# Introducción

## Introducción

El Jardín de JavaScript es una guía de documentación acerca de las partes más peculiares de este lenguaje de programación. Brinda consejos para evitar los errores más comunes y sutiles, así como problemas de rendimiento y de malas prácticas que los programadores menos experimentados en JavaScript pueden resolver en sus esfuerzos por profundizar en el lenguaje.

El Jardín de JavaScript no prentende enseñar JavaScript. Se recomienda un conocimiento sobre el lenguaje para entender los temas tratados en esta guía. Con el fin de aprender los conceptos básicos del lenguaje, por favor diríjase a la excelente [guía](https://developer.mozilla.org/en/JavaScript/Guide) de los desarrolladores de Mozilla.

## Los autores

Esta guía es el trabajo de dos encantadores usuarios del foro Stack Overflow, [Ivo Wetzel](http://stackoverflow.com/users/170224/ivo-wetzel) (Escrito) y [Zhang Yi Jiang](http://stackoverflow.com/users/313758/yi-jiang) (Diseño).

## Colaboradores

* [Colaboradores](https://github.com/BonsaiDen/JavaScript-Garden/graphs/contributors)

## Hosting

JavaScript Garden es hospedado en GitHub, además [Cramer Development](http://cramerdev.com/) nos apoya con un sitio espejo en [JavaScriptGarden.info](http://javascriptgarden.info/).

## Licencia

El Jardín de JavaScript es publicado bajo la [licencia MIT](https://github.com/BonsaiDen/JavaScript-Garden/blob/next/LICENSE) y es hospedado en [GitHub](https://github.com/BonsaiDen/JavaScript-Garden). Si encuentra algún error o errata por favor publique [una incidencia](https://github.com/BonsaiDen/JavaScript-Garden/issues) o envie un pull request a nuestro repositorio. También nos puede encontrar en la [sala de chat de JavaScript](http://chat.stackoverflow.com/rooms/17/javascript) en Stack Overflow.

# Objetos

## Uso de objetos y propiedades

Todo en JavaScript actúa como un objeto, con las dos únicas excepciones de [null](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "core.undefined) y [undefined](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "core.undefined).

false.toString(); // 'false'

[1, 2, 3].toString(); // '1,2,3'

function Foo(){}

Foo.bar = 1;

Foo.bar; // 1

Un error muy común es el uso de literales númericos como objetos. Esto se debe a un error en el parser de JavaScript que intenta analizar la notación de puntos como un literal de punto flotante.

2.toString(); // lanza SyntaxError

Existe un par de soluciones que pueden utilizarse para hacer que los literales númericos actúen como objetos.

2..toString(); // el segundo punto es reconocido correctamente

2 .toString(); // observe el espacio a la izquierda del punto

(2).toString(); // el número 2 se evalúa primero

### Objetos como un tipo de datos

Los objetos en JavaScript también pueden ser utilizados como una Tabla Hash o conocido como [*Hashmap*](http://en.wikipedia.org/wiki/Hashmap) en inglés, consisten principalmente en nombres de propiedades, y asignándoles valores a éstas.

El uso de un objeto literal - con notación {} - puede crear un objeto plano. Este nuevo objeto [heredado](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.prototype) desde Object.prototype no posee [propiedades propias](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.hasownproperty) definidas.

var foo = {}; // un nuevo objeto vacío

// un nuevo objeto con la propiedad llamada 'test' con el valor 12

var bar = {test: 12};

### Acceso a las propiedades

Se puede acceder a las propiedades de un objeto de dos maneras, ya sea a través de la notación de punto o desde la notación de corchetes.

var foo = {name: 'kitten'}

foo.name; // kitten

foo['name']; // kitten

var get = 'name';

foo[get]; // kitten

foo.1234; // SyntaxError

foo['1234']; // ¡funciona!

Ambas notaciones son idénticas en su funcionamiento, la única diferencia es la notación de corchetes permite el ajuste dinámico de las propiedades, así como el uso de propiedades que de otro modo daría lugar a error de sintaxis.

### Eliminando propiedades

La única manera de eliminar una propiedad desde un objeto es usando el operador delete; establecer la propiedad a undefined o null solamente elimina el valor asociado a la propiedad, pero no la key (valor clave).

var obj = {

    bar: 1,

    foo: 2,

    baz: 3

};

obj.bar = undefined;

obj.foo = null;

delete obj.baz;

for(var i in obj) {

    if (obj.hasOwnProperty(i)) {

        console.log(i, '' + obj[i]);

    }

}

Los resultados de la salida son bar undefined y foo null - sólo baz ha sido removido y por lo tanto no aparece en la salida.

### Notación de Keys

var test = {

    'case': 'Soy una palabra clave y debo ser anotado como string',

    delete: 'Soy una palabra clave también' // lanza SyntaxError

};

Las propiedades de los objetos puede ser simbolizados como caracteres planos y como strings. Debido a otro mal diseño del parser de JavaScript, lo anterior es una excepción de SyntaxError antes de ECMAScript 5.

Este error se debe al eliminar una keyword; por lo tanto, debe ser anotado como un string literal para asegurarse que será interpretado correctamente por diversos motores de JavaScript.

## Prototipo

JavaScript no posee en sus características un sistema clásico de herencia, sino que utiliza un prototipo para esto.

Si bien a menudo se considera uno de los puntos débiles de JavaScript, el modelo de herencia prototipado es de hecho más poderoso que el modelo clásico. Por ejemplo, es bastante trivial construir un modelo clásico a partir del modelo prototipado, mientras que al contrario es una tarea mucho más difícil.

Debido al hecho que JavaScript es básicamente el único lenguaje que utiliza ampliamente la herencia prototipada, se necesita algo de tiempo para adaptarse a las diferencias entre los dos modelos.

La primera gran diferencia es que la herencia en JavaScript se realiza usando llamadas de cadenas de prototipo (prototype chains).

Nota: Simplemente usando Bar.prototype = Foo.prototype dará lugar a dos objetos que comparten el mismo prototipo. Por lo tanto, los cambios que se realicen en un objeto afectará al otro objeto, así, en la mayoría de los casos no es el efecto deseado.

function Foo() {

    this.value = 42;

}

Foo.prototype = {

    method: function() {}

};

function Bar() {}

// Asigna el prototipo de Bar como una nueva instancia de Foo

Bar.prototype = new Foo();

Bar.prototype.foo = 'Hello World';

// Asegura que el constructor sea Bar

Bar.prototype.constructor = Bar;

var test = new Bar() // crea una nueva instancia de Bar

// Resultado de cadena de prototipos (prototype chain)

test [instance of Bar]

    Bar.prototype [instance of Foo]

        { foo: 'Hello World' }

        Foo.prototype

            { method: ... }

            Object.prototype

                { toString: ... /\* etc. \*/ }

En el código anterior, el objeto test hereda de Bar.prototype y Foo.prototype; por lo tanto, tendrá acceso a la función method que se ha definido en Foo. También se tendrá acceso a a la propiedad value de la única instancia de Foo que compone su prototipo. Es importante tomar en cuenta que new Bar() no creará una nueva instancia de Foo, pero retornará lo asignado en su prototipo; de este modo, todas las instancias de Bar tendrán que compartir el mismo valor de la propiedad.

Nota: No utilice Bar.prototype = Foo, ya que no apunta al prototipo de Foo, sino al objeto de la función Foo. Así la cadena de prototipo cambiará a Function.prototype y no a Foo.prototype; Por lo tanto, el método no estará disponible en la cadena de prototipo.

### Búsqueda de propiedades

Cuando se accede a las propiedades de un objeto, JavaScript recorre la cadena de prototipo hacia arriba hasta encontrar la propiedad con el nombre solicitado.

Cuando se llega al final de la cadena - concretamente Object.prototype - y aún no se ha encontrado la propiedad especificada, se retornará un valor [undefined](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "core.undefined) en su lugar.

### La propiedad prototype

Aunque la propiedad prototype es usada por el lenguaje para construir la cadena de prototipos, es posible asignar cualquier valor. Aunque los tipos primitivos serán ignorados cuando se asigne en prototype.

function Foo() {}

Foo.prototype = 1; // no tendrá efecto

La asignación de objetos, como se muestra en el ejemplo anterior, funcionará, y permitirá la creación dinámica de cadena de prototipos.

### Rendimiento

El tiempo tomado en la búsqueda de propiedades es alta y la cadena de prototipo puede presentar un impacto negativo crítico en el rendimiento en partes del código. Además, si ha tratado de acceder a propiedades que no existen, esto provoca que se recorra la cadena de prototipo completa.

Además, al recorrer en [iteración](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.forinloop) las propiedades de un objeto , cada propiedad encontrada en la cadena de prototipo será enumerada.

### Extensión de prototipos nativos

Una mala característica que se suele utilizar para extender Object.prototype o cualquier otro prototipo construido.

Esta técnica es conocida en inglés como [monkey patching](http://en.wikipedia.org/wiki/Monkey_patch) y rompe la encapsulación del código. Si bien es utilizado en frameworks como [Prototype](http://prototypejs.org/), todavía no existen buenas razones para adoptarlo o integrarlo como tipos de dato o como funcionalidad no estándar.

La única razón coherente para extender un prototipo es para adaptarle nuevas características de los motores JavaScript más modernos; por ejemplo, [Array.forEach](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Array/forEach).

### En conclusión

Se debe entender por completo el módelo de herencia prototipado antes de escribir código complejo que lo utilice. Además, observe la longitud de la cadena de prototipo y modifíquela si es necesario para evitar posibles problemas de rendimiento. Con relación a los prototipos nativos, estos nunca deben ser extendidos a menos que sea para mantener la compatibilidad con nuevas características de JavaScript.

## hasOwnProperty

Con el fin de comprobar si un objeto posee una propiedad definida en sí mismo y no en algún lugar de su [cadena de prototipo](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.prototype), es necesario utilizar el método hasOwnProperty ya que todos los objetos herendan de Object.prototype.

Nota: No es suficiente con comprobar si una propiedad está definida. La propiedad bien podría existir, pero su valor sólo pasa a ser definido como undefined.

hasOwnProperty es la única utilidad en JavaScript que se ocupa de las propiedades y no las salta en la cadena de prototipo.

// Envenenamiento en Object.prototype

Object.prototype.bar = 1;

var foo = {goo: undefined};

foo.bar; // 1

'bar' in foo; // true

foo.hasOwnProperty('bar'); // false

foo.hasOwnProperty('goo'); // true

Sólo hasOwnProperty retornará el resultado correcto y esperado, esto es ensencial cuando se repite una iteración en las propiedades de cualquier objeto. No hay otra maner de excluir las propiedades que no están definidas en el mismo objeto, pero en alguna parte de su cadena de prototipo si.

### hasOwnProperty como propiedad

JavaScript no protege el nombre de la propiedad hasOwnProperty; de este modo, si existe la posibilidad de que un objeto tenga una propiedad con el mismo nombre, es necesario utilizar hasOwnProperty como propiedad externa con el fin de obtener resultados correctos.

var foo = {

    hasOwnProperty: function() {

        return false;

    },

    bar: 'Here be dragons'

};

foo.hasOwnProperty('bar'); // siempre devolverá false

// Utilice otro objeto con hasOwnProperty y llamelo con 'this' para asignarlo a foo

({}).hasOwnProperty.call(foo, 'bar'); // true

### En conclusión

Cuando se necesite comprobar la existencia de una propiedad en un objeto, hasOwnProperty es el único método para hacerlo. También se recomienda el uso de hasOwnProperty como parte de un [bucle](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.forinloop)for in, esto evitará errores desde extenciones de [prototipos](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.prototype) nativos.

## El bucle for in

Al igual que el operador in, el bucle for in también recorre sobre la cadena de prototipo cuando este se repite en una iteración en las propiedades de un objeto.

Nota: El bucle for in no se repetirá en cualquier propiedad que tenga atributos enumerables asignados a false; por ejemplo, la propiedad length de un array.

// Envenenamiento en Object.prototype

Object.prototype.bar = 1;

var foo = {moo: 2};

for(var i in foo) {

    console.log(i); // Imprime ambos bar y moo

}

Dado que no es posible cambiar el comportamiento del bucle for in en sí mismo, es necesario filtrar las propiedades internas no deseadas dentro del bucle, esto se hace mediante el uso del método [hasOwnProperty](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.hasownproperty) del Object.prototype.

Nota: Dado que for in siempre recorre por completo la cadena de prototipo, este se pondrá más lento con cada capa adicional que un objeto herede.

### Usando hasOwnProperty para filtrado

// Aún es el foo del código de arriba

for(var i in foo) {

    if (foo.hasOwnProperty(i)) {

        console.log(i);

    }

}

Está versión es la única forma correcta de uso. Esto se debe sólo al uso de hasOwnProperty que imprimirá moo. Cuando hasOwnProperty se omita, el código es propenso a errores en los casos de prototipos nativos - ej. Object.prototype - se ha extendedido.

Uno de los frameworks más usado que implementa estas funcionalidades es [Prototype](http://www.prototypejs.org/). Cuando el framework es incluido, el bucle for in que no utilicen hasOwnProperty no podrá garantizar que se interrumpa.

### En conclusión

Se recomienda utilizar siempre el uso de hasOwnProperty. Nunca debe suponer  
ningún entorno donde el código se ejecute, o si los prototipos nativos han sido extendidos o no.

# Funciones

## La declaración de funciones y expresiones

Las funciones en JavaScript son las primeras clases de objetos. Esto significa que se puede pasar como cualquier otro valor. Un uso común de está característica es pasar de una función anónima a otra, posiblemente una función asíncrona. Esto se conoce como callback.

### La declaración function

function foo() {}

La función anterior se [carga](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.scopes) así mismo antes de iniciar la ejecución del programa; por lo tanto, está disponible en todo el scope (ámbito) de la aplicación donde se ha definido, aunque hubiera sido llamado antes de definirse en el código.

foo(); // Funciona porque foo ha sido creado antes que este código se ejecute

function foo() {}

### La expresión function

var foo = function() {};

Este ejemplo asigna una función sin nombre y anónima a la variable foo.

foo; // 'undefined'

foo(); // Lanza TypeError

var foo = function() {};

Debido a la declaración de var, que carga el nombre de la variable foo antes de la ejecución real del inicio del código, foo ya estará definidido cuando se ejecute el script.

Pero se asigna sólo si ocurre en tiempo de ejecución, el valor de foo de forma predetermina es [undefined](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "core.undefined) antes de que el código se ejecute.

### Expresión nombre de función

Otro caso especial de asignación de nombre de funciones.

var foo = function bar() {

    bar(); // Funciona

}

bar(); // ReferenceError

Aquí bar no está disponible en el ámbito externo (scope), ya que la función sólo es asignada a foo; Sin embargo, dentro de bar si está disponible. Esto se debe a la forma en como trabaja la [resolución de nombres](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.scopes) en JavaScript, el nombre de la función esta siempre disponible en el ámbito local de la propia función.

## Cómo trabaja this

JavaScript tiene un concepto diferente sobre el nombre especial this referido a la mayoría de lenguajes de programación. Hay exactamente cinco formas distintas en donde es posible ver el valor de this dentro de lo posible en el lenguaje.

### El ámbito global (Global Scope)

this;

Cuando se utiliza this en el ámbito global, simplemente se refiere al objeto global.

### Llamar a una función

foo();

Aquí this se refiere al objeto global.

Nota ES5: En modo estricto (strict mode), global ya no existe. this tendrá el valor de undefined en este caso.

### Llamar a un método

test.foo();

En este ejemplo this se referiere a test.

### Llamar a un constructor

new foo();

Llamar a una función que esta precedida por la palabra clave new actúa como un [constructor](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.constructors). Dentro de la función, this se refiere al Objeto recién creado.

### Ajuste explícito de this

function foo(a, b, c) {}

var bar = {};

foo.apply(bar, [1, 2, 3]); // array que se apilará

foo.call(bar, 1, 2, 3); // resultados a = 1, b = 2, c = 3

Cuando se utiliza los métodos call o apply en Function.prototype, el valor de this dentro de la función llamada se ajustará explícitamente al primer argumento correspondiente a la llamada de la función.

Como resultado, el ejemplo anterior sobre los casos de métodos estos no se aplican, y this dentro de foo puede establecerse en bar.

Nota: this no puede ser usado para referirse a un objeto dentro de un Objeto literal. Así var obj = {me: this} no dará ninǵun resultado en me refiriendose a obj, ya que this sólo será obtenido por uno de los cincos casos enumerados.

### Errores comunes

Si bien en la mayoría de los casos esto tiene sentido, el primero puede cosiderarse como otro mal diseño del lenguaje, ya que nunca tiene un uso práctico.

Foo.method = function() {

    function test() {

        // this es establecido como un objeto global

    }

    test();

}

Un error común es que this dentro de test haga referencia a Foo, mientras que en realidad esto no es así.

Con el fin de acceder a Foo desde dentro de test es necesario crear una variable local dentro del método para referirse a Foo.

Foo.method = function() {

    var that = this;

    function test() {

        // Use that instead of this here

    }

    test();

}

that es justo un nombre normal, pero es comúnmente usado para referenciar a this de forma externa. En combinación con [closures](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.closures), esto puede ser también usado para pasar this como valor.

### Asignación de métodos

Otra cosa que no funciona en JavaScript son los alias en las funciones, es decir, asignar un método a una variable.

var test = someObject.methodTest;

test();

Debido al primer caso, test actúa como una función de llamada; por lo que this dentro de este no estará referido a someObject.

Mientras que la unión de this puede parecer una mala idea en un principio, esto es en realidad lo que hace trabajar a la [herencia de prototipo](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.prototype).

function Foo() {}

Foo.prototype.method = function() {};

function Bar() {}

Bar.prototype = Foo.prototype;

new Bar().method();

Cuando los métodos son llamados desde una instancia de Bar, this se referirá a una instancia.

## Closures y referencias

Una de las características más poderosas de JavaScript es la disponibilidad de closures (cerraduras), esto significa que los ámbitos siempre podrán ser accedidos por ámbitos externos donde fueron definidos. Dado que sólo el alcance es único en JavaScript en el [ámbito de la función](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.scopes), todas las funciones, por omisión, actúan como closures.

### Emulando variables privadas

function Counter(start) {

    var count = start;

    return {

        increment: function() {

            count++;

        },

        get: function() {

            return count;

        }

    }

}

var foo = Counter(4);

foo.increment();

foo.get(); // 5

En este caso, Counter retorna dos closures. La función increment y la función get. Ambas funciones mantienen el ámbito de la referencia de  
Counter y, por lo tanto, siempre accede a la variable count que fue definido en el ámbito.

### ¿Por qué las variables privadas trabajan?

Dado que no es posible referenciar o asignar ámbitos en JavaScript, no hay manera de acceder a la variable count desde fuera. Sólo existe una forma para interactuar con estos vía los dos closures.

var foo = new Counter(4);

foo.hack = function() {

    count = 1337;

};

El código anterior no ha cambiado la variable count en el ámbito de Counter, desde foo.hack no es definido en ese ámbito. En su lugar se creará - o se anulará - la variable global count.

### Closures dentro de bucles

Un error frecuente en el uso de closures dentro de bucles, es como si se tratará de copiar el valor del índice de la variable del bucle.

for(var i = 0; i < 10; i++) {

    setTimeout(function() {

        console.log(i);

    }, 1000);

}

El código anterior no tendrá como salida los números del 0 al 9, sino simplementemente se imprimirá el número 10 diez veces.

La función anónima hace referencia a i y se llama a console.log, el bucle for ya ha terminado y finalizo el valor de i a 10.

Con el fin de obtener el comportamiento deseado, es necesario crear una copia del valor de i.

### Evitando el problema de referencia

Con el fin de copiar el valor de la variable índice del bucle, lo mejor es utilizar un [contenedor anónimo](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.scopes).

for(var i = 0; i < 10; i++) {

    (function(e) {

        setTimeout(function() {

            console.log(e);

        }, 1000);

    })(i);

}

La función anónima externa llamará inmediatamente a i como su primer argumento y recibirá la copia del valor de i como parámetro de e.

La función anónima que se pasa a setTimeout ahora es una referencia a e, cuyo valor no han sido cambiados por el bucle.

No hay otra manera de lograr esto; se debe retornar una función desde el contenedor anónimo, que tendrá el mismo comportamiento que el código anterior.

for(var i = 0; i < 10; i++) {

    setTimeout((function(e) {

        return function() {

            console.log(e);

        }

    })(i), 1000)

}

## El objeto arguments

Cada ámbito de la función de JavaScript puede acceder a la variable especial arguments. Está variable contiene una lista de todos los argumentos que se pasan a la función.

Nota: En este caso arguments ya se ha definido dentro del ámbito de la función ya sea através de la sentencia var o como un parámetro formal, el objeto arguments no se creará.

El objeto arguments no es un Array. Si bien cuenta con la semántica de un array - concretamente la propiedad length - no hereda de Array.prototype y es de hecho un Objeto.

Debido a esto, no es posible usar los métodos estándar de los arrays como push, pop o slice en arguments. Mientras que la iteración es un simple bucle for que funciona muy bien, esto se convierte necesariamente en un Array real con el fin de utilizar los métodos de un Array.

### Conversión de un Array

El siguiente código devuelve un nuevo Array que contiene todos los elementos del objeto arguments.

Array.prototype.slice.call(arguments);

Esta conversión es lenta, no es recomendable usarlo en puntos criticos que afecten el rendimiento del código.

### Pasar Argumentos

El siguiente método es recomendado para pasar argumentos desde una función a otra.

function foo() {

    bar.apply(null, arguments);

}

function bar(a, b, c) {

    // do stuff here

}

Otro truco es utilizar tanto call y apply juntos para crear contenedores rápidos y consolidados.

function Foo() {}

Foo.prototype.method = function(a, b, c) {

    console.log(this, a, b, c);

};

// Crea una versión sin consolidar de "method"

// Se toma los parámetros: this, arg1, arg2...argN

Foo.method = function() {

    // Resultado: Foo.prototype.method.call(this, arg1, arg2... argN)

    Function.call.apply(Foo.prototype.method, arguments);

};

### Los parámetros formales y argumentos de índices

El objeto arguments crea las funciones de getter y setter para sus propiedades, así como parámetros formales de la función.

Como resultado, se ha cambiado el valor formal del parámetro también se cambio el valor de la propiedad correspondiente del objeto arguments, y al revés.

function foo(a, b, c) {

    arguments[0] = 2;

    a; // 2

    b = 4;

    arguments[1]; // 4

    var d = c;

    d = 9;

    c; // 3

}

foo(1, 2, 3);

### Mitos y verdades sobre el rendimiento

El objeto arguments es siempre creado con las dos únicas excepciones cuando es el caso en que declarado como un nombre dentro de la función o uno de los parámetros formales. No importa si se utiliza o no.

Ambos getters y setters son siempre creados; por lo tanto, con que casi no se tiene un impacto en el rendimiento en todo, especialemente no en el código real donde no es más que un simple acceso a las propiedades del objeto arguments.

Nota ES5: Estos getters y setters no son creados en modo estricto.

Sin embargo, hay casos en que se reducirá drásticamente el rendimiento en los motores modernos de JavaScript. Este es el caso del uso de arguments.callee.

function foo() {

    arguments.callee; // realiza algo con la función del objeto

    arguments.callee.caller; // y llama a la función del objeto

}

function bigLoop() {

    for(var i = 0; i < 100000; i++) {

        foo(); // Debería ser normalmente entre líneas...

    }

}

El código anterior, foo no puede estar sujeto a la [expansión en línea](http://en.wikipedia.org/wiki/Inlining) ya que se necesita saber acerca de sí mismo y la llamada. Esto no sólo denota los posibles beneficios de rendimiento que surgen con la expansión en línea, ya que también interrumpe la encapsulación ya que la función ahora puede ser dependiente de un contexto específico de llamada.

Es muy recomendable nunca hacer uso de arguments.callee o de cualquier de sus propiedades.

Nota ES5: En modo estricto, arguments.callee generará una excepción de TypeError ya que su uso ha quedado obsoleto.

## Constructores

Los constructores en JavaScript todavía son diferentes a los de otros lenguajes. Cualquier llamada que es precedida por la palabra new actua como un constructor.

Dentro del constructor - la función llama - el valor de this se refiere a un Objeto recién creado. El [prototipo](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.prototype) de este nuevo objeto se establece en el prototipo de la funcióno que es invocado como el constructor.

Si la función que se llama no tiene una sentencia return explícita, entonces implícitamente devuelve el valor de this - el nuevo objeto.

function Foo() {

    this.bla = 1;

}

Foo.prototype.test = function() {

    console.log(this.bla);

};

var test = new Foo();

La llamada de Foo por encima del constructor y establece el prototipo del objeto recién creado a Foo.prototype.

En caso explícito de la sentencia return de la función devuelva el valor especificado que la declaración, pero sólo si el valor devuelto es un Object.

function Bar() {

    return 2;

}

new Bar(); // a new object

function Test() {

    this.value = 2;

    return {

        foo: 1

    };

}

new Test(); // the returned object

Cuando una nueva keyword es omitidad, la función no devuelve un nuevo objeto.

function Foo() {

    this.bla = 1; // se establece en el objeto global

}

Foo(); // undefined

Auqnue el ejemplo anterior puede parecer que trabaja en algunos casos, debido a los trabajos de [this](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.this) en JavaScript, que usará el objeto global como valor de this.

### Fábricas

Con el fin de ser capaz de omitir un nuevo keyword, la función del tiene explícitamente devolver un valor.

function Bar() {

    var value = 1;

    return {

        method: function() {

            return value;

        }

    }

}

Bar.prototype = {

    foo: function() {}

};

new Bar();

Bar();

Ambos llamadas a Bar devuelven exactamente lo mismo, un reciente objeto creado que tiene como propiedad llamada el method, esto es un [Closure](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.closures).

También hay que notar que la llamada new Bar() no afecta al prototipo del objeto devuelto. Mientras que el prototipo se establece en el objeto recién creado, Bar nunca devuelve un nuevo objeto.

En el ejemplo anterior, no hay diferencia funcional entre usar y no usar el keyword new.

### Creación de nuevos objetos vía Factorias

Una recomendación a menudo es no utilizar new ya que su uso puede conducir a errores.

Con el fin de crear un nuevo objeto, uno bien debe utilizar una fábrica y un constructor para crear un nuevo objeto dentro de la fábrica.

function Foo() {

    var obj = {};

    obj.value = 'blub';

    var private = 2;

    obj.someMethod = function(value) {

        this.value = value;

    }

    obj.getPrivate = function() {

        return private;

    }

    return obj;

}

Aunque lo anterior es robuesto frente a la keyword new y, ciertamente hace que el uso de [variables privadas](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.closures) sea fácil, esto viene con algunas desventajas.

1. Se utiliza más memoria, ya que los objetos creados no comparten los métodos de un prototipo.
2. Con el fin de heredar de una fábrica se necesita copiar todos los métodos a otro objeto o poner todo en un prototipo de nuevo objeto.
3. La eliminación de una cadena de prototipo sólo por dejar la keyword new de alguna manera va en contra del espíritu del lenguaje.

### En conclusión

Mientras que se omite el keyword new podría dar a errores, no es ciertamente una razón para abandonar el uso de prototipos por completo. Al final todo se reduce a la solución que se adapta mejor a las necesidades de la aplicación, especialmente si es importante elegir un estilo específico en la creación de objetos y resistirse.

## Ámbitos y Namespaces

A pesar que JavaScript tiene una muy buena sintaxis de dos llaves para los bloques, está no es compatible con el soporte de ámbito de bloques; por lo que todo se deja al lenguaje con el ámbito de la función.

function test() { // un ámbito

    for(var i = 0; i < 10; i++) { // no es un ámbito

        // cuenta

    }

    console.log(i); // 10

}

Nota: Cuando no use una instrucción, de retorno o una función como argumento, la notación de {...} serán interpretadas como una declaración de bloques y no como un objeto literal. Esto, en conjunto con la [inserción automática de punto y coma](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "core.semicolon), puede conducir a errores sutiles.

Tampoco hay distintos namespaces en JavaScript, lo que significa que todo se define en un namespace global y compartido.

Cada vez que una variable es referenciada, JavaScript recorre hacia arriba a través de todos los ámbitos hasta encontrarlo. En este caso que llegue al ámbito global y todavía no ha encontrado el nombre solicitado, se generará un error ReferenceError.

### El terror de las variables globales

// script A

foo = '42';

// script B

var foo = '42'

Estos dos scripts no tienen el mismo efecto. El script A define una variable llamada foo en el ámbito global y el script B define foo en el actual ámbito.

Una vez más, esto no tiene el mismo efecto para todo, no usar var puede tener mayor implicación.

// ámbito global

var foo = 42;

function test() {

    // ámbito local

    foo = 21;

}

test();

foo; // 21

Dejando de lador la sentencia var dentro de la función test sobre escribiría el valor de foo. Si bien al principio puede parecer un gran cambio, se tiene miles de líneas de código en JavaScript y no se usaría var introduciendose en un horrible y difícil detección de errores.

// ámbito global

var items = [/\* some list \*/];

for(var i = 0; i < 10; i++) {

    subLoop();

}

function subLoop() {

    // ámbito de subLoop

    for(i = 0; i < 10; i++) { // falta la sentencia var

        // ¡realizar cosas asombrosas!

    }

}

El bucle externo terminará después de la primera llamada a subLoop, desde subLoop sobreescribe el valor global de i. Usando var para el segundo bucle for se hace fácil evitar este error. La sentencia var no debe nunca dejarse a menos que el efecto deseado es afectado por el ámbito exteriror.

### Variables locales

La única fuente para las variables locales en JavaScript son los parámetros de la [función](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.general) y variables que fueron declaradas vía la sentencia var.

// ámbito global

var foo = 1;

var bar = 2;

var i = 2;

function test(i) {

    // ámbito local de la función test

    i = 5;

    var foo = 3;

    bar = 4;

}

test(10);

Mientras foo y i son variables locales dentro del ámbitor de la función test, ela asignación de bar sobreescribe la variable global con el mismo nombre.

### Hoisting

La declaración de hoists en JavaScript. Esto significa que tanto la declaración de var y la función declarada se translada a la parte superior de su ámbito que lo contiene.

bar();

var bar = function() {};

var someValue = 42;

test();

function test(data) {

    if (false) {

        goo = 1;

    } else {

        var goo = 2;

    }

    for(var i = 0; i < 100; i++) {

        var e = data[i];

    }

}

El código anterior transforma antes de ejecutarse. JavaScript mueve la declaracione var aspi como las declaraciones de la función a la parte superior a lo más cercano del ámbito circundante.

// declaraciones var movidas aquí

var bar, someValue; // por omisión 'undefined'

// la función declarada es movida aquí también

function test(data) {

    var goo, i, e; // se pierde el ámbito del bloque movido aquí

    if (false) {

        goo = 1;

    } else {

        goo = 2;

    }

    for(i = 0; i < 100; i++) {

        e = data[i];

    }

}

bar(); // falla con TypeError desde bar sigue en 'undefined'

someValue = 42; // las asignaciones no se ven afectadas por hoisting

bar = function() {};

test();

La falta de alcance del bloque no sólo moverá la declaración var fuera de los bucles y su contenido, sino también hará que los resultados de ciertos constructores if no sean intuitivas.

En el código original la declaración de if si parecía modificar la variable global goo, mientras actualmente este modifica la variable local - después hoisting ha sido aplicado.

Sin el conocimiento acerca de hoisting, a continuación el código puede parecer un ReferenceError.

// comprueba si SomeImportantThing ha iniciado

if (!SomeImportantThing) {

    var SomeImportantThing = {};

}

Pero, por supuesto, lo anterior funciona debido a que la declaración var es movida a la parte superior del ámbito global.

var SomeImportantThing;

// otro código podría iniciar SomeImportantThing aqui, o no

// asegúrese de que está ahí

if (!SomeImportantThing) {

    SomeImportantThing = {};

}

### Name Resolution Order

All scopes in JavaScript, including the global scope, have the special name [this](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.this), defined in them, which refers to the current object.

Function scopes also have the name [arguments](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.arguments), defined in them, which contains the arguments that were passed to a function.

For example, when trying to access a variable named foo inside the scope of a function, JavaScript will lookup the name in the following order:

1. In case there is a var foo statement in the current scope, use that.
2. If one of the function parameters is named foo, use that.
3. If the function itself is called foo, use that.
4. Go to the next outer scope, and start with #1 again.

Note: Having a parameter called arguments will prevent the creation of the default arguments object.

### Namespaces

A common problem of having only one global namespace is the likeliness of running into problems where variable names clash. In JavaScript, this problem can easily be avoided with the help of anonymous wrappers.

(function() {

    // a self contained "namespace"

    window.foo = function() {

        // an exposed closure

    };

})(); // execute the function immediately

Unnamed functions are considered [expressions](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.general); so in order to being callable, they must first be evaluated.

( // evaluate the function inside the paranthesis

function() {}

) // and return the function object

() // call the result of the evaluation

There are other ways for evaluating and calling the function expression; which, while different in syntax, do behave the exact same way.

// Two other ways

+function(){}();

(function(){}());

### In Conclusion

It is recommended to always use an anonymous wrapper for encapsulating code in its own namespace. This does not only protect code against name clashes, but it also allows for better modularization of programs.

Additionally, the use of global variables is considered bad practice. Any use of them indicates badly written code that is prone to errors and hard to maintain.

# Arrays

## Iteración de un Array y sus propiedades

A pesar que los arrays en JavaScript son objetos, no existe un buena razón para usarlo en un [bucle for](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.forinloop) para una interación de este. De hecho, hay un número de buenas razones contra el uso de for in en arrays.

Nota: Los arrays de JavaScript no son arrays asociativos. JavaScript sólo tiene [objetos](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.general) para el mapeo de keys a valores. Y mientras que los arrays asociativos preservan el orden, los objetos no.

Dado que el bucle for in enumera todas las propiedades que están en una cadena de prototipo y la única manera para excluir estas propiedades es el uso de [hasOwnProperty](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "object.hasownproperty), ya que es veinte veces más lento que un bucle for normal.

### Iteración

Con el fin de obtener el mejor rendimiento cuando se repite la interación de arrays, es lo mejor hacer uso del clásico bucle for.

var list = [1, 2, 3, 4, 5, ...... 100000000];

for(var i = 0, l = list.length; i < l; i++) {

    console.log(list[i]);

}

Hay una captura adicional en el ejemplo anterior, que es el almacenamiento de la caché de longitud del array vía l = list.length.

Aunque la propiedad length es definida en el mismo array, todavía posee una sobrecarga para realizar la búsqueda en cada interación del bucle. Y mientras que los últimos motores de JavaScript pueden aplicar optimizaciones en este caso, no hay manera de saber si el ćodigo se ejecutará en uno de estos nuevos motores nuevos o no.

De hecho, dejando de lado el almacenamiento en caché puede resultar que el bucle inicie sólo la mitad de rápido que con la longitud de la caché.

### La propiedad length

Mientras que getter de la propiedad length simplemente retorne el número de elementos son contenidos en un array, el setter puede ser usado para truncar el array.

var foo = [1, 2, 3, 4, 5, 6];

foo.length = 3;

foo; // [1, 2, 3]

foo.length = 6;

foo; // [1, 2, 3]

La asignación de un menor número de longitud trunca al array, pero incrementando la longitud no tiene ningún efecto sobre el array.

### En conclusión

Para obtener el mejor rendimiento es recomendable siempre usar el bucle for y alamacenar en caché la propiedad length. El uso del bucle for in en un array es señal de un código mal escrito propenso a errores y un mal desempeño.

## El constructor Array

Desde el constructor Array es ambiguo en la forma en que ocupa sus párametros, es recomendable siempre el uso de arrays literales - la notación [] - cuando se crean nuevos arrays.

[1, 2, 3]; // Resultado: [1, 2, 3]

new Array(1, 2, 3); // Resultado: [1, 2, 3]

[3]; // Resultado: [3]

new Array(3); // Resultado: []

new Array('3') // Resultado: ['3']

En casos cuando sólo hay un argumento pasado al constructor del Array, y que el argumento es un Número, el contructor devolverá un array disperso con la propiedad length establecida al valor del argumento. Esto debe señalarse que la propiedad length sólo del nuevo array se establecerá de esa manera, los índices reales de la matriz no se iniciará.

var arr = new Array(3);

arr[1]; // undefined

1 in arr; // falso, el índice no se ha establecido

El comportamiento de poder establecer la longitud de un array inicial sólo es útil en algunos casos array, como la repetición de una cadena, en la que se evita el uso del código de bucle for.

new Array(count + 1).join(stringToRepeat);

### En conclusión

El uso de un constructor Array debe ser devuelto como sea posible. Los literales son definitivamente preferidos. Estos son más cortos y tienen una sintaxis más limpia; por lo tanto, también se incrementa la legibilidad del código.

# Tipos

## Igualdad y Comparación

JavaScript posee 2 maneras diferentes para comparar valores entre objetos para comprobar igualdad.

### El Operador de Igualdad

El operador de igualdad consiste en 2 signos es igual: ==

JavaScript utiliza tipado débil. Esto significa que el operador de igualdad obliga una conversión de tipos para poder compararlos.

""           ==   "0"           // false

0            ==   ""            // true

0            ==   "0"           // true

false        ==   "false"       // false

false        ==   "0"           // true

false        ==   undefined     // false

false        ==   null          // false

null         ==   undefined     // true

" \t\r\n"    ==   0             // true

La tabla anterior muestra los resultados de la conversión de tipos, éste es el motivo principal de por qué el uso de == es ampliamente considerado una mala práctica. Introduce errores difíciles de identificar debido a la complejidad de sus reglas de conversión.

Además, existe un impacto en el rendimiento cuando entra en juego la conversión de tipos; por ejemplo, una cadena debe ser convertida a número antes de poder ser comparada con otro número.

### El Operador de Igualdad Estricto

El operador de igualdad estricto consiste en tres signos es igual: ===:

Funciona exactamente igual que el operador de igualdad, excepto que el operador de igualdad estricto no utiliza conversión de tipos entre sus operandos.

""           ===   "0"           // false

0            ===   ""            // false

0            ===   "0"           // false

false        ===   "false"       // false

false        ===   "0"           // false

false        ===   undefined     // false

false        ===   null          // false

null         ===   undefined     // false

" \t\r\n"    ===   0             // false

Los resultados anteriores son mucho más claros y permiten una detección de errores temprana. Esto permite un código más sólido en cierto grado y también mejora el rendimiento en el caso que los operandos sean de tipos diferentes.

### Comparando Objetos

Aunque == como === son considerados operadores de igualdad, se comportan de maneras diferentes cuando al menos uno de sus operandos es Object.

{} === {};                   // false

new String('foo') === 'foo'; // false

new Number(10) === 10;       // false

var foo = {};

foo === foo;                 // true

En este caso, los dos operadores comparan por referencia y no por igualdad; esto es, comparan por la misma instancia del objeto, parecido al operador is en Python y la comparación entre punteros en C.

### En Conclusión

Es altamente recomendable usar sólo el operador de igualdad estricta. En los casos donde los tipos de datos necesitan ser convertidos, debe hacerse [explícitamente](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "types.casting) y no dejárselo a las complicadas reglas de conversión del lenguaje.

## The typeof Operator

The typeof operator (together with [instanceof](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "types.instanceof)) is probably the biggest design flaw of JavaScript, as it is near of being completely broken.

Although instanceof still has its limited uses, typeof really has only one practical use case, which does not happen to be checking the type of an object.

Note: While typeof can also be called with a function like syntax i.e. typeof(obj), this is not a function call. The two parenthesis will behave like normal and the return value will be used as the operand of the typeof operator. There is no typeof function.

### The JavaScript Type Table

Value               Class      Type

-------------------------------------

"foo"               String     string

new String("foo")   String     object

1.2                 Number     number

new Number(1.2)     Number     object

true                Boolean    boolean

new Boolean(true)   Boolean    object

new Date()          Date       object

new Error()         Error      object

[1,2,3]             Array      object

new Array(1, 2, 3)  Array      object

new Function("")    Function   function

/abc/g              RegExp     object (function in Nitro/V8)

new RegExp("meow")  RegExp     object (function in Nitro/V8)

{}                  Object     object

new Object()        Object     object

In the above table, Type refers to the value that the typeof operator returns. As can be clearly seen, this value is anything but consistent.

The Class refers to the value of the internal [[Class]] property of an object.

From the Specification: The value of [[Class]] can be one of the following strings. Arguments, Array, Boolean, Date, Error, Function, JSON, Math, Number, Object, RegExp, String.

In order to retrieve the value of [[Class]], one has to make use of the toString method of Object.prototype.

### The Class of an Object

The specification gives exactly one way of accessing the [[Class]] value, with the use of Object.prototype.toString.

function is(type, obj) {

    var clas = Object.prototype.toString.call(obj).slice(8, -1);

    return obj !== undefined && obj !== null && clas === type;

}

is('String', 'test'); // true

is('String', new String('test')); // true

In the above example, Object.prototype.toString gets called with the value of [this](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.this) being set to the object whose [[Class]] value should be retrieved.

ES5 Note: For convenience the return value of Object.prototype.toString for both null and undefined was changed from Object to Null and Undefined in ECMAScript 5.

### Testing for Undefined Variables

typeof foo !== 'undefined'

The above will check whether foo was actually declared or not; just referencing it would result in a ReferenceError. This is the only thing typeof is actually useful for.

### In Conclusion

In order to check the type of an object, it is highly recommended to use Object.prototype.toString because this is the only reliable way of doing so. As shown in the above type table, some return values of typeof are not defined in the specification; thus, they can differ across various implementations.

Unless checking whether a variable is defined, typeof should be avoided at all costs.

## The instanceof Operator

The instanceof operator compares the constructors of its two operands. It is only useful when comparing custom made objects. Used on built-in types, it is nearly as useless as the [typeof operator](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "types.typeof).

### Comparing Custom Objects

function Foo() {}

function Bar() {}

Bar.prototype = new Foo();

new Bar() instanceof Bar; // true

new Bar() instanceof Foo; // true

// This just sets Bar.prototype to the function object Foo

// But not to an actual instance of Foo

Bar.prototype = Foo;

new Bar() instanceof Foo; // false

### Using instanceof with Native Types

new String('foo') instanceof String; // true

new String('foo') instanceof Object; // true

'foo' instanceof String; // false

'foo' instanceof Object; // false

One important thing to note here is that instanceof does not work on objects that originate from different JavaScript contexts (e.g. different documents in a web browser), since their constructors will not be the exact same object.

### In Conclusion

The instanceof operator should only be used when dealing with custom made objects that originate from the same JavaScript context. Just like the [typeof](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "types.typeof) operator, every other use of it should be avoided.

## Type Casting

JavaScript is a weakly typed language, so it will apply type coercion wherever possible.

// These are true

new Number(10) == 10; // Number.toString() is converted

                      // back to a number

10 == '10';           // Strings gets converted to Number

10 == '+10 ';         // More string madness

10 == '010';          // And more

isNaN(null) == false; // null converts to 0

                      // which of course is not NaN

// These are false

10 == 010;

10 == '-10';

ES5 Note: Number literals that start with a 0 are interpreted as octal (Base 8). Octal support for these has been removed in ECMAScript 5 strict mode.

In order to avoid the above, use of the [strict equal operator](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "types.equality) is highly recommended. Although this avoids a lot of common pitfalls, there are still many further issues that arise from JavaScript's weak typing system.

### Constructors of Built-In Types

The constructors of the built in types like Number and String behave differently when being used with the new keyword and without it.

new Number(10) === 10;     // False, Object and Number

Number(10) === 10;         // True, Number and Number

new Number(10) + 0 === 10; // True, due to implicit conversion

Using a built-in type like Number as a constructor will create a new Number object, but leaving out the new keyword will make the Number function behave like a converter.

In addition, having literals or non-object values in there will result in even more type coercion.

The best option is to cast to one of the three possible types explicitly.

### Casting to a String

'' + 10 === '10'; // true

By prepending an empty string, a value can easily be casted to a string.

### Casting to a Number

+'10' === 10; // true

Using the unary plus operator, it is possible to cast to a number.

### Casting to a Boolean

By using the not operator twice, a value can be converted a boolean.

!!'foo';   // true

!!'';      // false

!!'0';     // true

!!'1';     // true

!!'-1'     // true

!!{};      // true

!!true;    // true

# Núcleo

## ¿Por qué no usar eval?

La función eval ejecuta un string como código JavaScript en el ámbito local.

var foo = 1;

function test() {

    var foo = 2;

    eval('foo = 3');

    return foo;

}

test(); // 3

foo; // 1

Pero eval sólo ejecutará en ámbito local cuando es llamado directamente y el nombre de la función llamada es eval.

var foo = 1;

function test() {

    var foo = 2;

    var bar = eval;

    bar('foo = 3');

    return foo;

}

test(); // 2

foo; // 3

El uso de eval debe evitarse a toda costa. El 99.9% de su "uso" puede lograrse sin su uso..

### eval disfrazado

Las funciones de [tiempo de espera](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "other.timeouts) setTimeout y setInterval pueden tomar un string como primer argumento. En este caso, el string siempre se ejecutará en el ámbito global ya que eval no ha sido llamado directamente.

### Problemas de seguridad

eval es también un problema de seguridad ya que ejecuta cualquier código enviado, y nunca debe usarse con strings que no se conozcan o tengan un origen no confiable.

### En conclusión

eval nunca debe ser usado, cualquier código que haga uso del mismo debe ser cuestionado en su funcionamiento, rendimiento y seguridad. En caso de que se necesite trabajar con eval, el diseño ha de ser cuestionado y no debe utilizarse en primer lugar, se debe usar un mejor diseño, que no requiera el uso de eval.

## undefined y null

JavaScript tiene dos valores distintos para nothing, el más útil de estos dos es undefined.

### El valor undefined

undefined es un tipo de dato con exactamente el mismo valor: undefined.

El lenguaje también define una variable global que tiene el valor de undefined, Esta variable es también llamada undefined. Sin embargo, esta variable no es una constante, ni es una palabra reservada del lenguaje. Esto significa que el valor puede ser sobreescrito fácilmente.

Nota ES5: undefined en ECMAScript 5 ya no es modificable en modo esstricto, pero su nombre todavía puede por ejemplo establecer una función con el nombre undefined.

Algunos ejemplos cuando el valor retorna undefined:

* Acceso a la variable global (sin modificar) undefined.
* Retorna implícitamente las funciones que no posean la sentencia return.
* Sentencia return que no retorna nada de forma explicíta.
* Búsquedas de propiedades inexistentes.
* Párametros de la función que no tienen ningún valor explicíto pasado.
* Cualquier valor que se estable en undefined.

### Manejar los cambios en el valor deChanges undefined

Dado que la variable undefined sólo tiene una copia del value de  
undefined, assigna un nuevo valor que no cambie el valor del tipo undefined.

Aún con el fin de comparar con el valor de undefined es necesario recuperar el valor de undefined primero.

Con el fin de proteger una posible sobreescritura en la variable undefined, una técnica común es agregar un párametro adicional a un [wrapper anónimo](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.scopes), que consiga ningún párametro que se le pase.

var undefined = 123;

(function(something, foo, undefined) {

    // undefined en el ámbito local

    // ahora hace referencia al valor

})('Hello World', 42);

Otra forma de lograrlo un mismo efecto es declarar dentro un wrapper.

var undefined = 123;

(function(something, foo) {

    var undefined;

    ...

})('Hello World', 42);

La única diferencia, es que está versión es de 4 bytes más que utiliza, y en caso se comprima y no hay otra declaración de 'var' dentro del wrapper anónimo.

### Uso de null

Mientras que undefined en el contexto del lenguaje JavaScript es muy usado en el sentido tradicional como null, el actual null (ambos literal y de un tipo) es más o menos que otro tipo de datos.

Es utilizado en algunos detalles internos de JavaScript (como declarar al final de un cadena de prototipo estableciendo Foo.prototype = null), pero en casi todos los casos, puede ser reemplazado por undefined.

## Automatic Semicolon Insertion

Although JavaScript has C style syntax, it does not enforce the use of semicolons in the source code, so it is possible to omit them.

JavaScript is not a semicolon-less language. In fact, it needs the semicolons in order to understand the source code. Therefore, the JavaScript parser automatically inserts them whenever it encounters a parse error due to a missing semicolon.

var foo = function() {

} // parse error, semicolon expected

test()

Insertion happens, and the parser tries again.

var foo = function() {

}; // no error, parser continues

test()

The automatic insertion of semicolon is considered to be one of biggest design flaws in the language because it can change the behavior of code.

### How it Works

The code below has no semicolons in it, so it is up to the parser to decide where to insert them.

(function(window, undefined) {

    function test(options) {

        log('testing!')

        (options.list || []).forEach(function(i) {

        })

        options.value.test(

            'long string to pass here',

            'and another long string to pass'

        )

        return

        {

            foo: function() {}

        }

    }

    window.test = test

})(window)

(function(window) {

    window.someLibrary = {}

})(window)

Below is the result of the parser's "guessing" game.

(function(window, undefined) {

    function test(options) {

        // Not inserted, lines got merged

        log('testing!')(options.list || []).forEach(function(i) {

        }); // <- inserted

        options.value.test(

            'long string to pass here',

            'and another long string to pass'

        ); // <- inserted

        return; // <- inserted, breaks the return statement

        { // treated as a block

            // a label and a single expression statement

            foo: function() {}

        }; // <- inserted

    }

    window.test = test; // <- inserted

// The lines got merged again

})(window)(function(window) {

    window.someLibrary = {}; // <- inserted

})(window); //<- inserted

Note: The JavaScript parser does not "correctly" handle return statements which are followed by a new line, while this is not neccessarily the fault of the automatic semicolon insertion, it can still be an unwanted side-effect.

The parser drastically changed the behavior of the code above. In certain cases, it does the wrong thing.

### Leading Parenthesis

In case of a leading parenthesis, the parser will not insert a semicolon.

log('testing!')

(options.list || []).forEach(function(i) {})

This code gets transformed into one line.

log('testing!')(options.list || []).forEach(function(i) {})

Chances are very high that log does not return a function; therefore, the above will yield a TypeError stating that undefined is not a function.

### In Conclusion

It is highly recommended to never omit semicolons; it is also advocated to keep braces on the same line with their corresponding statements and to never omit them for one single-line if / else statements. Both of these measures will not only improve the consistency of the code, but they will also prevent the JavaScript parser from changing its behavior.

# Otros

## setTimeout and setInterval

Since JavaScript is asynchronous, it is possible to schedule the execution of a function by using the setTimeout and setInterval functions.

Note: Timeouts are not part of the ECMAScript Standard. They are implemented as part of the [DOM](http://en.wikipedia.org/wiki/Document_Object_Model).

function foo() {}

var id = setTimeout(foo, 1000); // returns a Number > 0

When setTimeout gets called, it will return the ID of the timeout and schedule foo to run in approximately one thousand milliseconds in the future. foo will then get executed exactly once.

Depending on the timer resolution of the JavaScript engine that is running the code, as well as the fact that JavaScript is single threaded and other code that gets executed might block the thread, it is by no means a safe bet that one will get the exact delay that was specified in the setTimeout call.

The function that was passed as the first parameter will get called by the global object, which means that [this](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.this) inside the called function refers to that very object.

function Foo() {

    this.value = 42;

    this.method = function() {

        // this refers to the global object

        console.log(this.value); // will log undefined

    };

    setTimeout(this.method, 500);

}

new Foo();

Note: As setTimeout takes a function object as its first parameter, an often made mistake is to use setTimeout(foo(), 1000), which will use the return value of the call foo and not foo. This is, most of the time, a silent error, since when the function returns undefined setTimeout will not raise any error.

### Stacking Calls with setInterval

While setTimeout only runs the function once, setInterval - as the name suggests - will execute the function every X milliseconds, but its use is discouraged.

When code that is being executed blocks the timeout call, setInterval will still issue more calls to the specified function. This can, especially with small intervals, result in function calls stacking up.

function foo(){

    // something that blocks for 1 second

}

setInterval(foo, 100);

In the above code, foo will get called once and will then block for one second.

While foo blocks the code, setInterval will still schedule further calls to it. Now, when foo has finished, there will already be ten further calls to it waiting for execution.

### Dealing with Possible Blocking Code

The easiest solution, as well as most controllable solution, is to use setTimeout within the function itself.

function foo(){

    // something that blocks for 1 second

    setTimeout(foo, 100);

}

foo();

Not only does this encapsulate the setTimeout call, but it also prevents the stacking of calls and it gives additional control. foo itself can now decide whether it wants to run again or not.

### Manually Clearing Timeouts

Clearing timeouts and intervals works by passing the respective ID to clearTimeout or clearInterval, depending which set function was used in the first place.

var id = setTimeout(foo, 1000);

clearTimeout(id);

### Clearing all timeouts

Because there is no built-in method for clearing all timeouts and/or intervals, it is necessary to use brute force in order to achieve this functionality.

// clear "all" timeouts

for(var i = 1; i < 1000; i++) {

    clearTimeout(i);

}

There might still be timeouts that are unaffected by this arbitrary number; therefore, is is instead recommended to keep track of all the timeout IDs, so they can be cleared specifically.

### Hidden use of eval

setTimeout and setInterval can also take a string as their first parameter. This feature should never be used because it internally makes use of eval.

Note: Since the timeout functions are not specified by the ECMAScript standard, the exact workings when a string is passed to them might differ in various JavaScript implementations. For example, Microsoft's JScript makes use of the Function constructor in place of eval.

function foo() {

    // will get called

}

function bar() {

    function foo() {

        // never gets called

    }

    setTimeout('foo()', 1000);

}

bar();

Since eval is not getting called [directly](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "core.eval) in this case, the string passed to setTimeout will get executed in the global scope; thus, it will not use the local variable foo from the scope of bar.

It is further recommended to not use a string for passing arguments to the function that will get called by either of the timeout functions.

function foo(a, b, c) {}

// NEVER use this

setTimeout('foo(1,2, 3)', 1000)

// Instead use an anonymous function

setTimeout(function() {

    foo(1, 2, 3);

}, 1000)

Note: While it is also possible to use the syntax setTimeout(foo, 1000, 1, 2, 3), it is not recommended, as its use may lead to subtle errors when used with [methods](http://bonsaiden.github.io/JavaScript-Garden/es/" \l "function.this).

### In Conclusion

Never should a string be used as the parameter of setTimeout or setInterval. It is a clear sign of really bad code, when arguments need to be supplied to the function that gets called. An anonymous function should be passed that then takes care of the actual call.

Furthermore, the use of setInterval should be avoided because its scheduler is not blocked by executing JavaScript.

Copyright ©. Built with |[Node.js](http://nodejs.org/)using a[jade](https://github.com/visionmedia/jade/)template.