
};

Jedi.prototype.setHeight = function(height) {
  this.height = height;
};

var luke = new Jedi();
luke.jump(); // => true
luke.setHeight(20) // => undefined

// good
Jedi.prototype.jump = function() {
  this.jumping = true;
  return this;
};

Jedi.prototype.setHeight = function(height) {
  this.height = height;
  return this;
};

var luke = new Jedi();

luke.jump()
  .setHeight(20);
```

- It's okay to write a custom toString() method, just make sure it works successfully and causes no side effects.

```
function Jedi(options) {
  options || (options = {});
```

```
  this.name = options.name || 'no name';
}

Jedi.prototype.getName = function getName() {
  return this.name;
};

Jedi.prototype.toString = function toString() {
  return 'Jedi - ' + this.getName();
};
```

Events

- When attaching data payloads to events (whether DOM events or something more proprietary like Backbone events), pass a hash instead of a raw value. This allows a subsequent contributor to add more data to the event payload without finding and updating every handler for the event. For example, instead of:

```
// bad
$(this).trigger('listingUpdated', listing.id);

...

$(this).on('listingUpdated', function(e, listingId) {
  // do something with listingId
});
```

  prefer:

```
// good
$(this).trigger('listingUpdated', { listingId : listing.id });

...

$(this).on('listingUpdated', function(e, data) {
  // do something with data.listingId
});
```

Modules

- The module should start with a `!`. This ensures that if a malformed module forgets to include a final semicolon there aren't errors in production when the scripts get concatenated. Explanation
- The file should be named with camelCase, live in a folder with the same name, and match the name of the single export.
- Add a method called `noConflict()` that sets the exported module to the previous version and returns this one.
- Always declare `'use strict';` at the top of the module.

```
// fancyInput/fancyInput.js

!function(global) {
  'use strict';

  var previousFancyInput = global.FancyInput;

  function FancyInput(options) {
    this.options = options || {};
  }

  FancyInput.noConflict = function noConflict() {
    global.FancyInput = previousFancyInput;
    return FancyInput;
  };

  global.FancyInput = FancyInput;
```

```
}(this);
```

jQuery

- Prefix jQuery object variables with a `$`.

```
// bad
var sidebar = $('.sidebar');

// good
var $sidebar = $('.sidebar');
```

- Cache jQuery lookups.

```
// bad
function setSidebar() {
  $('.sidebar').hide();

  // ...stuff...

  $('.sidebar').css({
    'background-color': 'pink'
  });
}

// good
function setSidebar() {
  var $sidebar = $('.sidebar');
  $sidebar.hide();

  // ...stuff...

  $sidebar.css({
    'background-color': 'pink'
  });
}
```

- For DOM queries use Cascading `$('.sidebar ul')` or parent > child `$('.sidebar > ul')`. jsPerf
- Use `find` with scoped jQuery object queries.

```
// bad
$('ul', '.sidebar').hide();

// bad
$('.sidebar').find('ul').hide();

// good
$('.sidebar ul').hide();

// good
$('.sidebar > ul').hide();

// good
$sidebar.find('ul').hide();
```

ECMAScript 5 Compatibility

- Refer to Kangax's ES5 compatibility table

Testing

- **Yup.**

```
function() {
  return true;
}
```

Performance

- On Layout & Web Performance
- String vs Array Concat
- Try/Catch Cost In a Loop
- Bang Function
- jQuery Find vs Context, Selector
- innerHTML vs textContent for script text
- Long String Concatenation
- Loading...

Resources

Read This

- Annotated ECMAScript 5.1

Tools

- Code Style Linters
  - JSHint - Airbnb Style .jshintrc
  - JSCS - Airbnb Style Preset

Other Styleguides

- Google JavaScript Style Guide
- jQuery Core Style Guidelines
- Principles of Writing Consistent, Idiomatic JavaScript

Other Styles

- Naming this in nested functions - Christian Johansen
- Conditional Callbacks - Ross Allen
- Popular JavaScript Coding Conventions on Github - JeongHoon Byun
- Multiple var statements in JavaScript, not superfluous - Ben Alman

Further Reading

- Understanding JavaScript Closures - Angus Croll
- Basic JavaScript for the impatient programmer - Dr. Axel Rauschmayer
- You Might Not Need jQuery - Zack Bloom & Adam Schwartz
- ES6 Features - Luke Hoban

Books

- JavaScript: The Good Parts - Douglas Crockford
- JavaScript Patterns - Stoyan Stefanov
- Pro JavaScript Design Patterns - Ross Harmes and Dustin Diaz
- High Performance Web Sites: Essential Knowledge for Front-End Engineers - Steve Souders
- Maintainable JavaScript - Nicholas C. Zakas
- JavaScript Web Applications - Alex MacCaw
- Pro JavaScript Techniques - John Resig
- Smashing Node.js: JavaScript Everywhere - Guillermo Rauch
- Secrets of the JavaScript Ninja - John Resig and Bear Bibeault
- Human JavaScript - Henrik Joreteg

- Superhero.js - Kim Joar Bekkelund, Mads Mobæk, & Olav Bjorkoy
- JSBooks - Julien Bouquillon
- Third Party JavaScript - Ben Vinegar and Anton Kovalyov

Blogs

- DailyJS
- JavaScript Weekly
- JavaScript, JavaScript...
- Bocoup Weblog
- Adequately Good
- NCZOnline
- Perfection Kills
- Ben Alman
- Dmitry Baranovskiy
- Dustin Diaz
- nettuts

The JavaScript Style Guide Guide

- Reference

Contributors

- View Contributors

License