readme.md

JavaScript Coding Standards

Tabla de Contenido

Basado en Airbnb JavaScript Style Guide

Sintaxis y formato

A manera general queremos:

- Utilizamos ECMAScript 5.1 (ES5) - Annotated ECMAScript 5.1
- Indentar con **2** espacios, no tabs.
- Deja un espacio antes de la llave de apertura `{`.
- Usar llaves con todos los bloques de múltiples líneas (`if`, `function`).
- Un espacio antes del paréntesis de apertura en las sentencias de control (`if`, `while`, etc.).
- Bloques de muchas líneas con `if` y `else`, poner el `else` en la misma línea que el `if`.

- No espacio entre el keyword function o el nombrede la funcion y el primer parentesis (`function() {}`)
- Separar a los operadores con espacios. `var x = y + 5;`
- Deja una línea en blanco luego de los bloques y antes de la siguiente sentencia.
- Ser descriptivo con nombres de variables, métodos, funciones, etc.
- camelCase para nombrar objetos, funciones e instancias, ejem: `thisIsMyObject` , `thisIsMyFunction`
- PascalCase cuando nombre constructores o clases.
- Un guión bajo (_) adelante de la variable cuando se nombre propiedades privadas, ejem: `var __firstName`
- Cuando se guarde una referencia a `this` usa `_this`
- No usar palabras reservadas para nombres de propiedades, funciones o variables.
- Nombre sus funciones. Esto será de ayuda en caso de errores.

Puntuacion: comas, puntos y coma

- **No** usar comas al inicio de línea:

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

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

- **Sip.**

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

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

// super bien (evita que la funcion se vuelva un argumento
// cuando dos archivos con IIFEs sean concatenados)
;(function() {
  var name = 'Skywalker';
  return name;
})();
```

Tipos

- **Primitivos**: Cuando accesas a un tipo primitivo, manejas directamente su valor
  - `string`
  - `number`
  - `boolean`
  - `null`

    - undefined

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

bar = 9;

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

- **Complejo**: Cuando accesas a un tipo complejo, manejas la referencia a su valor.

    - object
    - array
    - function

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

bar[0] = 9;

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

[⬆ regresar a la Tabla de Contenido]

Objetos

- Usa la sintaxis literal para la creación de un objeto.

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

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

[⬆ regresar a la Tabla de Contenido]

Arreglos

- Usa la sintaxis literal para la creación de arreglos

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

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

- Usa Array#push, en vez de asignación directa, para agregar elementos a un arreglo.

```
var someStack = [];

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

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

- Cuando necesites copiar un arreglo usa Array#slice. jsPerf

```
var len = items.length;
```

```
var itemsCopy = [];
var i;

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

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

- Para convertir un objeto "array-like" (similar a un arreglo) a un arreglo, usa Array#slice.

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

[⬆ regresar a la Tabla de Contenido]

Cadenas de Texto

- Usa comillas simples `''` para las cadenas de texto

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

// bien
var name = 'Bob Parr';

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

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

- Las cadenas de texto con una longitud mayor a 100 caracteres deben ser escritas en múltiples líneas usando concatenación.
- Cuando se crea programáticamente una cadena de texto, use Array#join en vez de concatenación. Sobretodo por 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;

// mal
function inbox(messages) {
```

```
  items = '<ul>';

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

  return items + '</ul>';
}

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

  for (i = 0; i < length; i++) {
    // usa asignacion directa aqui porque estamos micro-optimizando
    items[i] = '<li>' + messages[i].message + '</li>';
  }

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

[⬆ regresar a la Tabla de Contenido]

Funciones

- Expresiones de función:

```
// expresion de funcion anonima
var anonymous = function() {
  return true;
};

// expresion de funcion nombrada
var named = function named() {
  return true;
};

// expresion de funcion inmediatamente invocada (IIFE)
(function() {
  console.log('Welcome to the Internet. Please follow me.');
})();
```

- Nunca declares una función en un bloque que no sea de función (if, while, etc). En vez de ello, asigna la función a una variable. Los navegadores te permitirán hacerlo pero todos ellos lo interpretarán de modo diferente, lo que es lamentable.
- Nunca nombres a un parámetro como `arguments`, esto tendrá precedencia sobre el objeto `arguments` que es brindado en cada ámbito de función.

[⬆ regresar a la Tabla de Contenido]

Propiedades

- Usa la notación de punto `.` cuando accedas a las propiedades.

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

// mal
var isJedi = luke['jedi'];
```

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

- Usa la notación subscript `[]` cuando accedas a las propiedades con una variable.

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

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

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

[⬆ regresar a la Tabla de Contenido]

Variables

- Siempre usar `var` para declarar variables. No hacerlo resultará en variables globales. Debemos evitar contaminar el espacio global (global namespace).
- Usar una declaración `var` por variable.

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

- Declarar a las variables sin asignación al final. Esto es útil cuando necesites asignar una variable luego dependiendo de una de las variables asignadas previamente.

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

- Asigna las variables al inicio de su ámbito. Esto ayuda a evitar inconvenientes con la declaración de variables y temas relacionados a 'hoisting'.

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

  //..otras cosas..

  var name = getName();

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

  return name;
}

// bien
function() {
```

```
  var name = getName();

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

  //..otras cosas..

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

  return name;
}

// mal - llamada a funcion innecesaria
function() {
  var name = getName();

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

  this.setFirstName(name);

  return true;
}

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

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

  return true;
}
```

[⬆ regresar a la Tabla de Contenido]

Hoisting

- Las declaraciones de variables son movidas a la parte superior de su ámbito, sin embargo su asignación no.

```
// sabemos que esto no funcionara (asumiendo
// que no hay una variable global notDefined)
function example() {
  console.log(notDefined); // => lanza un ReferenceError
}

// crear una declaracion de variable luego
// que referencies a la variable funcionara
// por el hoisting. Nota: A la asignacion
// del valor `true` no se le aplico hoisting.
function example() {
  console.log(declaredButNotAssigned); // => undefined
  var declaredButNotAssigned = true;
}

// El interprete lleva la declaracion de la
// variable a la parte superior de la funcion.
// Eso significa que nuestro ejemplo
// podria ser reescrito como:
function example() {
  var declaredButNotAssigned;
```

```
  console.log(declaredButNotAssigned); // => undefined
  declaredButNotAssigned = true;
}
```

- Expresiones de función anónimas hacen hoisting de su nombre de variable, pero no de la asignación de la función.

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

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

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

- Expresiones de función nombradas hacen hoisting de su nombre de variable, pero no del nombre de la función ni del contenido de la función.

```
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');
  };
}

// lo mismo es cierto cuando el nombre
// de la funcion es igual al nombre de
// la variable.
function example() {
  console.log(named); // => undefined

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

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

- Las declaraciones de función hacen hoist de su nombre y del contenido de la función.

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

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

[⬆ regresar a la Tabla de Contenido]

Expresiones de comparación e igualdad

- Usa `===` y `!==` en vez de `==` y `!=` respectivamente.
- Expresiones condicionales son evaluadas usando coerción con el método `ToBoolean` y siempre obedecen a estas reglas sencillas:

  - **Objects** son evaluados como **true** (también considera así al objeto vacío `{}` y arreglos sin contenido `[]`)
  - **Undefined** es evaluado como **false**
  - **Null** es evaluado como **false**
  - **Booleans** son evaluados como **el valor del booleano**
  - **Numbers** son evaluados como **false** si **+0**, **-0**, o **NaN**, de otro modo **true**
  - **Strings** son evaluados como **false** si es una cadena de texto vacía `''`, de otro modo son **true**

```
if ([0]) {
  // true
  // un arreglo es un objeto, los objetos son evaluados como true
}
```

- Usa atajos.

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

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

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

// bien
if (collection.length) {
  // ...stuff...
}
```

[⬆ regresar a la Tabla de Contenido]

Comentarios

- Usa `/** ... */` para comentarios de múltiples líneas. Incluye una descripción, especificación de tipos y valores para todos los parámetros y valores de retorno.

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

  // ...stuff...

  return element;
}
```

- Usa `//` para comentarios de una sola línea. Ubica los comentarios de una sola línea encima de la sentencia comentada. Deja una línea en blanco antes del comentario.

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

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

  return type;
}
```

[⬆ regresar a la Tabla de Contenido]

Casting de Tipos & Coerción

- Ejecuta coerción al inicio de una sentencia.
- Strings:

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

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

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

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

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

- Usa `parseInt` para números y siempre con la base numérica para el casting de tipo.

```
var inputValue = '4';

// mal
var val = new Number(inputValue);

// mal
var val = +inputValue;

// mal
var val = inputValue >> 0;

// mal
var val = parseInt(inputValue);

// bien
var val = Number(inputValue);

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

- Booleans:

```
var age = 0;

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

// bien
var hasAge = Boolean(age);

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

[⬆ regresar a la Tabla de Contenido]

Funciones de Acceso

- Funciones de acceso para las propiedades no son requeridas, pero sí se crean, usar `getVal()` y `setVal('hello')`. Ejem: `getAge()`, `setAge(25)`.
- Si la propiedad es un booleano, usa `isVal()` o `hasVal()`.

[⬆ regresar a la Tabla de Contenido]

Constructores

- Asigna métodos al objeto prototype, en vez de sobreescribir prototype con un nuevo objeto. La sobreescritura de prototype hace la herencia imposible: ¡reseteando prototype sobreescribirás la base!
- Métodos pueden retornar `this` para ayudar con el encadenamiento de métodos (chaining).

```
// mal
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

// bien
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);
```

[⬆ regresar a la Tabla de Contenido]

Eventos

- Cuando envíes paquetes de datos a los eventos (ya sea con eventos del DOM o algo propietario como los eventos de Backbone), pasa un mapa en vez de un valor directo. Esto permitirá a un próximo colaborador a agregar más datos al paquete de datos sin que tenga que encontrar o actualizar un handler para cada evento. Por ejemplo, en vez de:

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

...
```

```
$(this).on('listingUpdated', function(e, listingId) {
  // hacer algo con listingId
});
```

prefiere:

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

...

$(this).on('listingUpdated', function(e, data) {
  // hacer algo con data.listingId
});
```

[⬆ regresar a la Tabla de Contenido]

Módulos

- El módulo debe empezar con un `!`. Esto asegura que si un módulo mal formado olvide incluir al final un punto y coma, no hayan errores en producción cuando los scripts sean concatenados. Explicación
- El archivo debe ser nombrado con camelCase, residir en un fólder con el mismo nombre, y corresponder al nombre de la función a exportar.
- Agrega un método `noConflict()` que reestablezca el módulo exportado a la versión anterior y retorne este módulo (para ser asignado a una variable).
- Siempre declara `'use strict';` al inicio de cada módulo.

```
// 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);
```

[⬆ regresar a la Tabla de Contenido]

Recursos

- JavaScript | MDN
- Truth Equality and JavaScript
- JavaScript Scoping & Hoisting
- Google JavaScript Style Guide (Guía de Estilo de Javascript de Google)
- jQuery Core Style Guidelines (Lineamientos de Estilo con el núcleo de jQuery)
- Principles of Writing Consistent, Idiomatic JavaScript (Idiomatic Javascript: Principios de Escritura Consistente)
- JavaScript Standard Style (Estilo estándar de JavaScript)

- Airbnb JavaScript Style Guide

Licencia

[⬆ regresar a la Tabla de Contenido]

};