JavaScript Coding Standards

Types

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

```
var foo = 1;
var 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];
var 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 = [];
```

- Use Array#push instead of direct assignment to add items to an array.

```
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;
var itemsCopy = [];
var 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 100 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 anything to do with this, you would get nowhere fast.';

// 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;
var messages;
var length;
var 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++) {
    // use direct assignment in this case because we're micro-optimizing.
    items[i] = '<li>' + messages[i].message + '</li>';
  }

  return '<ul>' + items.join('') + '</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 per variable. It's easier to add new variable declarations this way, and you never have to worry about swapping out a `;` for a `,` or introducing punctuation-only diffs.

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

// bad
// (compare to above, and try to spot the mistake)
var items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

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

```
var items = getItems();
var dragonball;
var goSportsTeam = true;
var len;

// good
var items = getItems();
var goSportsTeam = true;
var dragonball;
var length;
var 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 - unnecessary function call
function () {
  var name = getName();
```

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

    this.setFirstName(name);

    return true;
  }

  // good
  function () {
    var name;

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

    name = getName();
    this.setFirstName(name);

    return true;
  }
```

Hoisting

- Variable declarations get hoisted to the top of their scope, but 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');
  }
}
```

Comparison Operators & Equality

- Use `===` and `!==` over `==` and `!=`.
- Conditional statements such as the `if` statement evaluate their expression using coercion with the `ToBoolean` abstract 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...
}
```

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;
}
```

- If you're using multi-line blocks with `if` and `else`, put `else` on the same line as your `if` block's closing brace.

```
// bad
if (test) {
  thing1();
  thing2();
}
else {
  thing3();
}

// good
if (test) {
  thing1();
  thing2();
} else {
```

```
    thing3();
}
```

Comments

- Use `/** ... */` for multi-line 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'
});
```

- Place 1 space before the opening parenthesis in control statements (`if`, `while` etc.). Place no space before the argument list in function calls and declarations.

```
// bad
if(isJedi) {
```

```
  fight ();
}

// good
if (isJedi) {
  fight();
}

// bad
function fight () {
  console.log ('Swooosh!');
}

// good
function fight() {
  console.log('Swooosh!');
}
```

- 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. Use a leading dot, which emphasizes that the line is a method call, not a new statement.

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

```

```
// 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').classed('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')
    .classed('led', true)
    .attr('width', (radius + margin) * 2)
  .append('svg:g')
    .attr('transform', 'translate(' + (radius + margin) + ',' + (radius +
margin) + ')')
    .call(tron.led);
```

- Leave a blank line after blocks and before the next statement

```
// bad
if (foo) {
  return bar;
}
return baz;

// good
if (foo) {
  return bar;
}

return baz;
```

```
// bad
var obj = {
  foo: function () {
  },
  bar: function () {
  }
};
return obj;

// good
var obj = {
  foo: function () {
  },

  bar: function () {
  }
};

return obj;
```

Commas

- Leading commas: **Nope.**

```
// bad
var story = [
    once
  , upon
  , aTime
];

// good
var story = [
  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;
})();
```

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 forperformance 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 = {};
var o = {};
function c() {}

// good
var thisIsMyObject = {};
function thisIsMyFunction() {}
```

- 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.

- If your file exports a single class, your filename should be exactly the name of the class.

```
// file contents
class CheckBox {
  // ...
}
module.exports = CheckBox;

// in some other file
// bad
var CheckBox = require('./checkBox');

// bad
var CheckBox = require('./check_box');

// good
var CheckBox = require('./CheckBox');
```

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 set(key, val) {
  this[key] = val;
};

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

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

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

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

Jedi.prototype.setHeight = function setHeight(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();
```