Principios para escribir JavaScript consistente e idiomático

> Todo el código en cualquier proyecto debería verse como si una sola persona lo hubiera escrito, sin importar cuánta gente haya contribuído. La lista que se presenta a continuación destaca las prácticas que se deben seguir en todo el código.

Tabla de contenidos

Prólogo

Las siguientes secciones delinean una guía de estilos *razonable* para desarrollo moderno de JavaScript, y no intentan ser prescriptivas. Lo más importante es la **ley de consistencia de estilo de código**. Cualquiera sea el estilo que escojas para tu proyecto debería ser considerado ley. Piensa en este documento como una declaración del compromiso para con la consistencia de estilo, legibilidad y mantenibilidad.

1. Espacios en blanco

- Nunca mezclar espacios y tabulaciones.
- Al comenzar un proyecto, antes de escribir código, escoger entre indentación blanda (espacios), o tabulaciones — Esto es **LEY**. * Para mayor legibilidad, siempre recomiendo ajustar las preferencias de tu editor para que el tamaño de la indentación sea de dos caracteres — esto significa, usar dos espacios, o que dos espacios representen una tabulación.
- Si tu editor lo soporta, trabajar siempre con la preferencia activada para que se muestren los caracteres invisibles. Los beneficios de esta práctica son: * Reforzar la consistencia. * Eliminar el espacio en blanco del fin de línea. * Eliminar los espacios en blanco de las líneas vacías. * Commits y diffs más fáciles de leer.

2. Sintaxis elegante

A. Paréntesis, Llaves, Fines de línea

if/else/for/while/try siempre tienen espacios y se extienden a múltiples líneas.
Esto mejora la legibilidad

Ejemplos de sintaxis toda comprimida:

```
if(condition) doSomething();

while(condition) iterating++;

for(var i=0;i<100;i++) someIterativeFn();
```

Uso de espacios para mejorar la legibilidad

```
if ( condition ) {
    // statements
}

while ( condition ) {
    // statements
}

for ( var i = 0; i < 100; i++ ) {
    // statements
}
```

Aún mejor:

```
var i,
    length = 100;

for ( i = 0; i < length; i++ ) {
    // statements
}
```

O...

```
var i = 0,
    length = 100;

for ( ; i < length; i++ ) {
    // statements
}

var prop;

for ( prop in object ) {
    // statements
}


if ( true ) {
```

```
    // statements
} else {
    // statements
}
```

B. Asignaciones, Declaraciones, Funciones (Nombradas, Expresiones, Constructores)

Variables

```
var foo = "bar",
    num = 1,
    undef;
```

Notaciones literales

```
var array = [],
    object = {};
```

Usando solo una instancia de var por scope (o sea, dentro de la función), mejora la legibilidad y mantiene tu lista de declaraciones claras (también ahorra de tipear un poco más)

Mal

```
var foo = "";
var bar = "";
var qux;
```

Bien

```
var foo = "",
    bar = "",
    quux;
```

Bien

```
var // comentar en éstos
foo = "",
bar = "",
quux;
```

Sentencias var deberían estar siempre al principio de su respectivo scope (alcance), que sería la función.

Mal

```
function foo() {

  // algunas sentencias

  var bar = "",
    qux;
}
```

Bien

```
function foo() {
  var bar = "",
    qux;

  // todas las sentencias luego de la declaración de variables.
}
```

const y let, de ECMAScript 6, de la misma manera deberían aparecer al principio de su scope (alcance), que sería el bloque.

Mal

```
function foo() {
  let foo,
    bar;
  if (condition) {
    bar = "";
    // sentencias
  }
}
```

Bien

```
function foo() {
  let foo;
  if (condition) {
    let bar = "";
    // sentencias
  }
}
```

Declaración De Función Nombrada

```
function foo( arg1, argN ) {

}

// Uso
foo( arg1, argN );
```

Declaración De Función Nombrada

```
function square( number ) {
    return number * number;
}

// Uso
square( 10 );

// Really contrived continuation passing style
function square( number, callback ) {
    callback( number * number );
}
```

```
square( 10, function( square ) {
    // callback statements
});
```

Expresión de función

```
var square = function( number ) {
    // Retornar algo relevante y con valor agregado
    return number * number;
};
```

Expresión de función con identificador
Esta forma es preferida porque tiene el valor agregado de poder ser llamada a sí misma y ser identificable en el seguimiento de la pila (MUY útil para debugging)

```
var factorial = function factorial( number ) {
    if ( number < 2 ) {
        return 1;
    }

    return number * factorial( number-1 );
};
```

Declaración de Constructor

```
function FooBar( options ) {

    this.options = options;
}

// Uso
var fooBar = new FooBar({ a: "alpha" });

fooBar.options;
// { a: "alpha" }
```

C. Algunas excepciones

Funciones con callbacks

```
foo(function() {
    // Como se ve no hay espacio entre el primer paréntesis
    // y la palabra "function"
});
```

Función que acepta un Array como parámetro, sin espacio entre ([

```
foo([ "alpha", "beta" ]);
```

Función que acepta un objeto, sin espacio

```
    foo({
        a: "alpha",
        b: "beta"
    });
```

Argumento único de tipo string, sin espacio

```
    foo("bar");

    // Grupo de paréntesis dentro de paréntesis, sin espacio
    if ( !("foo" in obj) ) {

    }
```

D. La consistencia siempre gana

```
En las secciones 2.A-2.C, las reglas para los espacios son puestas con un objetivo más simple y con
un propósito más general: consistencia.
Es importante destacar que algunas preferencias de formato, deberían ser consideradas opcionales,
pero solo un estilo debería existir a través de todo el código de fuente de tu proyecto.
```

2.D.1.1

```
    if (condition) {
        // sentencias
    }

    while (condition) {
        // sentencias
    }

    for (var i = 0; i < 100; i++) {
        // sentencias
    }

    if (true) {
        // sentencias
    } else {
        // sentencias
    }
```

E. Comillas

Si prefieres comillas simples o dobles no debería importar, no hay diferencias en como JavaScript las parsea. Lo que **ABSOLUTAMENTE DEBE** ser cumplido es la consistencia. **Nunca mezclar comillas en el mismo proyecto. Elegir un estilo y cumplirlo.**

F. Fin de línea y líneas vacías

Espacios en blanco pueden arruinar diffs y hacer los cambios imposibles de leer. Considera agregar algún mecanismo para remover automáticamente los espacios que se encuentran al final de la línea o en líneas vacías.

3. Checkeo de tipos (Cortesía de la guía de estilo de jQuery)

A. Tipos

- String:

```
typeof variable === "string"
```

- Number:

```
typeof variable === "number"
```

- Boolean:

```
typeof variable === "boolean"
```

- Object:

```
typeof variable === "object"
```

- Array:

```
Array.isArray( arrayLikeObject )
(cuando sea posible / hay implementaciones que no tienen esta función)
```

- Node:

```
elem.nodeType === 1
```

- null:

```
variable === null
```

- null o undefined:

```
variable == null
```

- undefined:

```
Variables globales:

  typeof variable === "undefined"

Variables locales:
```

```
  variable === undefined

Propiedades:

  object.prop === undefined
  object.hasOwnProperty( prop )
  "prop" in object
```

B. Conversiones implícitas de tipos

Considera lo que causaría lo siguiente...

Dado este HTML:

```
```html

<input type="text" id="foo-input" value="1">

```
```

3.B.1.1 foo ha sido declarado con el valor 0 y su tipo es number ``` var foo = 0;

```
// typeof foo;
// "number"
...

// En algún lugar, más tarde en tu código, necesitas modificar `foo`
// con un nuevo valor derivado de el elemento input del HTML

foo = document.getElementById("foo-input").value;

// Si vas a testear `typeof foo` ahora, el resultado sería `string`
// Esto significa que si hubieras tenido lógica que comparara `foo` así:

if ( foo === 1 ) {

    importantTask();

}

// `importantTask()` nunca habría sido evaluada, incluso con `foo` teniendo un valor de "1"
```

```
3.B.1.2 Te puedes adelantar a los problemas, usando conversión de tipos con los operadores unarios +
o - :
```

```
foo = +document.getElementById("foo-input").value;
//      ^ el operador unario + va a convertir su operando derecho a number

// typeof foo;
// "number"

if ( foo === 1 ) {
```

```
    importantTask();

}

// `importantTask()` va a ser llamada
```

Aquí hay algunos casos comunes de conversiones implícitas:

3.B.2.1

```
var number = 1,
    string = "1",
    bool = false;

number;
// 1

number + "";
// "1"

string;
 // "1"

+string;
// 1

+string++;
// 1

string;
// 2

bool;
// false

+bool;
// 0

bool + "";
// "false"
```

3.B.2.2

```
var number = 1,
    string = "1",
    bool = true;

string === number;
// false

string === number + "";
// true
```

```
+string === number;
// true

bool === number;
// false

+bool === number;
// true

bool === string;
// false

bool === !!string;
// true
```

3.B.2.3

```
var array = [ "a", "b", "c" ];

!!~array.indexOf( "a" );
// true

!!~array.indexOf( "b" );
// true

!!~array.indexOf( "c" );
// true

!!~array.indexOf( "d" );
// false

// Nótese que el ejemplo anterior debería ser considerado "innecesariamente inteligente"
// Prefiérase el approach obvio de comparar el valor retornado de
// indexOf, como en:

if ( array.indexOf( "a" ) >= 0 ) {
  // ...
}
```

3.B.2.4

```
var num = 2.5;

parseInt( num, 10 );

// es lo mismo que...

~~num;

num >> 0;
```

```
num >>> 0;

// Todos resultan en 2


// Ten en cuenta que números negativos serán tratados de otra manera...

var neg = -2.5;

parseInt( neg, 10 );

// es lo mismo que...

~~neg;

neg >> 0;

// Todos resultan en -2
// Sin embargo...

neg >>> 0;

// Va a resultar en 4294967294
```

4. Evaluación condicional

4.1.1 Cuando evaluamos para saber si un array tiene algún elemento, en lugar de hacer esto:

```
if ( array.length > 0 ) ...
```

...evalua por el valor de verdad de la expresión, como en este caso:

```
if ( array.length ) ...
```

4.1.2 Cuando solo evaluamos que un array está vacío, en lugar de esto:

```
if ( array.length === 0 ) ...
```

...evalua el valor de verdad de la expresión:

```
if ( !array.length ) ...
```

4.1.3 Cuando evaluamos que un string no es vacío, en vez de hacer esto:

```
if ( string !== "" ) ...
```

...evalua el valor de verdad de la expresión:

```
if ( string ) ...
```

4.1.4 Cuando evaluamos si un string es vacío, en lugar de:

```
if ( string === "" ) ...
```

...evalua la expresión por falso, como aquí:

```
if ( !string ) ...
```

4.1.5 Cuando evaluamos si una variable es true, en lugar de esto:

```
if ( foo === true ) ...
```

...expresalo directamente de la siguiente manera:

```
if ( foo ) ...
```

4.1.6 Cuando evaluamos si una variable es false, en vez de:

```
if ( foo === false ) ...
```

```
...usa negación y expresalo de esta manera
```

```
if ( !foo ) ...
```

`` Ten cuidado, porque la anterior expresión va a ser true también para: 0, "", null, undefined, NaN.
Si se *DEBE* testear por el valor false únicamente, utilizar el ===

```
    if ( foo === false ) ...
```

4.1.7 Cuando solamente se esta evaluando una referencia que puede ser null o undefined, pero NO false, "" or 0, en lugar de:

```
    if ( foo === null || foo === undefined ) ...
```

...toma ventaja del uso del operador ==: if (foo == null) ...

```
// Recuerda, usar == va a comparar `null` con AMBOS `null` y `undefined`
// pero no `false`, "" o 0
null == undefined

```
```

SIEMPRE evalua por el mejor, más preciso resultado - lo de arriba es sólo una guía, no un dogma.

4.2.1 Sobre los tipos y evaluación de expresiones

Usa === y no == (a menos que el caso particular requiera una evaluación no fuertemente tipada) ya que `===' no genera coerción de tipos, lo que significa que:

```
    "1" === 1;
    // false
```

```
`==` genera coerción de tipos, lo que significa que:
```

```
    "1" == 1;
    // true
```

4.2.2 Booleanos, Verdaderos y Falsos

```
    //Booleanos:
    true, false
```
```

```
//Verdaderos:
"foo", 1

//Falsos:
"", 0, null, undefined, NaN, void 0
```

5. Estilo práctico

5.1.1 Un módulo práctico

```
(function( global ) {
    var Module = (function() {

        var data = "secret";

        return {
            // Esta es una propiedad booleana
            bool: true,
            // Algún valor string
            string: "a string",
            // Una propiedad Array
            array: [ 1, 2, 3, 4 ],
            // Una propiedad objeto
            object: {
                lang: "es-AR"
            },
            getData: function() {
                // Obtener el valor de `data`
                return data;
            },
            setData: function( value ) {
                // asigna el valor de `data` y lo retorna
                return ( data = value );
            }
        };
    })();

    // Otras cosas podrían pasar aquí

    // Exponer nuestro módulo al módulo global
    global.Module = Module;

})( this );
```

5.2.1 Un constructor práctico

```
(function( global ) {

    function Ctor( foo ) {

        this.foo = foo;

        return this;
    }

    Ctor.prototype.getFoo = function() {
        return this.foo;
```

```
    };

    Ctor.prototype.setFoo = function( val ) {
        return ( this.foo = val );
    };


    // para llamar al constructor sin usar `new`, se podría hacer de la siguiente manera:
    var ctor = function( foo ) {
        return new Ctor( foo );
    };


    // Exponer nuestro constructor al objeto global
    global.ctor = ctor;

})( this );
```

6. Naming

Si no eres un compilador/compresor humano de código, no te esfuerces por de serlo.

A continuación ejemplos "tristes" de nombramiento de variables

6.1.1 Ejemplo de código con nombres de variables pobres / poco descriptivos

```
function q(s) {
    return document.querySelectorAll(s);
}
var i,a=[],els=q("#foo");
for(i=0;i<els.length;i++){a.push(els[i]);}
```

Seguramente, alguna vez hayas escrito código como éste. Con un poco de suerte dejarás de hacerlo hoy mismo.

Aquí está el mismo fragmento, pero con un nombramiento de variables más inteligente (y una estructura más legible):

6.A.2.1 Ejemplo de código con nombres mejorados

```
function query( selector ) {
    return document.querySelectorAll( selector );
}

var idx = 0,
    elements = [],
    matches = query("#foo"),
    length = matches.length;

for( ; idx < length; idx++ ){
    elements.push( matches[ idx ] );
}
```

Algunos tips más para nombramiento de variables:

6.A.3.1 Nombrando strings

```
    `dog` es un string
```

6.A.3.2 Nombrando arrays

```
    `dogs` es un array de `dog` strings
```

6.A.3.3 Nombrando funciones, objetos, instancias, etc.

```
camelCase; función y declaración de variable
```

6.A.3.4 Nombrando constructores, prototypes, clases, etc.

```
PascalCase; función constructora
```

6.A.3.5 Nombrando expresiones regulares

```
rDesc = //;
```

6.A.3.6 Extraído de la guía de estilo de la Google Closure Library

```
    functionNamesLikeThis;
    variableNamesLikeThis;
    ConstructorNamesLikeThis;
    EnumNamesLikeThis;
    methodNamesLikeThis;
    SYMBOLIC_CONSTANTS_LIKE_THIS;
```

B. Caras de this

Mas allá de los generalmente bien conocidos casos de uso de call y apply, siempre preferir .bind(this) o equivalente, para crear definiciones de BoundFunction para invocar más tarde. Sólo recurrir a alias cuando no hay disponible una opción preferible.

6.B.1

```
    function Device( opts ) {

      this.value = null;

      // abre un stream async,
      // esto va a ser llamado continuamente
      stream.read( opts.path, function( data ) {

        // Actualizar el valor de la instancia
        // con el valor mas reciente del
        // data stream
        this.value = data;

      }.bind(this) );

      // Regular la frecueancia de eventos emitidos de
```

```
  // esta instancia de Device
  setInterval(function() {

    // Emitir un evento regulado
    this.emit("event");

  }.bind(this), opts.freq || 100 );
}

// Sólo hagamos de cuenta que heredamos de EventEmitter ;)
```

Cuando no esté disponible, equivalentes funcionales a .bind existen en muchas bibliotecas de JavaScript modernas.

6.B.2 ejemplo con lodash/underscore, _.bind()

```
function Device( opts ) {

  this.value = null;

  stream.read( opts.path, _.bind(function( data ) {

    this.value = data;

  }, this) );

  setInterval(_.bind(function() {

    this.emit("event");

  }, this), opts.freq || 100 );
}

// ejemplo con jQuery.proxy
function Device( opts ) {

  this.value = null;

  stream.read( opts.path, jQuery.proxy(function( data ) {

    this.value = data;

  }, this) );

  setInterval( jQuery.proxy(function() {

    this.emit("event");

  }, this), opts.freq || 100 );
}

// ejemplo con dojo.hitch
function Device( opts ) {

  this.value = null;

  stream.read( opts.path, dojo.hitch( this, function( data ) {

    this.value = data;

  }) );

  setInterval( dojo.hitch( this, function() {
```

```
        this.emit("event");

      }), opts.freq || 100 );
    }
```

Como última opción, crear un alias a this usando self como identificador. Esto es extremadamente sujeto a fallas y debería ser evitado cuando sea posible.

6.B.3

```
    function Device( opts ) {
      var self = this;

      this.value = null;

      stream.read( opts.path, function( data ) {

        self.value = data;

      });

      setInterval(function() {

        self.emit("event");

      }, opts.freq || 100 );
    }
```

C. Usar thisArg

Varios prototype methods de ES 5.1 built-ins vienen con una firma especial thisArg, la cual debería ser usada cuando sea posible

6.C.1

```
    var obj;

    obj = { f: "foo", b: "bar", q: "qux" };

    Object.keys( obj ).forEach(function( key ) {

      // |this| now refers to `obj`

      console.log( this[ key ] );

    }, obj ); // <-- el último argumento es `thisArg`

    // Imprime...

    // "foo"
    // "bar"
    // "qux"

    ```
`thisArg` puede ser usado con `Array.prototype.every`, `Array.prototype.forEach`,
`Array.prototype.some`, `Array.prototype.map`, `Array.prototype.filter`

7. Varios
```
```

Esta sección servirá para describir ideas y conceptos que no deberían se considerados dogmas, pero existen para alentar a cuestionarse prácticas, en un intento para encontrar mejores maneras de llevar a cabo tareas comunes de programación en JavaScript.

A. El uso de `switch` debería ser evitado, ya que los métodos modernos de tracing marcarán como negativas las funciones que contengan sentencias `switch`.

Parecen haber mejoras drásticas en la ejecución de sentencias `switch` en las últimas versiones de Firefox y Chrome.
http://jsperf.com/switch-vs-object-literal-vs-module

Mejoras destacables pueden ser vistas aquí también:
https://github.com/rwldrn/idiomatic.js/issues/13

7.A.1.1 Ejemplo de una sentencia switch

```
switch( foo ) {
    case "alpha":
        alpha();
        break;
    case "beta":
        beta();
        break;
    default:
        // hacer algo por defecto
        break;
}
```

7.A.1.2 Una manera alternativa que soporta componibilidad y reusabilidad es usar un objeto para guardad "cases" y una función para delegar:

```
var cases, delegator;

// returns de ejemplo sólo para demostración.
cases = {
  alpha: function() {
    // sentencias
    // un return
    return [ "Alpha", arguments.length ];
  },
  beta: function() {
    // sentencias
    // un return
    return [ "Beta", arguments.length ];
  },
  _default: function() {
    // sentencias
    // un return
    return [ "Default", arguments.length ];
  }
};

delegator = function() {
  var args, key, delegate;

  // Transforma la lista de argumentos en un array
  args = [].slice.call( arguments );
```

```
  // Obtiene la llave del "case" del array
  key = args.shift();

  // Asigna el handler por defecto para el "case"
  delegate = cases._default;

  // Deriva el método a la operación delegada
  if ( cases.hasOwnProperty( key ) ) {
    delegate = cases[ key ];
  }

  // El argumento de alcance podría ser asignado a algo específico,
  // en este caso, |null| va a ser suficiente
  return delegate.apply( null, args );
};
```

7.A.1.3 Pon la API de 7.A.1.2 a trabajar:

```
delegator( "alpha", 1, 2, 3, 4, 5 );
// [ "Alpha", 5 ]

// Por supuesto, el argumento de la llave del `case` podría ser basado en alguna otra condición
arbitraria.

var caseKey, someUserInput;

// Possibly some kind of form input?
someUserInput = 9;

if ( someUserInput > 10 ) {
  caseKey = "alpha";
} else {
  caseKey = "beta";
}
```

o...

```
caseKey = someUserInput > 10 ? "alpha" : "beta";
```

Y luego...

```
delegator( caseKey, someUserInput );
// [ "Beta", 1 ]
```

Y por supuesto...

```
    delegator();
    // [ "Default", 0 ]
```

B. Hacer un return temprano mejora la legibilidad del código y no tiene un impacto significativo en el rendimiento

7.B.1.1 Mal:

```
function returnLate( foo ) {
    var ret;

    if ( foo ) {
        ret = "foo";
    } else {
        ret = "quux";
    }
    return ret;
}
```

Bien:

```
function returnEarly( foo ) {

    if ( foo ) {
        return "foo";
    }
    return "quux";
}
```

8. Native & Host Objects

El principio fundamental aquí es:

- No hagas cosas estúpidas y todo irá bien.

Para reforzar este concepto, mirar esta presentación:

- "Everything is Permitted: Extending Built-ins" by Andrew Dupont (JSConf2011, Portland, Oregon) - recomendada!

http://www.everytalk.tv/talks/441-JSConf-Everything-is-Permitted-Extending-Built-ins

9. Comentarios

1. Comentar en la línea que está justo arriba del código del que se está hablando
2. Comentarios en múltiples líneas son buenos
3. Comentarios en el fin de línea están prohibidos!
4. El estilo de JSDoc es bueno, pero requiere una inversión de tiempo significativa

Cosas importantes, no relacionadas directamente con el JS idiomático:

Calidad de código: herramientas, recursos y referencias

- Plug-in de JavaScript para Sonar
- Plato
- jsPerf
- jsFiddle
- jsbin
- JavaScript Lint (JSL)
- jshint
- jslint
- Editorconfig

Conociendo mejor el lenguaje

Annotated ECMAScript 5.1

EcmaScript Language Specification, 5.1 Edition

Los siguientes artículos son 1) incompletos y 2) *OBLIGATORIOS*. No siempre estoy de acuerdo con el estilo escrito por sus autores, pero una cosa es cierta: son consistentes. Además, son autoridades en el lenguaje.

- Baseline For Front End Developers
- Eloquent JavaScript
- JavaScript, JavaScript
- Adventures in JavaScript Development
- Perfection Kills
- Douglas Crockford's Wrrrld Wide Web
- JS Assessment (Todos estos artículos están en inglés)

Proceso de Build y Deployment

Los proyectos deberían tratar de incluir siempre algún mecanismo para que el código pueda ser verificado, comprimido y optimizado para su uso en producción. Para esta tarea, grunt por Ben Alman es la primera en ganar una gran popularidad, y ha reemplazado oficialmente la carpeta "kits/" que había en este repo.

Testing

Los proyectos *deben* incluir alguna forma de testing (test unitario, test funcional, etc). Las demos NO CUENTAN como "tests". A continuación, una lista de frameworks para testing, ninguno de los cuales recomiendo más que otro.

- QUnit
- Jasmine
- Vows
- Mocha
- Hiro
- JsTestDriver
- Buster.js
- Sinon.js