CODE GUID FOR ES5

## 1. Objects

1.1 Use the literal syntax for object creation.

// bad

var item = new Object();

// good

var item = {};

1.2 Don't use [reserved words](http://es5.github.io/#x7.6.1) as keys.

// bad

var superman = {

default: { clark: 'kent' },

private: true

};

// good

var superman = {

defaults: { clark: 'kent' },

hidden: true

};

## 2. Arrays

2.1 Use the literal syntax for array creation.

// bad

var items = new Array();

// good

var items = [];

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

var someStack = [];

// bad

someStack[someStack.length] = 'abracadabra';

// good

someStack.push('abracadabra');

2.3 When you need to copy an array use Array#slice.( [jsPerf](http://jsperf.com/converting-arguments-to-an-array/7))

var len = items.length;

var itemsCopy = [];

var i;

// bad

for (i = 0; i < len; i++) {

itemsCopy[i] = items[i];

}

// good

itemsCopy = items.slice();

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

function trigger() {

var args = Array.prototype.slice.call(arguments);

...

}

## 3. Strings

3.1 Use single quotes '' for strings.

// bad

var name = "Bob Parr";

// good

var name = 'Bob Parr';

## 4. Functions

4.1 Never name a parameter arguments

// bad

function nope(name, options, arguments) {

// ...stuff...

}

// good

function yup(name, options, args) {

// ...stuff...

}

## 5. Variables

5.1 Always use var to declare variables

// bad

superPower = new SuperPower();

// good

var superPower = new SuperPower();

5.2 Use one var declaration per variable

// bad

var items = getItems(),

goSportsTeam = true,

dragonball = 'z';

// good

var items = getItems();

var goSportsTeam = true;

var dragonball = 'z';

5.3 Declare unassigned variables last

// bad

var i, len, dragonball,

items = getItems(),

goSportsTeam = true;

// good

var items = getItems();

var goSportsTeam = true;

var dragonball;

var length;

var i;

## 6. Comparison Operators & Equality

6.1 Use === and !== over == and !=

6.2 Conditional statements

- **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

6.3 Use shortcuts.

// bad

if (name !== '') {

// ...stuff...

}

// good

if (name) {

// ...stuff...

}

// bad

if (collection.length > 0) {

// ...stuff...

}

// good

if (collection.length) {

// ...stuff...

}

## 7. Blocks

7.1 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;

}

## 8. Comments

8.1 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;

}

8.2 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;

}

8.3 Prefixing your comments with FIXME or TODO

// 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;

}

## 9. Type Casting & Coercion

9.1 Booleans

var age = 0;

// bad

var hasAge = new Boolean(age);

// good

var hasAge = Boolean(age);

// good

var hasAge = !!age;

## 10. Naming Conventions

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

// bad

function q() {

// ...stuff...

}

// good

function query() {

// ..stuff..

}

10.2 Use camelCase when naming objects, functions, and instances

// bad

var OBJEcttsssss = {};

var this\_is\_my\_object = {};

var o = {};

function c() {}

// good

var thisIsMyObject = {};

function thisIsMyFunction() {}

10.3 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'

});

10.4 Use a leading underscore \_ when naming private properties

// bad

this.\_\_firstName\_\_ = 'Panda';

this.firstName\_ = 'Panda';

// good

this.\_firstName = 'Panda';

10.5 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);

};

}

10.6 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);

};

## 11. Accessors

11.1 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);

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

// bad

if (!dragon.age()) {

return false;

}

// good

if (!dragon.hasAge()) {

return false;

}

## 12. Constructors

## 12.1 Assign methods to the prototype object, instead of overwriting the prototype with a new object

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');

};

## 12.2 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);

## 13. Events

13.1 attaching data payloads to events

// bad

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

$(this).on('listingUpdated', function(e, listingId) {

// do something with listingId

// good

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

$(this).on('listingUpdated', function(e, data) {

// do something with data.listingId

});